# *Java Programming Language*

*SL-275/SL-276*

Student Guide With Instructor Notes

Sun microsystems

Enterprise Services
MS BRM01
500 Eldorado Blvd.
Broomfield, Colorado 80021
U.S.A.

Revision B.1, January 1999

Please
Recycle

Adobe PostScript

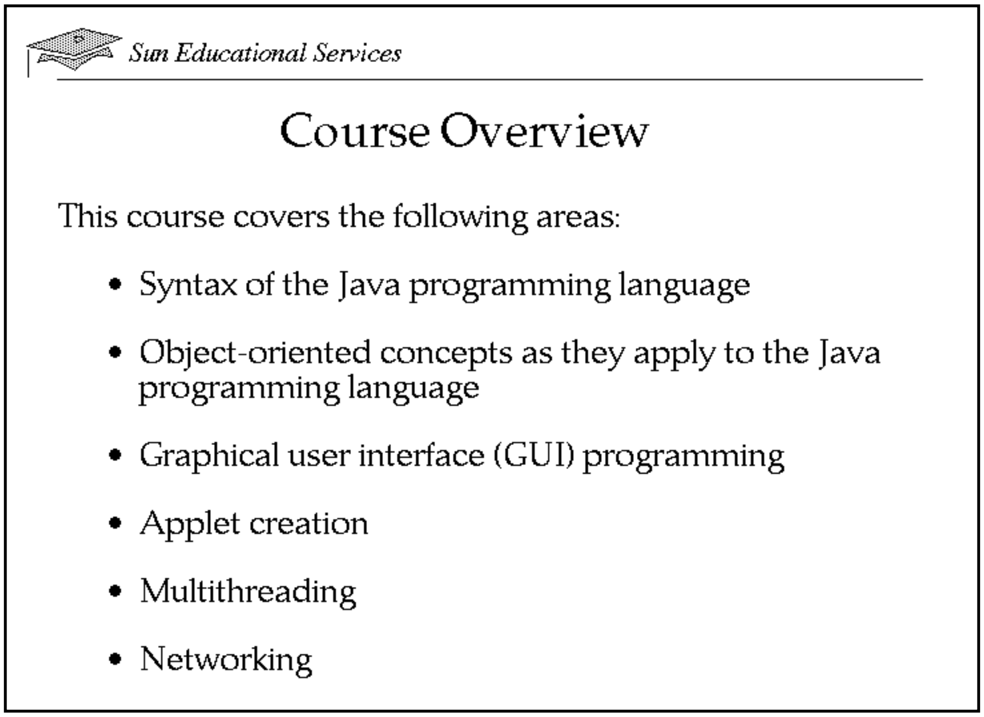
# Contents

# About the Course

## Course Goal

The main goal of Java™ applications programming is to provide you with the knowledge and skills necessary for object-oriented programming of advanced Java applications and applets. You will learn Java programming language syntax and object-oriented concepts, as well as the more sophisticated features of the Java runtime environment such as support for graphical user interfaces (GUIs), multithreading, and networking. This course covers prerequisite knowledge for the passage of the Sun Certified Java Programmer and Sun Certified Java Developer examinations.

- ✓ ***Use this module to get the students excited about this course.***
- ✓ ***With regard to the overheads: To avoid confusion among the students, it is very important to tell them that the page numbers on the overheads have no relation to the page numbers in their course materials. They should use the title of each overhead as a reference.***
- ✓ ***The strategy provided by the "About This Course" is to introduce students to the course before they introduce themselves to you and one another. By familiarizing them with the content of the course first, their introductions will have more meaning in relation to the course prerequisites and objectives.***
- ✓ ***Use this introduction to the course to determine how well students are equipped with the prerequisite knowledge and skills.***


Sun Educational Services

## Course Overview

This course covers the following areas:

- Syntax of the Java programming language
- Object-oriented concepts as they apply to the Java programming language
- Graphical user interface (GUI) programming
- Applet creation
- Multithreading
- Networking

## *Course Overview*

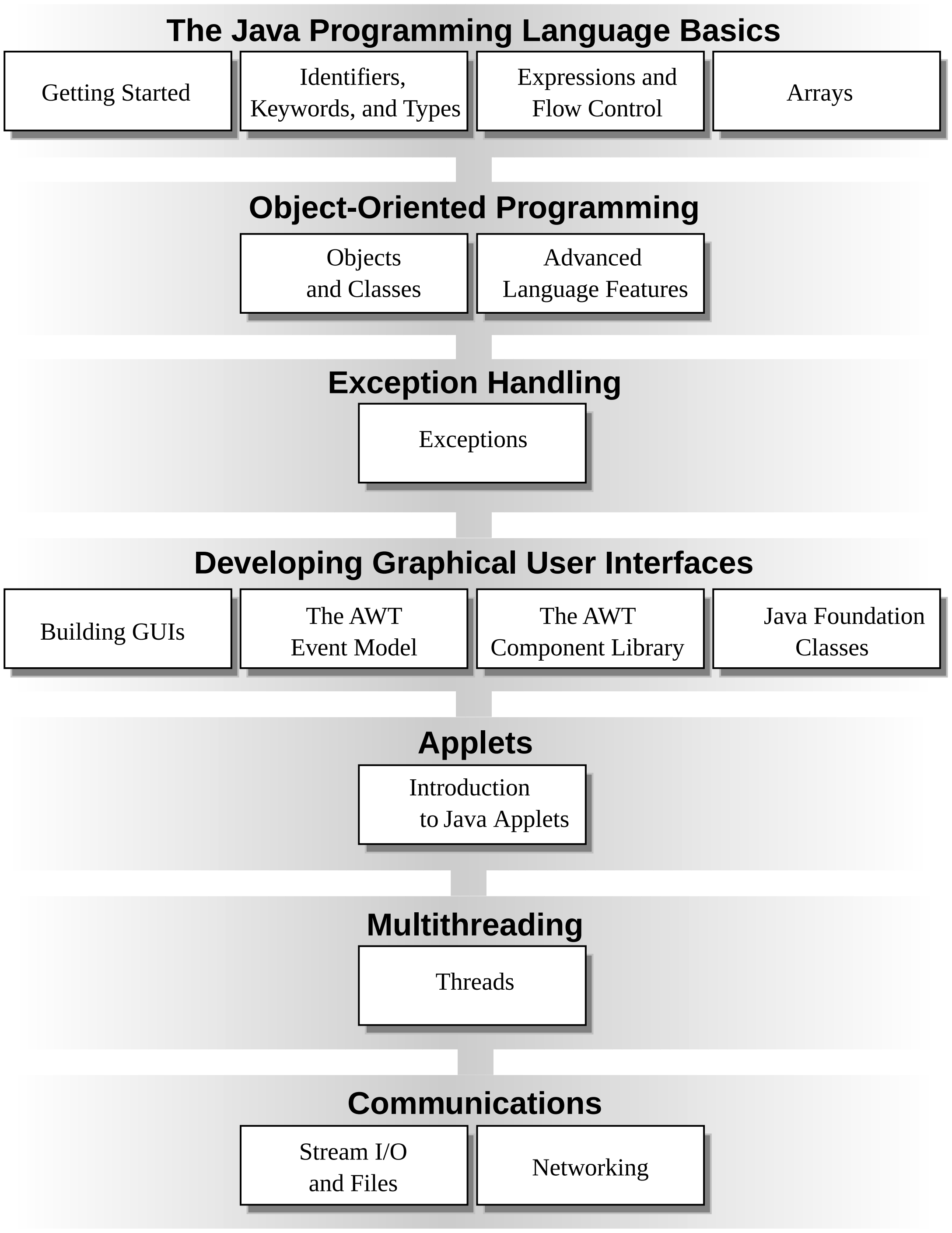
This course first discusses the Java runtime environment and the syntax of the Java programming language. The course then covers object-oriented concepts as they apply to the language. As the course progresses, advanced features of the Java platform are discussed.

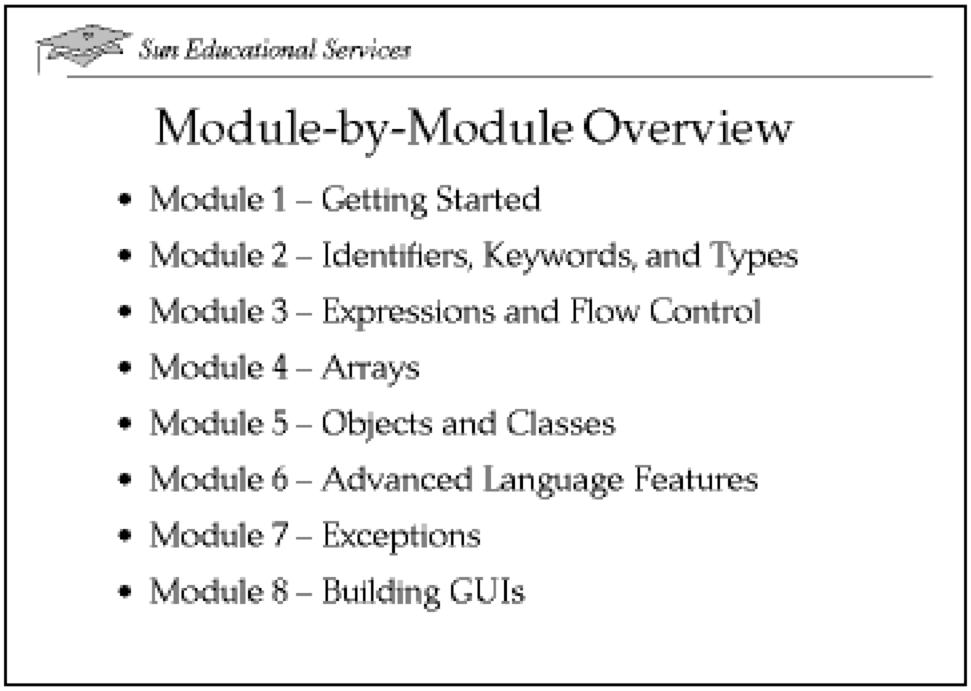
The audience for this course is those familiar with implementing elementary programming concepts using the Java programming language or other languages. This is the follow-up course to *Java Programming for Non-Programmers* (SL-110).

While the Java programming language is operating system independent, the GUI that it produces can be dependent on the operating system on which the code is executed. The course material code examples were run on a Solaris™ operating environment; therefore, the graphics in this manual have a Motif GUI. The same code can produce a Microsoft Windows 95 GUI if run on the Windows 95 operating system. The contents of this course are applicable to all Java operating system ports.

## *Course Map*

Each module begins with a course map that enables you to see what you have accomplished and where you are going in reference to the course goal. A complete map of this course is shown below.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking


Sun Educational Services

## Module-by-Module Overview

- Module 1 – Getting Started
- Module 2 – Identifiers, Keywords, and Types
- Module 3 – Expressions and Flow Control
- Module 4 – Arrays
- Module 5 – Objects and Classes
- Module 6 – Advanced Language Features
- Module 7 – Exceptions
- Module 8 – Building GUIs

## *Module-by-Module Overview*

- Module 1 – Getting Started

  This module provides a general overview of the Java programming language and its main features, and introduces Java applications. This module also reviews the concepts of classes and packages and some of the more commonly used Java packages.

- Module 2 – Identifiers, Keywords, and Types

  The Java programming language contains many programming constructs similar to the C language. This module provides a general overview of the constructs available and the general syntax required for each construct. It also introduces the basic object-oriented approach to data association using aggregate data types.

- Module 3 – Expressions and Flow Control

  This module looks at expressions including operators and the syntax of Java program control.

## Module-by-Module Overview

- Module 4 – Arrays

  This module covers how Java arrays are declared, created, initialized, and copied.

- Module 5 – Objects and Classes

  This module takes the introduction to Java object concepts in Module 2 to the next level, including a discussion on overloading, overriding, subclassing, and constructors.

- Module 6 – Advanced Language Features

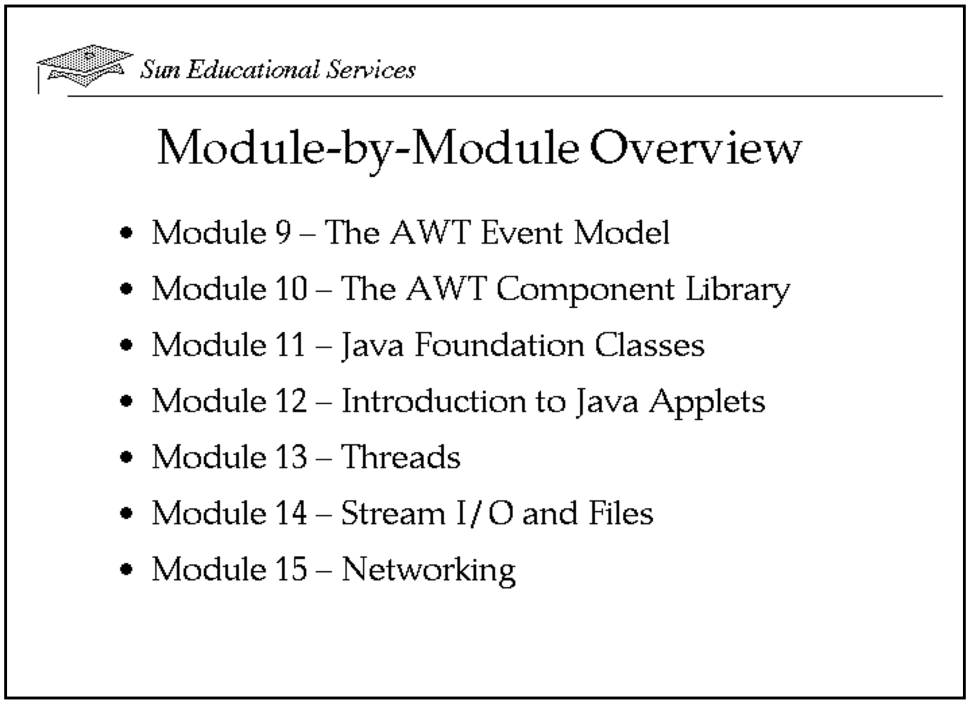
  This module completes the Java object-oriented programming model introduced in Module 5 and includes some of the new features of JDK 1.1—deprecation and inner classes. This module also introduces the concept of collections new to Java Development Kit (JDK™) 1.2.

- Module 7 – Exceptions

  Exceptions provide the Java programmer with a mechanism for trapping errors at runtime. This module explores both predefined and user-defined exceptions.

- Module 8 – Building GUIs

  All graphical user interfaces in the Java programming language are built on the concept of frames and panels. This module introduces layout management and containers.


Sun Educational Services

## Module-by-Module Overview

- Module 9 – The AWT Event Model
- Module 10 – The AWT Component Library
- Module 11 – Java Foundation Classes
- Module 12 – Introduction to Java Applets
- Module 13 – Threads
- Module 14 – Stream I/O and Files
- Module 15 – Networking

## *Module-by-Module Overview*

- Module 9 – The AWT Event Model

  One of the most significant changes in the Java programming language for Version 1.1 is the way events are sent to and from Java components. This module covers the differences between the JDK 1.0 and 1.1 event models, and demonstrates how to write compact event handlers.

- Module 10 – The AWT Component Library

  This module covers the abstract windowing toolkit (AWT) components you will use to create Java GUIs. In this module, you will see how Java AWT components and the 1.1 event model work together.

## Module-by-Module Overview

- Module 11 - Java Foundation Classes

  This module highlights one of the major features of JDK 1.2, the Java Foundation Classes (JFC). This module explains Swing components and their pluggable look and feel architecture. A basic Swing application and other aspects of JFC such as two-dimensional graphics, accessibility, and drag and drop are also introduced.

- Module 12 – Introduction to Java Applets

  This module illustrates the difference between applet and application development. Audio enhancements of JDK 1.2 are also introduced.

- Module 13 – Threads

  Threads are a complex topic; this module explains threading as it relates to the Java programming language and introduces a straightforward example of thread communication and synchronization.

- Module 14 – Stream I/O and Files

  This module explains the classes available for reading and writing both data and text files, and introduces object serialization.

- Module 15 – Networking

  This module introduces the Java network programming package and demonstrates a Transmission Control Protocol/Internet Protocol (TCP/IP) client-server model.

## *Course Objectives*

Upon completion of this course, you should be able to

- Describe key language features
- Compile and run a Java application
- Understand and use the online hypertext Java technology documentation
- Describe language syntactic elements and constructs
- Understand the object-oriented paradigm and use object-oriented features of the language
- Understand and use exceptions
- Develop a graphical user interface
- Describe the Java technology platform's Abstract Window Toolkit from which GUIs are built
- Develop a program to take input from a GUI
- Understand event handling
- Describe the main features of Swing
- Develop Java applets
- Read and write to files and other data sources
- Perform input/output (I/O) to all sources without the use of a GUI
- Understand the basics of multithreading
- Develop multithreaded Java applications and applets
- Develop Java client and server programs using TCP/IP and user datagram protocol (UDP)

## *Skills Gained by Module*

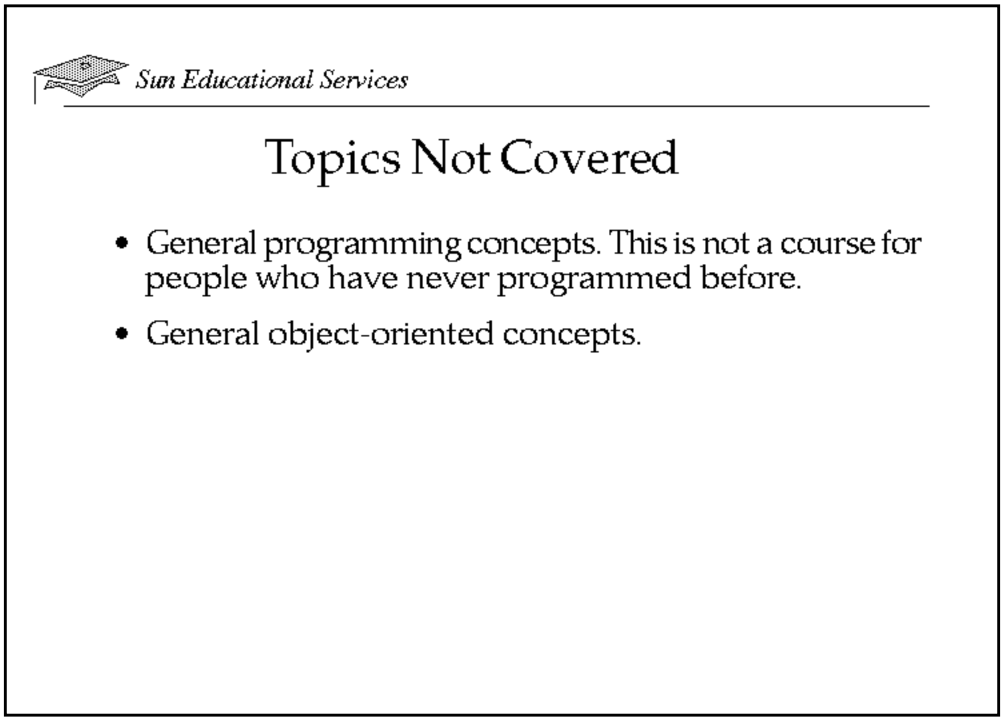
The skills for *Java Applications Programming* are shown in column 1 of the matrix below. The black boxes indicate the main coverage for a topic; the gray boxes indicate the topic is briefly discussed.

| Skills Gained | Module | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| Describe key language features | ■ | | | | | | | | | | | | | | |
| Compile and run a Java application | ■ | | | | | | | | | | | | | | |
| Understand and use the online hypertext Java technology documentation | ■ | | | | | | | | | | | | | | |
| Describe language syntactic elements and constructs | | ■ | ■ | ■ | ■ | ■ | | | | | | | | | |
| Understand the object-oriented paradigm and use object-oriented features | | ▨ | | | ■ | ■ | | | | | | | | | |
| Understand and use exceptions | | | | | | | ■ | | | | | | | | |
| Develop a GUI | | | | | | | | ■ | ■ | ■ | ■ | ▨ | | | |
| Describe the Java technology platform's Abstract Window Toolkit from which GUIs are built | | | | | | | | ■ | | ■ | ■ | | | | |
| Develop a program to take input from a graphical user interface | | | | | | | | | ■ | | ■ | | | | |
| Understand event handling | | | | | | | | | | ■ | ■ | | | | |
| Describe the main features of Swing | | | | | | | | | | | ■ | | | | |
| Develop Java applets | | | | | | | | | | | | ■ | | | |
| Understand the basics of multithreading | | | | | | | | | | | | | ■ | | |
| Develop multithreaded Java applications and applets | | | | | | | | | | | | | ■ | | |
| Perform I/O to all sources without the use of a GUI | | | | | | | | | | | | | | ■ | |
| Read and write to files and other data sources | | | | | | | | | | | | | | ■ | |
| Develop Java client and server programs using TCP/IP and UDP | | | | | | | | | | | | | | | ■ |

## Guidelines for Module Pacing

The table below provides a rough estimate of pacing for this course.

| **Module** | **Day 1** | **Day 2** | **Day 3** | **Day 4** | **Day 5** |
|---|---|---|---|---|---|
| About This Course | A.M. | | | | |
| Module 1 – Getting Started | A.M. | | | | |
| Module 2 – Identifiers, Keywords, and Types | A.M. | | | | |
| Module 3 – Expressions and Flow Control | P.M. | | | | |
| Module 4 – Arrays | P.M. | | | | |
| Module 5 – Objects and Classes | | A.M. | | | |
| Module 6 – Advanced Language Features | | P.M. | | | |
| Module 7 – Exceptions | | | A.M. | | |
| Module 8 – Building GUIs | | | A.M. | | |
| Module 9 – The AWT Event Model | | | P.M. | | |
| Module 10 – The AWT Component Library | | | | A.M. | |
| Module 11 – Java Foundation Classes | | | | A.M. | |
| Module 12 – Introduction to Java Applets | | | | P.M. | |
| Module 13 – Threads | | | | | A.M. |
| Module 14 – Stream I/O and Files | | | | | P.M. |
| Module 15 – Networking | | | | | P.M. |


Sun Educational Services

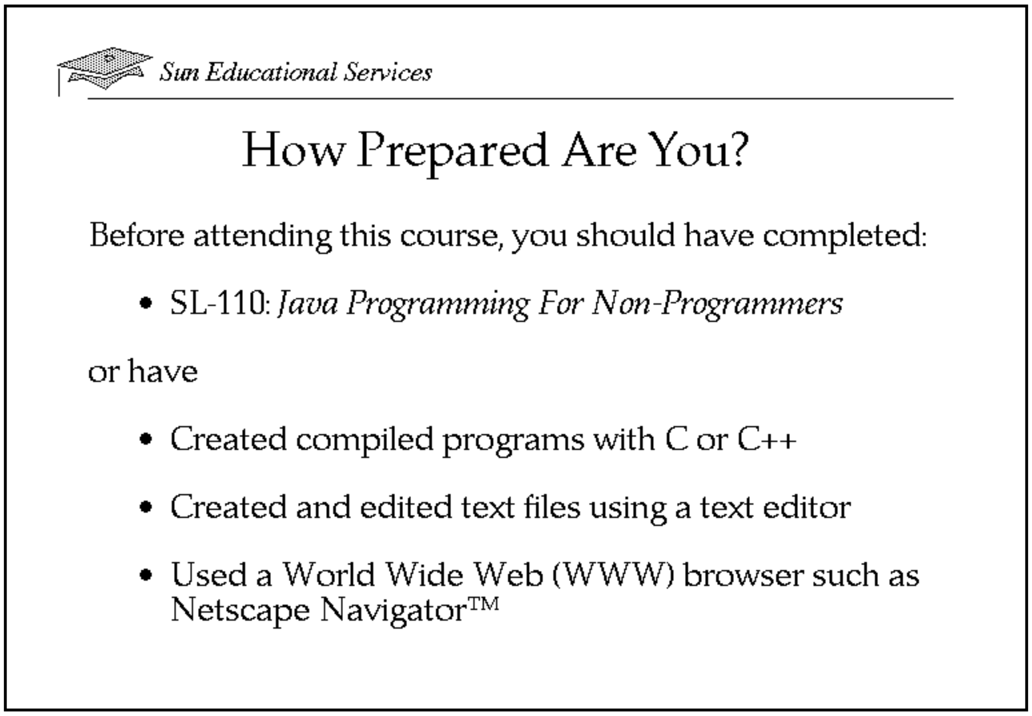
Topics Not Covered

- General programming concepts. This is not a course for people who have never programmed before.
- General object-oriented concepts.

## *Topics Not Covered*

This course does not cover the topics shown on the above overhead. Many of the topics listed on the overhead are covered in other courses offered by Sun Educational Services:

- Object-oriented concepts – Covered in OO-100: *Object-Oriented Technology and Concepts*.
- Object-oriented design and analysis – Covered in OO-120: *Object-Oriented Design and Analysis*.
- General programming concepts – Covered in SL-110: *Java Programming for Non-Programmers*.


Sun Educational Services

How Prepared Are You?

Before attending this course, you should have completed:

- SL-110: *Java Programming For Non-Programmers*

or have

- Created compiled programs with C or C++
- Created and edited text files using a text editor
- Used a World Wide Web (WWW) browser such as Netscape Navigator™

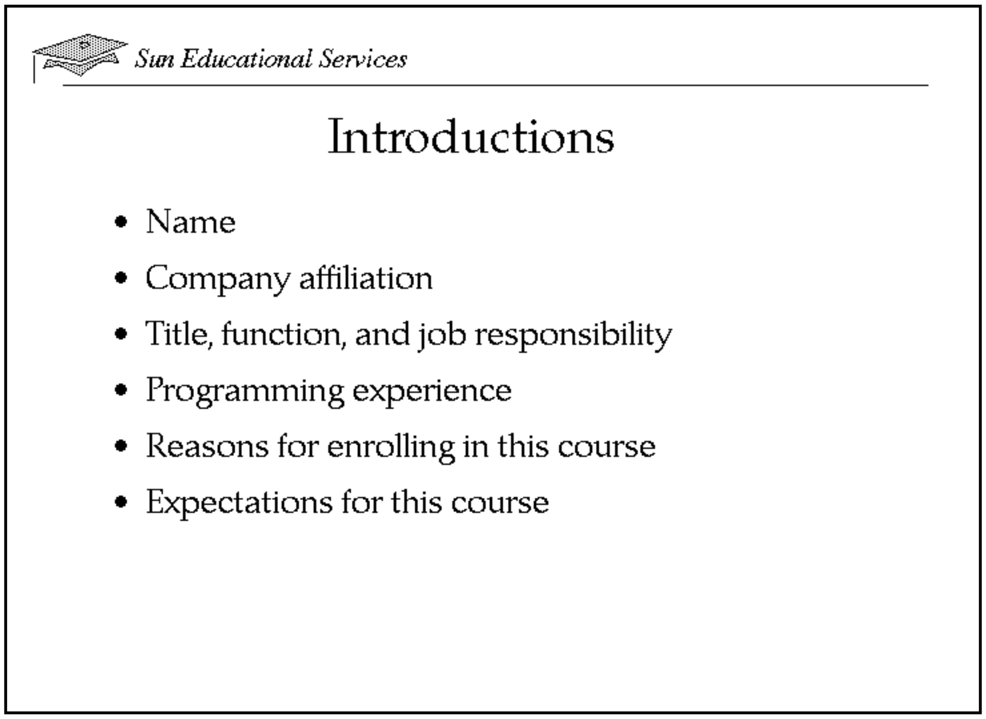
## *How Prepared Are You?*

Before attending this course, you should have completed:

- SL-110: *Java Programming For Non-Programmers*

or have

- Created compiled programs with C or C++
- Created and edited text files using `vi` or the OpenWindows™ text editor
- Used a World Wide Web (WWW) browser such as Netscape Navigator™

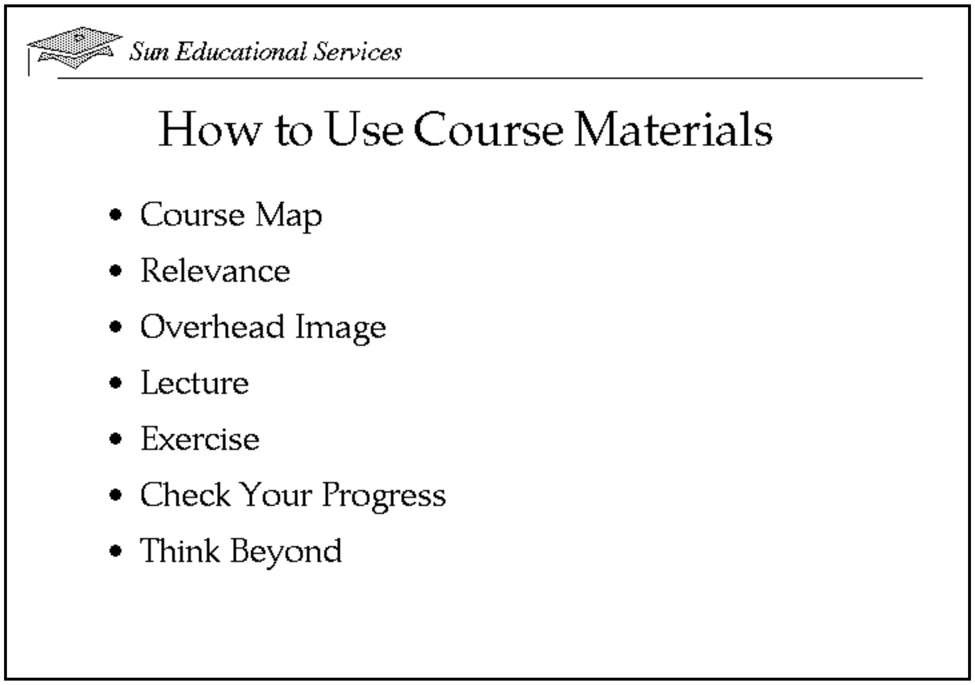

Sun Educational Services

## Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Programming experience
- Reasons for enrolling in this course
- Expectations for this course

## *Introductions*

Now that you have been introduced to the course, introduce yourself to each other and to the instructor, addressing the items shown on the above overhead.


Sun Educational Services

## How to Use Course Materials

- Course Map
- Relevance
- Overhead Image
- Lecture
- Exercise
- Check Your Progress
- Think Beyond

## *How to Use Course Materials*

To enable you to succeed in this course, these course materials employ a learning model that is composed of the following components:

- **Course Map** – Each module begins with an overview of the content so you can see how the module fits into your overall course goal.
- **Relevance** – The relevance section for each module provides scenarios or questions that introduce you to the information contained in the module and encourage you to think about how the module content relates to your interest in Java applications programming.
- **Overhead Image** – Reduced overhead images for the course are included in the course materials to help you easily follow where the instructor is at any point in time. Overheads do not appear on every page.

## *How to Use Course Materials*

- **Lecture** – The instructor will present information specific to the topic of the module. This information will help you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** – Lab exercises will give you the opportunity to practice your skills and apply the concepts presented in the lecture. The example code presented in the lecture should help you in completing the lab exercises.
- **Check Your Progress** – Module objectives are restated, sometimes in question format, so that before moving on to the next module you are sure that you can accomplish the objectives of the current module.
- **Think Beyond** – Thought-provoking questions are posed to help you apply the content of the module or predict the content in the next module.

# *Course Icons and Typographical Conventions*

The following icons and typographical conventions are used in this course to represent various training elements and alternative learning resources.

## *Course Icons*

**Additional resources –** Indicates additional reference materials are available.

**Discussion –** Indicates a small-group or class discussion on the current topic is recommended at this time.

**Exercise objective –** Indicates the objective for the lab exercises that follow. The exercises are appropriate for the material being discussed.

---

**Note –** Additional important, reinforcing, interesting or special information.

---

---

**Caution –** A potential hazard to data or machinery.

---

# *Course Icons and Typographical Conventions*

## *Typographical Conventions*

`Courier` is used for the names of commands, files, and directories, as well as on-screen computer output. For example:

> Use `ls -al` to list all files.
> `system% You have mail.`

**`Courier bold`** is used for characters and numbers that you type. For example:

```
system% su
Password:
```

*`Courier italic`* is used for variables and command-line placeholders that are replaced with a real name or value. For example:

> To delete a file, type `rm` *`filename`*.

*Palatino italics* is used for book titles, new words or terms, or words that are emphasized. For example:

> Read Chapter 6 in *User's Guide*.
> These are called *class* options
> You *must* be root to do this.

## *Solaris and Microsoft Windows Setup Differences*

One difference between the Solaris port of the Java Development Kit (JDK) and the Windows 95 and Windows NT port is the environment variables' setup. When using Microsoft Windows95/NT, set up the environment variables in the `autoexec.bat` file, using semicolons instead of colons. The entries should look like this:

```
set JAVA_HOME=c:\jdk1.2\

set PATH=%PATH%;%JAVA_HOME%\bin
```

The variables in Windows NT should be set by using Control Panel ➤ System Properties ➤ Environment window.

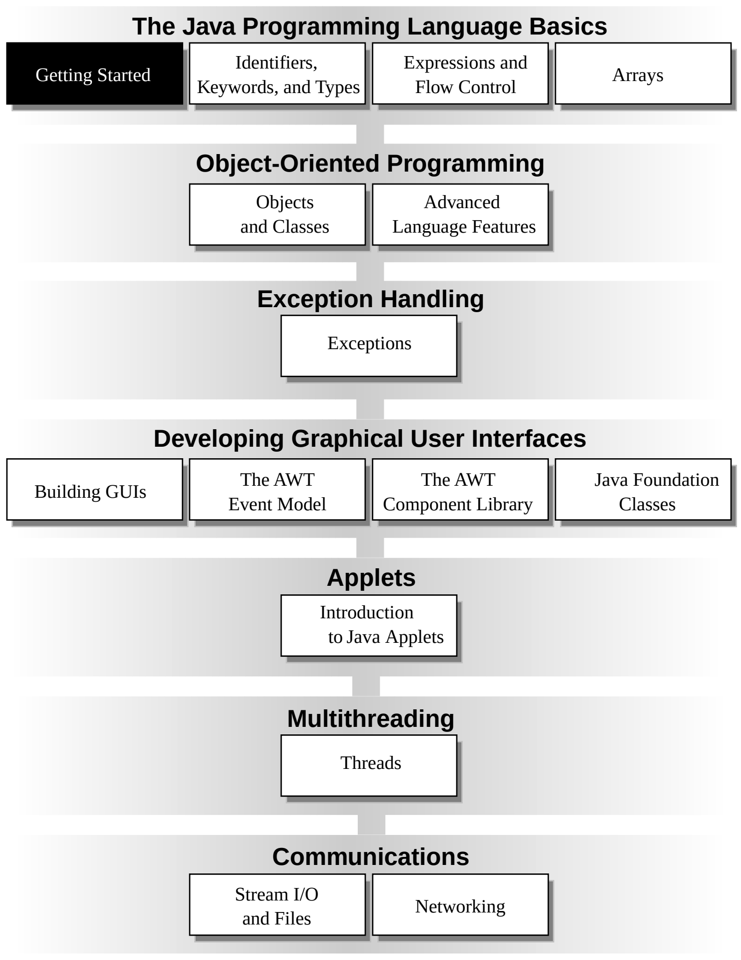
It is also important to know that when running the JDK from the command-line prompt in a Windows 95/NT environment, it must be run from a Windows 95/NT DOS emulation window that supports the 32-bit architecture. Files must be able to have long extensions; the 8.3 file format will not support Java source files.

Platform-specific differences that appear in this course material are due to the underlying native code that deals with calls from the Java programming language. Platform-specific difference regarding the `Scrollbar` also appears in this course material. The Windows 95/NT implementation of the Scrollbar object accounts for slider size when setting the maximum value of the Scrollbar. For example, if the Scrollbar is set for a minimum size of 0 and a maximum size of 10, and the slider size is 5, the range of the slider will be limited to 5. In the Solaris implementation, the slider size is added to the maximum size regaining full slider movement.

# Getting Started 1

## Course Map

This module provides a general overview of the Java programming language including the Java Virtual Machine, garbage collection, and security features.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following questions are relevant to the material presented in this module:

- Is the Java programming language a complete language or is it just useful for writing programs for the Web?

✓ ***There might still be some attitudes that the Java programming language is for Web programs. There are certain features (such as better security) that make this language ideal for the Web, but it is a complete object-oriented programming language useful for mainstream applications as well.***

- Why do you need another programming language?
- How does the Java platform improve on other language platforms?

✓ ***The Java platform was written with security in mind from the conception of the language thus security issues are at the core of the platform. Other languages addressed security issues later in the development cycle, and might still have security leaks.***

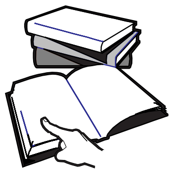
✓ ***Other issues you might discuss are platform-independence and ease of use. The Java programming language is easier to use than other programming languages (such as C or C++) because it has automatic garbage collection, stronger typing of objects and variables, and removal of problem areas such as pointers. You might want to use this question as an introduction to what will be discussed in this module.***

## Objectives

Upon completion of this module, you should be able to

- Describe key features of Java programming language
- Describe the Java virtual machine's (JVM) function
- Describe how garbage collection works
- List the three tasks performed by the Java platform that handle code security
- Define the terms *class, package, applet*, and *application*
- Write, compile, and run a simple Java application
- Use the Java technology application programming interface on-line documentation to identify information found in the `java.lang` package

## References



**Additional resources –** The following references can provide additional details on the topics discussed in this module:

- Lindholm and Yellin. 1997. *The Java Virtual Machine Specification.* Addison-Wesley.
- Yellin, Frank. Low-Level Security in Java, white paper. [Online]. Available: `http://www.javasoft.com/sfaq/verifier.html.`

## What Is the Java Programming Language?

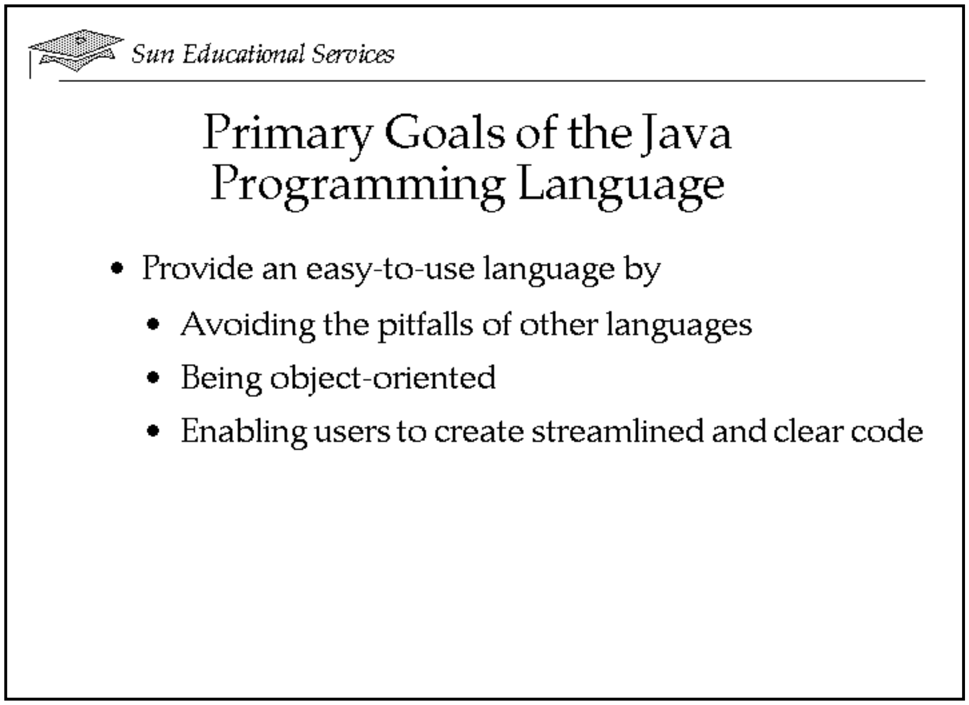
Java is

- A programming language
- A development environment
- An application environment
- A deployment environment

The syntax of the Java programming language is similar to C++ syntax and the semantics are similar to SmallTalk™ semantics. The Java programming language can be used to create all kinds of applications that you could create using any conventional programming language.

Java programming language is usually mentioned in the context of the World Wide Web (WWW) and browsers that are capable of running programs called *applets*. Applets are programs written in the Java programming language that reside on WWW servers, are downloaded by a browser to a client's system, and are run by that browser. Applets are usually small in size to minimize download time and are invoked by a hypertext markup language (HTML) Web page.

Java *applications* are stand-alone programs that do not require any Web browser to execute. They are typically general-purpose programs that run on any machine where the Java runtime environment is installed.



Sun Educational Services

## Primary Goals of the Java Programming Language

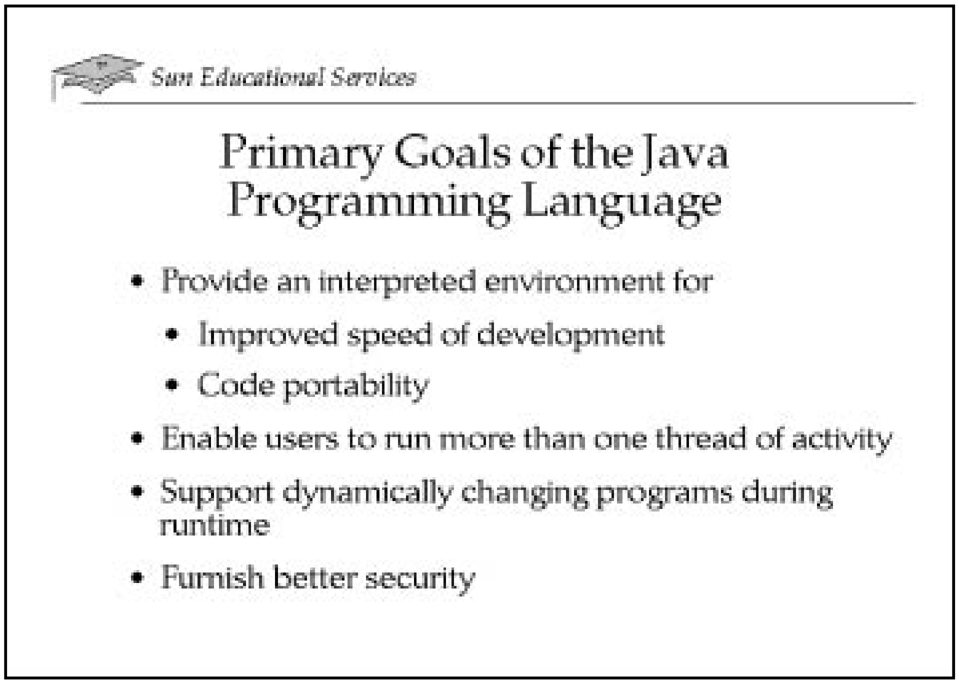
- Provide an easy-to-use language by
  - Avoiding the pitfalls of other languages
  - Being object-oriented
  - Enabling users to create streamlined and clear code

## *What Is the Java Programming Language?*

### *Primary Goals of the Java Programming Language*

The goals in developing the Java programming language were to provide:

- A language which is easy to program by
  - Eliminating the pitfalls of other languages such as pointer arithmetic and memory management which affect code robustness
  - Being object-oriented to help the programmer visualize the program in real-life terms
  - Providing a means to make code as streamlined and clear as possible


Sun Educational Services

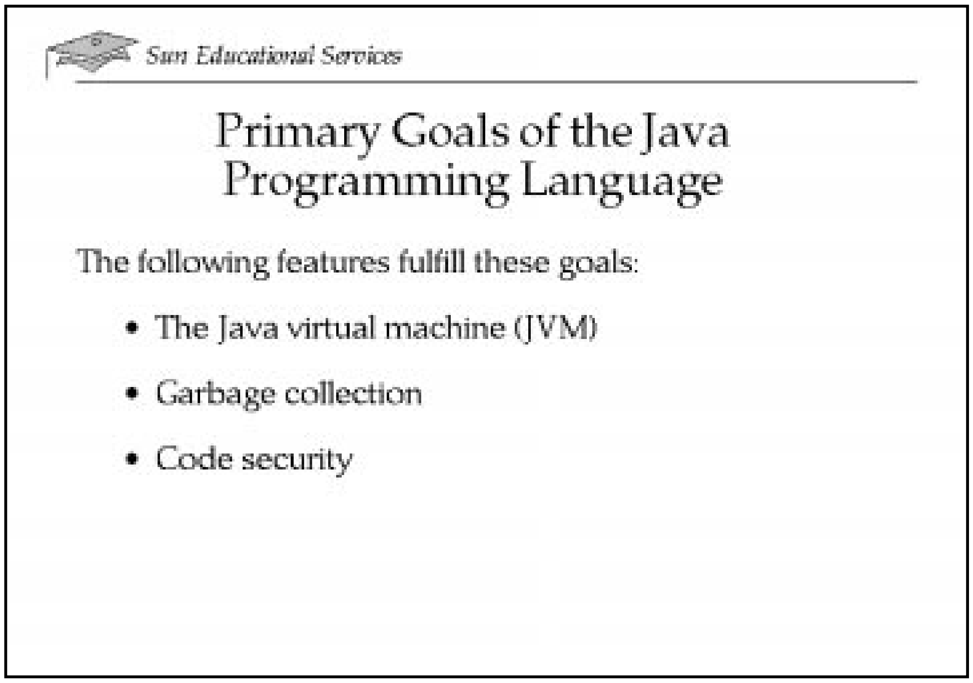
Primary Goals of the Java Programming Language

- Provide an interpreted environment for
  - Improved speed of development
  - Code portability
- Enable users to run more than one thread of activity
- Support dynamically changing programs during runtime
- Furnish better security

## What Is the Java Programming Language?

### Primary Goals of the Java Programming Language (Continued)

- An interpreted environment resulting in the following two benefits:
  - Speed of development – elimination of the compile-link-load-test cycle
  - Code portability – enable the operating system to make system level calls on behalf of the runtime environment
- A way for programs to run more than one thread of activity
- A means to change programs dynamically during their runtime life by allowing them to download code modules
- A means of checking code modules that are loaded to assure security


Sun Educational Services

Primary Goals of the Java Programming Language

The following features fulfill these goals:

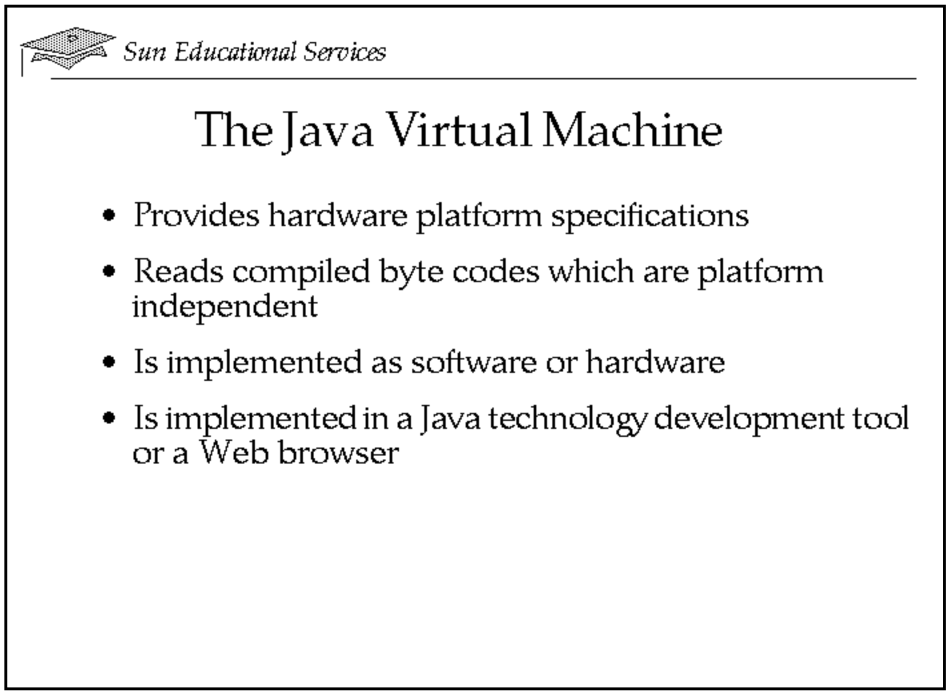
- The Java virtual machine (JVM)
- Garbage collection
- Code security

## *What Is the Java Programming Language?*

### *Primary Goals of the Java Programming Language (Continued)*

The Java technology architecture uses the following features to fulfill the previously listed goals:

- The Java virtual machine
- Garbage collection
- Code security



Sun Educational Services

## The Java Virtual Machine

- Provides hardware platform specifications
- Reads compiled byte codes which are platform independent
- Is implemented as software or hardware
- Is implemented in a Java technology development tool or a Web browser

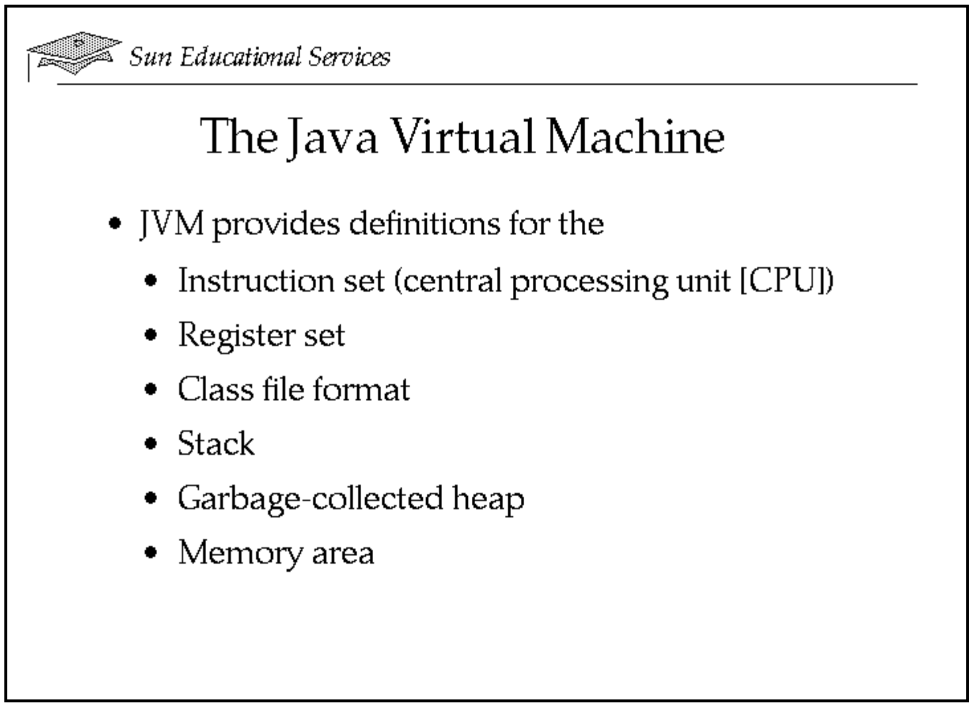
## *What Is the Java Programming Language?*

### *The Java Virtual Machine*

*The Java Virtual Machine Specification* defines the Java virtual machine (JVM) as:

> *An imaginary machine that is implemented by emulating it in software on a real machine. Code for the Java Virtual Machine is stored in* `.class` *files, each of which contains code for at most one public class.*

*The Java Virtual Machine Specification* provides the hardware platform specifications to which all Java technology code is compiled. This specification enables Java software to be platform independent because the compilation is done for a generic machine known as the Java virtual machine (JVM). This “generic machine” can be emulated in software to run on various existing computer systems or implemented in hardware.


Sun Educational Services

## The Java Virtual Machine

- JVM provides definitions for the
  - Instruction set (central processing unit [CPU])
  - Register set
  - Class file format
  - Stack
  - Garbage-collected heap
  - Memory area

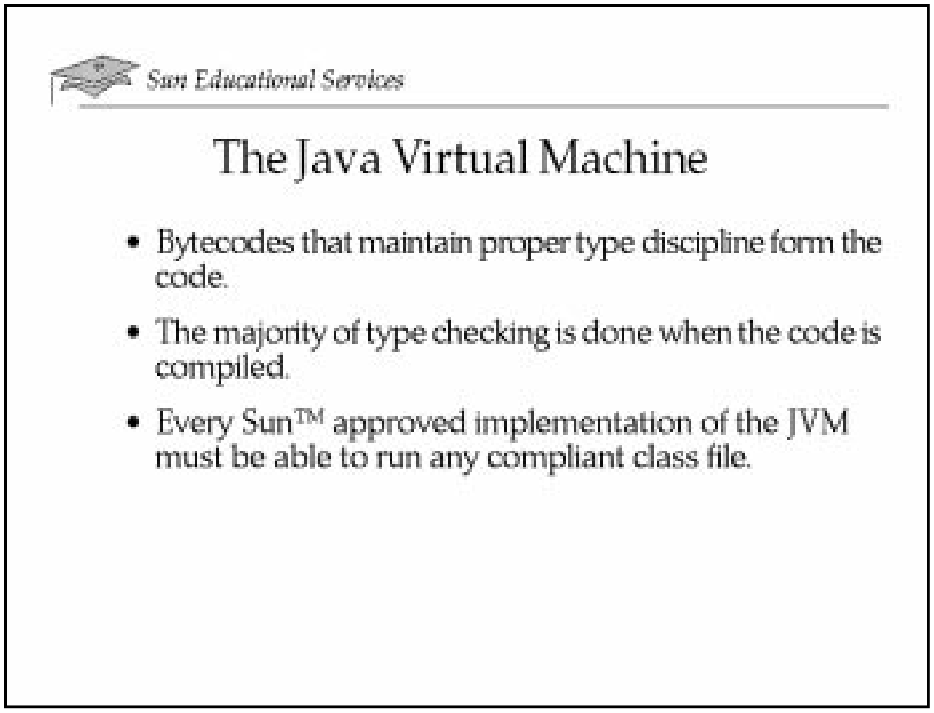
## *What Is the Java Programming Language?*

### *The Java Virtual Machine (Continued)*

The compiler takes the Java application source code and generates bytecodes. Bytecodes are machine code instructions for the JVM. Every Java interpreter, whether it is a Java technology development tool or a Web browser that can run applets, has an implementation of the JVM.

The JVM specification provides concrete definitions for the implementation of the following:

- Instruction set (equivalent to that of a central processing unit [CPU])
- Register set
- Class file format
- Stack
- Garbage-collected heap
- Memory area



Sun Educational Services

## The Java Virtual Machine

- Bytecodes that maintain proper type discipline form the code.
- The majority of type checking is done when the code is compiled.
- Every Sun™ approved implementation of the JVM must be able to run any compliant class file.

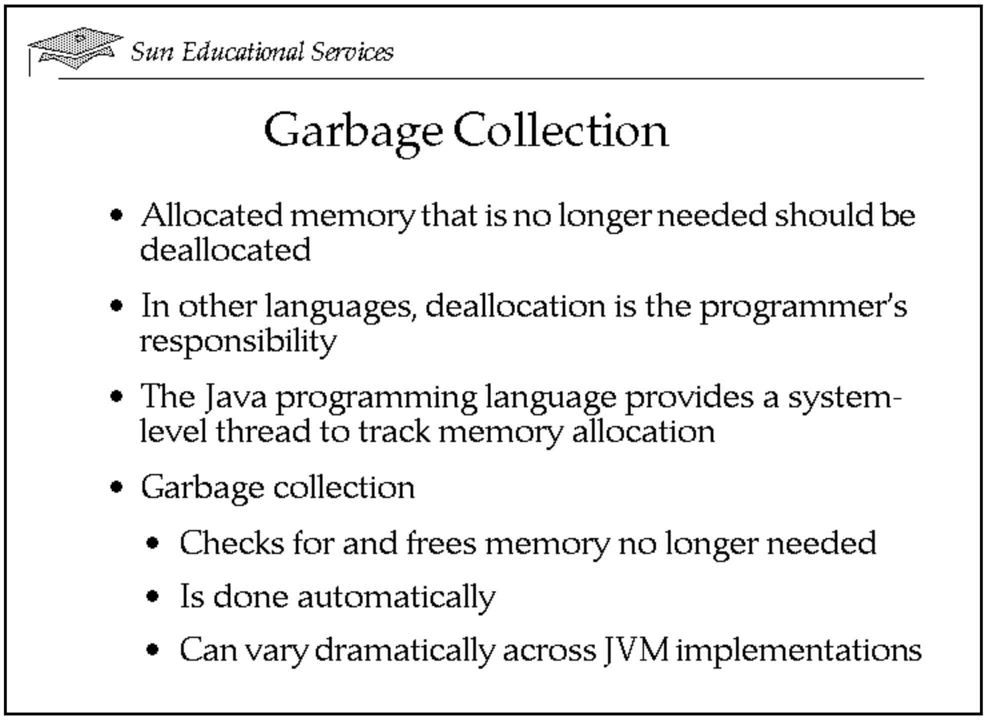
## What Is the Java Programming Language?

### The Java Virtual Machine (Continued)

The code format of the JVM consists of compact and efficient bytecodes. Programs represented by JVM bytecodes must maintain proper type discipline. The majority of type checking is done at compile time.

Any compliant Java technology interpreter must be able to run any program with class files that conform to the class file format specified in *The Java Virtual Machine Specification*.

✓ ***The JVM is supported by JavaSoft™ (who wrote the port) on the Solaris platform, 32-bit Microsoft Windows (Windows 95 and Windows NT), and MacOS. However, there are ports for OS/2, HPUX, AIX, IRIX, UNIXWare, Linux, NetWare 4.0, MVS, OS/2 Warp, OS/390, OS/400, and others.***



Sun Educational Services

Garbage Collection

- Allocated memory that is no longer needed should be deallocated
- In other languages, deallocation is the programmer's responsibility
- The Java programming language provides a system-level thread to track memory allocation
- Garbage collection
  - Checks for and frees memory no longer needed
  - Is done automatically
  - Can vary dramatically across JVM implementations

## *What Is the Java Programming Language?*

### *Garbage Collection*

Many programming languages allow the dynamic allocation of memory at runtime. The process of allocating memory varies based on the syntax of the language, but always involves returning a pointer to the starting address of a memory block.

Once the allocated memory is no longer required (the pointer that references the memory has gone *out of scope*), the program or runtime environment should deallocate the memory.

In C, C++, and other languages, the program developer is responsible for deallocating the memory. This can be a difficult exercise at times, since it is not always known in advance when memory should be released. Programs that do not deallocate memory can eventually crash when there is no memory left on the system to allocate. These programs are said to have memory leaks.

## What Is the Java Programming Language?

### Garbage Collection (Continued)

The Java programming language removes the responsibility for deallocating memory from the programmer. It provides a system-level thread that tracks each memory allocation. During idle cycles in the Java virtual machine, the garbage collection thread checks for and frees any memory that can be freed.

Garbage collection happens automatically during the lifetime of a Java technology program, eliminating the need to deallocate memory and avoiding memory leaks. However, garbage collection schemes can vary dramatically across JVM implementations.

## What Is the Java Programming Language?

### Code Security

#### Overview

The following figure illustrates the Java technology runtime environment and how it enforces code security:.

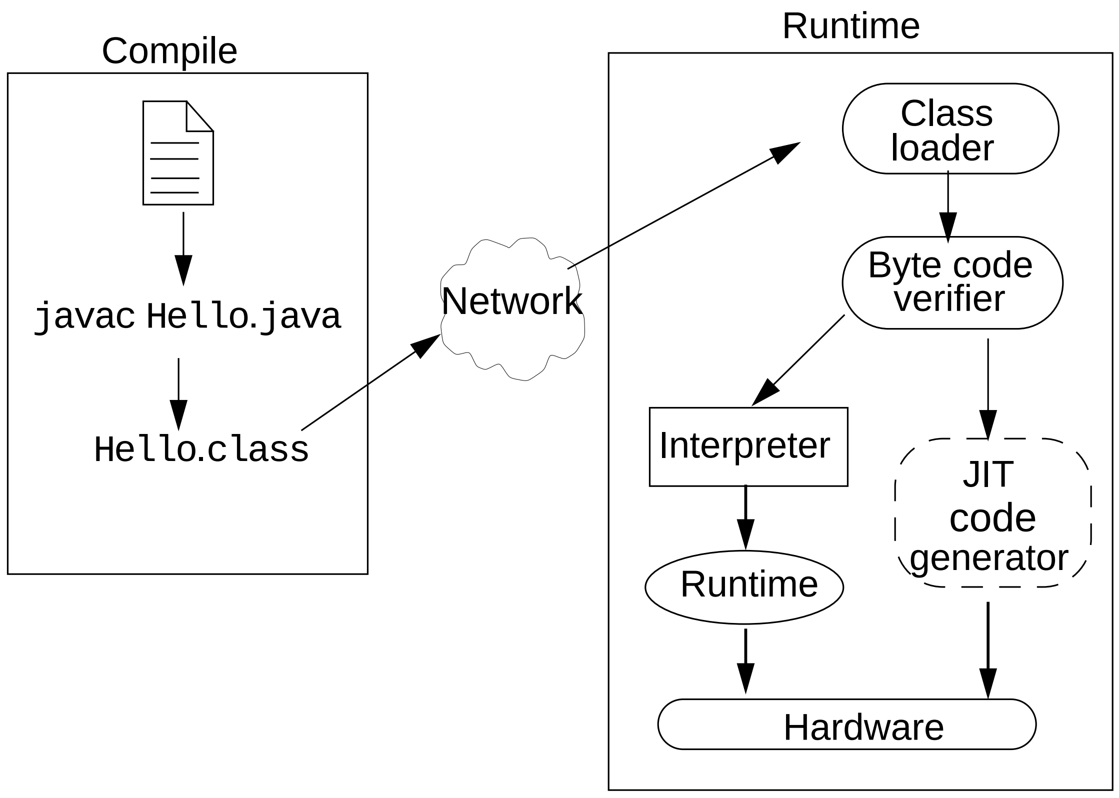


**Figure 1-1** Java Technology Runtime Environment

Java software source files are "compiled" in the sense that they are converted into a set of bytecodes from the text format in which programmers write them. The bytecodes are stored in `.class` files.

At runtime, the bytecodes that make up a Java software program are loaded, checked, and run in an interpreter. In the case of applets, the bytecodes can be downloaded and then interpreted by the JVM built into the browser. The interpreter has two functions: it executes bytecodes, and makes the appropriate calls to the underlying hardware.

## What Is the Java Programming Language?

### Code Security

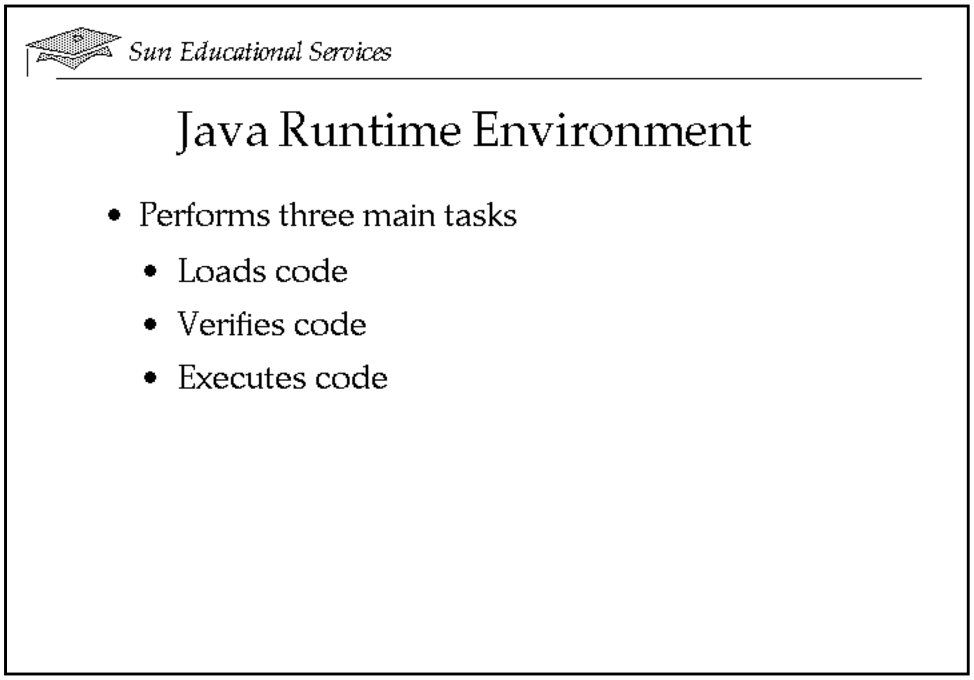
#### Overview (Continued)

In some Java technology runtime environments, a portion of the verified bytecode is compiled to native machine code and executed directly on the hardware platform. This allows Java software code to run close to the speed of C or C++ with a small delay at loadtime to allow compilation to the native machine code.

---

**Note –** Sun Microsystems™ has enhanced the Java virtual machine by adding new performance-enabling technologies. This new virtual machine is called the HotSpot™ virtual machine, and has the potential to enable the Java programming language to run as fast as compiled C++. The HotSpot virtual machine exploits the native multithreading support of the operating system rather than simulate multithreading. It thereby ensures that applications need not specifically have code to use this capability. HotSpot technology eliminates the trade-off between performance and portability.

---



Sun Educational Services

## Java Runtime Environment

- Performs three main tasks
  - Loads code
  - Verifies code
  - Executes code

## What Is the Java Programming Language?

### Code Security

#### The Java Runtime Environment

A Java technology runtime environment runs code compiled for a JVM and performs three main tasks:

- Loading code – Performed by the class loader
- Verifying code – Performed by the bytecode verifier
- Executing code – Performed by the runtime interpreter

#### Class Loader

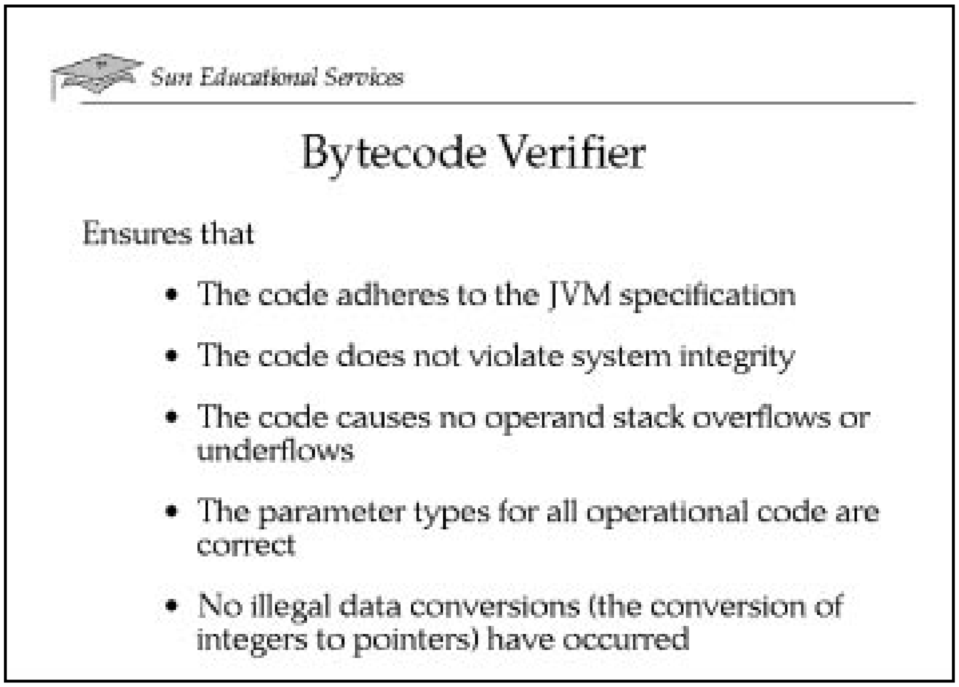
The class loader loads all classes needed for the execution of a program. The class loader adds security by separating the namespaces for the classes of the local file system from those imported from network sources. This limits any Trojan horse applications because local classes are always loaded first.

## What Is the Java Programming Language?

### Code Security (Continued)

✓ ***Classes that are imported from across the network are loaded into a private namespace associated with the origin. When a class from the private namespace accesses another class, the built-in (local system) classes are checked first, then those in the namespace of the referencing class. This prevents a class from spoofing (creating a hoax of) a built-in class.***

Once all of the classes have been loaded, the memory layout of the executable file is determined. At this point specific memory addresses are assigned to symbolic references and the lookup table is created. Since memory layout occurs at runtime, the Java technology interpreter adds protection against unauthorized access into the restricted areas of code.


Sun Educational Services

## Bytecode Verifier

Ensures that

- The code adheres to the JVM specification
- The code does not violate system integrity
- The code causes no operand stack overflows or underflows
- The parameter types for all operational code are correct
- No illegal data conversions (the conversion of integers to pointers) have occurred

## *What Is the Java Programming Language?*

### *Code Security (Continued)*

#### *Bytecode Verifier*

Java software code passes several tests before actually running on your machine. The JVM puts the code through a bytecode verifier that tests the format of code fragments and checks code fragments for illegal code—code that forges pointers, violates access rights on objects, or attempts to change object type.

---

**Note –** All class files imported across the network pass through the bytecode verifier.

---

## Code Security (Continued)

### Verification Process

The bytecode verifier makes four passes on the code in a program. It ensures that the code adheres to the JVM specifications and does not violate system integrity. If the verifier completes all four passes without returning an error message, then the following is ensured:

- The classes adhere to the class file format of the JVM specification.
- There are no access restriction violations.
- The code causes no operand stack overflows or underflows.
- The types of parameters for all operational codes are known to always be correct.
- No illegal data conversions, such as converting integers to object references, have occurred.

## A Basic Java Application

Like any other programming language, the Java programming language is used to create applications. One common minimum application is one which displays the string `Hello World!` on the screen. The following code shows this minimum Java application.

### HelloWorldApp

```
1  //
2  // Sample HelloWorld application
3  //
4  public class HelloWorldApp {
5    public static void main (String args[]) {
6      System.out.println("Hello World!");
7    }
8  }
```

These lines are the minimum components necessary to print `Hello World!` to your screen.

✓ ***The following pages describe this program line by line.***

## HelloWorldApp *Described*

### *Lines 1–3*

Lines 1–3 in the program are comment lines

```
1  //
2  // Sample HelloWorld application
3  //
```

### *Line 4*

Line 4 declares the class name as `HelloWorldApp`. A class name specified in a source file creates a *`classname`*`.class` file in the same directory as the source code. In this case, the compiler creates a file called `HelloWorldApp.class.`It contains the compiled code for the public class `HelloWorldApp.`

```
4  public class HelloWorldApp {
```

### *Line 5*

Line 5 is where the execution of the program starts. The Java technology interpreter must find this defined exactly as given or it will refuse to run the program.

Other programming languages, notably C and C++, also use the `main()` declaration as the starting point for execution. The various parts of this declaration will be covered later in this course.

If the program is given any arguments on its command line, these are passed into the `main()` method, in an array of `String` called `args`. In this example, no arguments are used.

## A Basic Java Application

### `HelloWorldApp` *Described*

#### *Line 5 (Continued)*

```
5    public static void main (String args[]) {
```

This line specifies the following:

- `public` – The method `main()` can be accessed by anything, including the Java technology interpreter.

- `static` – This keyword tells the compiler that the `main()` method is usable in the context of the class `HelloWorldApp` and should be run before the program does anything else.

✓ ***Statics are loaded first and are immediately instantiated at runtime.***

✓ ***As of JDL1.2, you are not allowed to override a `static` method. Therefore, `main` will be visible to the Java runtime environment only if it is defined correctly.***

- `void` – Indicates that the method `main()` does not return anything. This is important because the Java programming language performs careful type checking to confirm that the methods called return the types with which they were declared.

- `String args[]` – The declaration of a `String` array. Contains arguments typed on the command line following the class name. For example:

  `java HelloWorldApp` *args[0] args[1]* `. . .`

✓ ***The Java programming language does not pass the name of the class as an argument to a program.***

## `HelloWorldApp` *Described*

### *Line 6*

Line 6 illustrates the use of a class name, an object name, and a method call. It prints the string "Hello World!" to the standard output using the `println()` method of the `PrintStream` `out` object, referenced by the `out` field of the `System` class.

```
6     System.out.println("Hello World!");
```

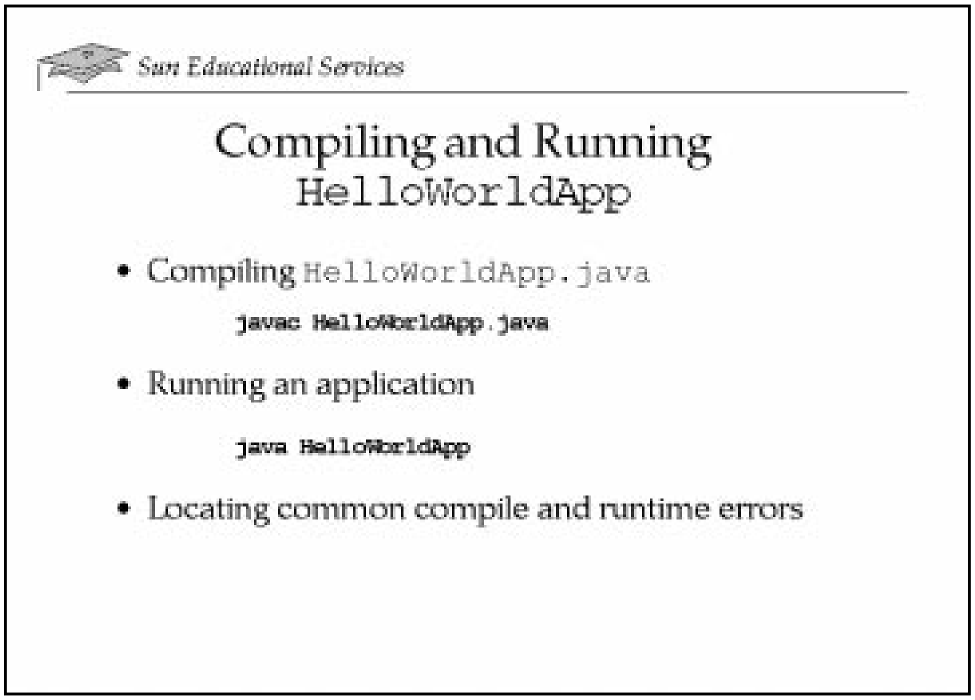
✓ ***The static variable*** `out` ***is defined in the*** `System` ***class to be of type*** `PrintStream`***;*** `PrintStream` ***is a class defined in the*** `java.io` ***package, and the source file is called*** `PrintStream.java`***. This is where the*** `println` ***method is declared.***

The `println()` method in this example takes a string argument and writes it to the standard output stream.

### *Lines 7–8*

Lines 7–8 of the program, the two braces, close the method `main()` and the class `HelloWorldApp`, respectively.

```
7     }
8  }
```



Sun Educational Services

Compiling and Running
HelloWorldApp

- Compiling HelloWorldApp.java

  `javac HelloWorldApp.java`

- Running an application

  `java HelloWorldApp`

- Locating common compile and runtime errors

## A Basic Java Application

### Compiling and Running `HelloWorldApp`

#### Compiling

Once you have created the `HelloWorldApp.java` source file, compile it with the following line:

```
javac HelloWorldApp.java
```

If the compiler does not return any messages, the new file `HelloWorldApp.class` is stored in the same directory as the source file, unless specified otherwise.

If you have a problem compiling the application, check the troubleshooting messages on page 1-25.

## Compiling and Running `HelloWorldApp` (Continued)

### Running

To run your `HelloWorldApp` application, use the Java interpreter, `java`, located in the `bin` directory.

```
java HelloWorldApp
Hello World!
```

**Note –** The `PATH` environment variable must be set to find `java` and `javac`; make sure it includes *java_root* / `bin` (where *java_root* represents the tree root where Java is installed).

✓ ***As of JDK 1.1,*** PATH ***is all that is required.*** CLASSPATH ***is required only when non-system classes or Java archive (JAR) files are loaded or specified.***

# A Basic Java Application

## Troubleshooting the Compilation

### Compile-Time Errors

The following are common errors seen at compile time:

- `javac: Command not found`

  The `PATH` variable is not set properly to include the `javac` compiler. The `javac` compiler is located in the `bin` directory below the installed JDK directory.

-
  ```
  HelloWorldApp.java:6: Method
  printl(java.lang.String) not found in class
  java.io.PrintStream.
  System.out.printl("Hello World!");
  ```

  The method name `println` is typed incorrectly.

- `In class HelloWorldApp: main must be public and static`

  This error occurs because either the word `static` or `public` was left out of the line containing the `main` method.

### Runtime Errors

Some of the errors generated when typing `java HelloWorldApp` are:

- `Can't find class HelloWorldApp`

  Generally, this means that the class name specified on the command line was spelled differently than the *`filename`*`.class` file. The Java programming language is *case sensitive*.

  For example,

  ```
  public class HelloWorldapp {
  ```

  creates a `HelloWorldapp.class`, which is not the class name (`HelloWorldApp.class`) the compiler expected.

## A Basic Java Application

### Troubleshooting the Compilation

#### Runtime Errors (Continued)

- Naming

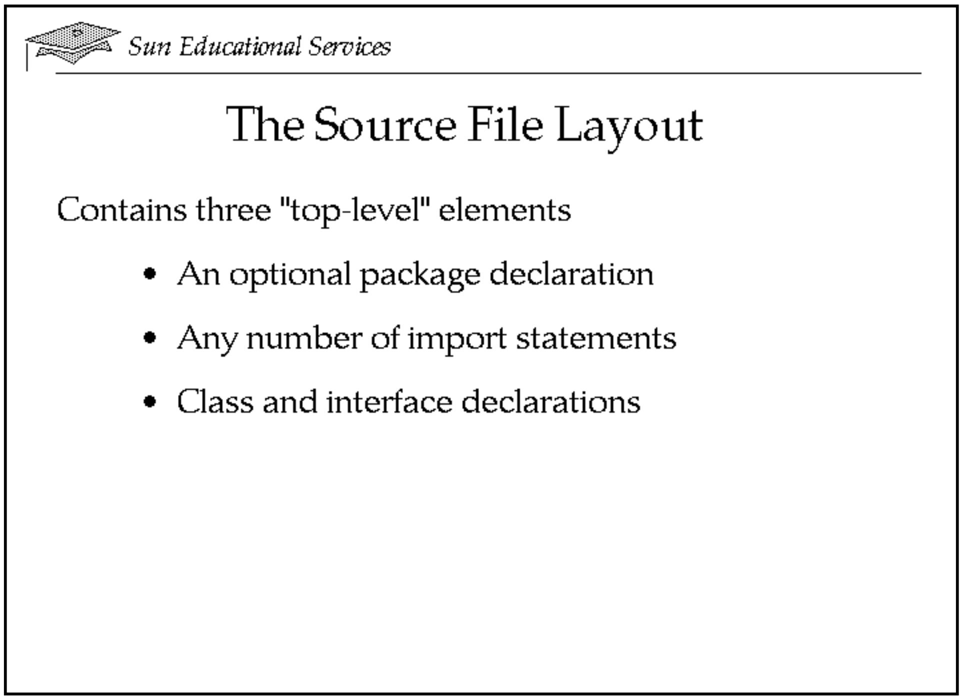
  If the `.java` file contains a public class, then it must have the same file name as that class. For example, the definition of the class in the previous example is

  ```
  public class HelloWorldapp
  ```

  The name of the source file must therefore be `HelloWorldapp.java.`

- Class count

  Only one public class can be defined in a source file.



Sun Educational Services

## The Source File Layout

Contains three "top-level" elements

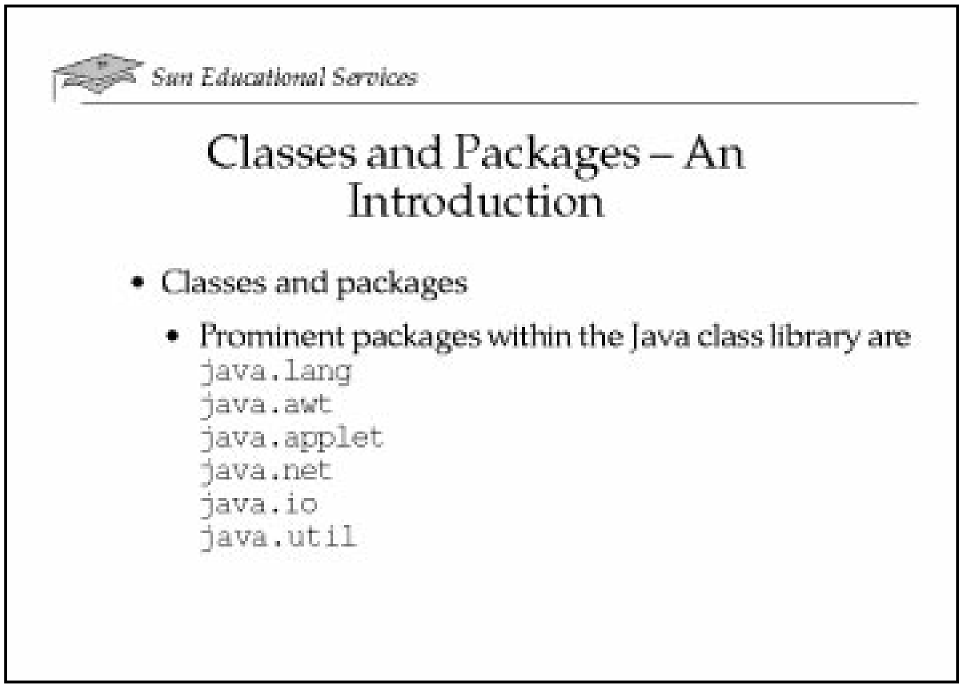
- An optional package declaration
- Any number of import statements
- Class and interface declarations

## *The Source File Layout*

A `.java` source file can contain three “top-level” elements:

- A package declaration (optional)
- Any number of import statements
- Class and interface definitions

These items must appear in this order. That is, any import statements must precede all class definitions and if a package declaration is used, it must precede both the classes and imports.


Sun Educational Services

Classes and Packages – An Introduction

- Classes and packages
  - Prominent packages within the Java class library are
    `java.lang`
    `java.awt`
    `java.applet`
    `java.net`
    `java.io`
    `java.util`

## *Classes and Packages – An Introduction*

A *class* is a generic term for a module that provides functionality. The Java Development Kit comes with a standard set of classes (called the class library) that implements most of the basic behaviors needed—not only for programming tasks (classes to provide basic math functions, arrays, and strings, for example), but also for graphics and networking.

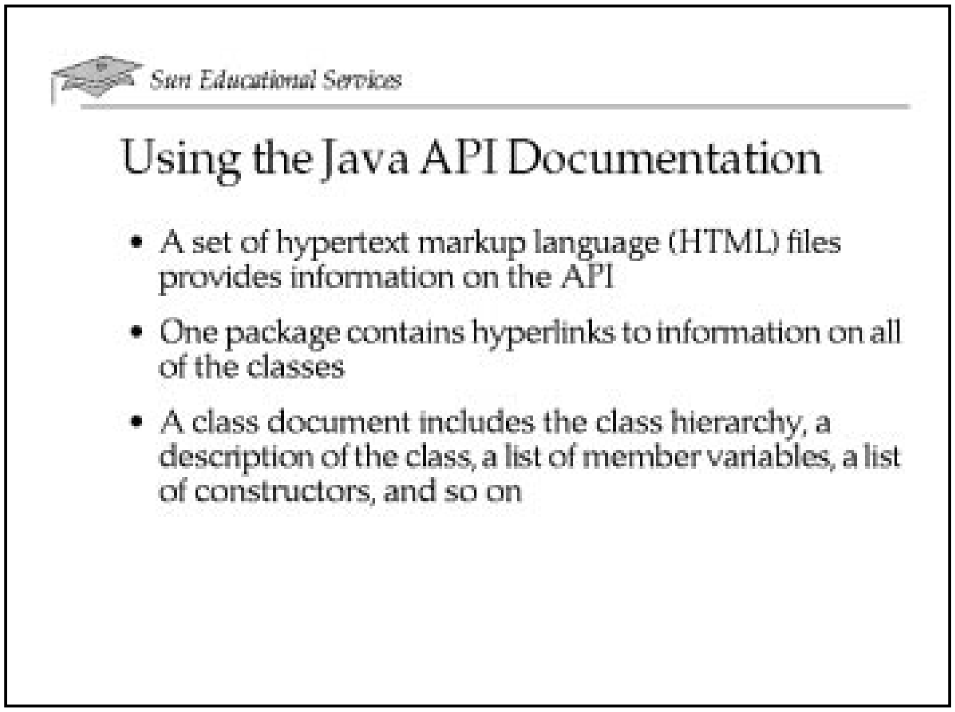
A *package* is a group of related classes.

The class library is organized into many *packages*, each containing several classes. The following packages are prominent:

- `java.lang` contains classes which form the core of the language, such as `String`, `Math`, `Integer`, and `Thread`.
- `java.awt` contains classes that make up the Abstract Window Toolkit (AWT). This package is used for constructing and managing the graphical user interface of the application.

## Classes and Packages – An Introduction

- `java.applet` contains classes that provide applet-specific behavior.
- `java.net` contains classes for performing network related operations and dealing with sockets and uniform resource locators (URLs).
- `java.io` contains classes that deal with file I/O.
- `java.util` contains utility classes for tasks such as random number generation, defining system properties, and using date and calendar related functions.



Sun Educational Services

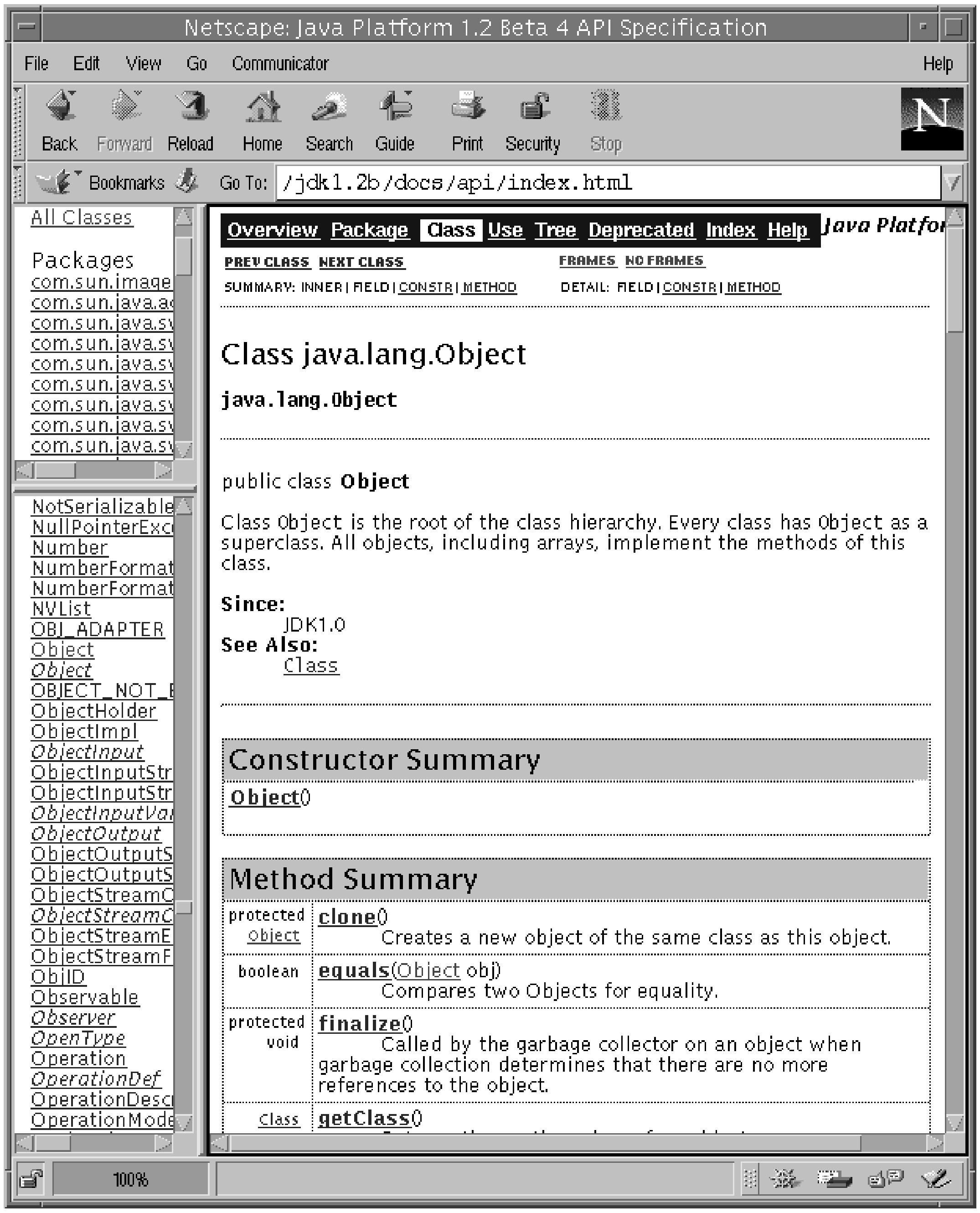
## Using the Java API Documentation

- A set of hypertext markup language (HTML) files provides information on the API
- One package contains hyperlinks to information on all of the classes
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on

## *Using the Java API Documentation*

A set of HTML files documents the supplied application programming interface (API). The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. If a particular package hotlink is selected, the classes which are members of that package will be listed. Selecting a class hotlink from a package page will present a page of information about that class. Figure 1-2 shows one such class.

## Using the Java API Documentation


Netscape: Java Platform 1.2 Beta 4 API Specification
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Go To: /jdk1.2b/docs/api/index.html
Overview Package Class Use Tree Deprecated Index Help
Java Platfo
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
100%

All Classes

Packages
com.sun.image
com.sun.java.a
com.sun.java.s
com.sun.java.s
com.sun.java.s
com.sun.java.s
com.sun.java.s
com.sun.java.s
com.sun.java.s

NotSerializable
NullPointerExc
Number
NumberFormat
NumberFormat
NVList
OBJ_ADAPTER
Object
*Object*
OBJECT_NOT_
ObjectHolder
ObjectImpl
*ObjectInput*
ObjectInputStr
ObjectInputStr
*ObjectInputVa*
*ObjectOutput*
ObjectOutputS
ObjectOutputS
ObjectStreamC
*ObjectStreamC*
ObjectStreamE
ObjectStreamF
ObjID
Observable
*Observer*
*OpenType*
Operation
*OperationDef*
OperationDesc
OperationMode

Class java.lang.Object

**java.lang.Object**

public class **Object**

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

**Since:**
JDK1.0
**See Also:**
Class

| Constructor Summary |
|---|
| **Object**() |

| Method Summary | |
|---|---|
| protected Object | **clone**() Creates a new object of the same class as this object. |
| boolean | **equals**(Object obj) Compares two Objects for equality. |
| protected void | **finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. |
| Class | **getClass**() |

**Figure 1-2** Java API Documentation With HTML

## Using the Java API Documentation

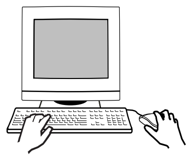
The main sections of a class document include

- The class hierarchy
- A description of the class and its general purpose
- A list of member variables
- A list of constructors
- A list of methods
- A detailed list of variables, with descriptions of the purpose and use of each
- A detailed list of constructors, with descriptions
- A detailed list of methods, with descriptions

## Exercise: Performing Basic Tasks

### About the Labs

- ✓ *Almost every module in this course has a set of exercises. Each exercise is marked as either a Level 1, Level 2, or Level 3 lab. The Level 1 labs are designed to reinforce the material presented in the module. Level 2 labs extend the material presented in the module, providing additional practice. The Level 3 labs require some additional research in order to complete—this might include external materials like books on the Java programming language, or material that is available in the Java API documentation.*
- ✓ *Every module will have a Level 2 and/or Level 3 lab.*



**Exercise objective –** In this exercise you will identify packages, classes, and methods in the Java API documents for standard input and output methods. You will also write, compile, and run two simple applications using these methods.

### *Preparation*

An understanding of the concepts and terminology presented in this module is critical to being able to navigate the documentation and to apply the information obtained from the documentation to writing a program.

### *Tasks*

#### *Level 1 Lab: Read the Documentation*

Complete the following steps:

1. Start the API browser and open the index page of the Java API on-line documents. Your instructor will give you instructions on how to do this.
2. Find the `java.lang` package.

## Exercise: Performing Basic Tasks

### Level 1 Lab: Read the Documentation (Continued)

3. Answer the following questions:

   ▼ What classes are defined in this package?

   ▼ What are some methods in the `System` class?

   ▼ What package is the `System.out.println` method defined in?

   ▼ What is the standard input method called?

### Level 2 Lab: Create a Java Application

Perform the following tasks:

1. Using any text editor, create an application similar to `HelloWorldApp` that prints a string of your choosing.

2. Compile the program and correct any errors.

3. Run the program using the interpreter.

### Level 3 Lab: Use Standard Input and Standard Output

Write an application called `MyCat` that will read a line from the standard input stream and write the line back out to the standard output stream.

## Notes

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Describe key features of Java programming language
- ❑ Describe the JVM function
- ❑ Describe how garbage collection works
- ❑ List the three tasks performed by the Java platform that handle code security
- ❑ Define the terms *class*, *packages*, *applets*, and *applications*
- ❑ Write, compile, and run a simple Java application
- ❑ Use the Java technology application programming interface on-line documentation to identify the methods of the `java.lang` package.

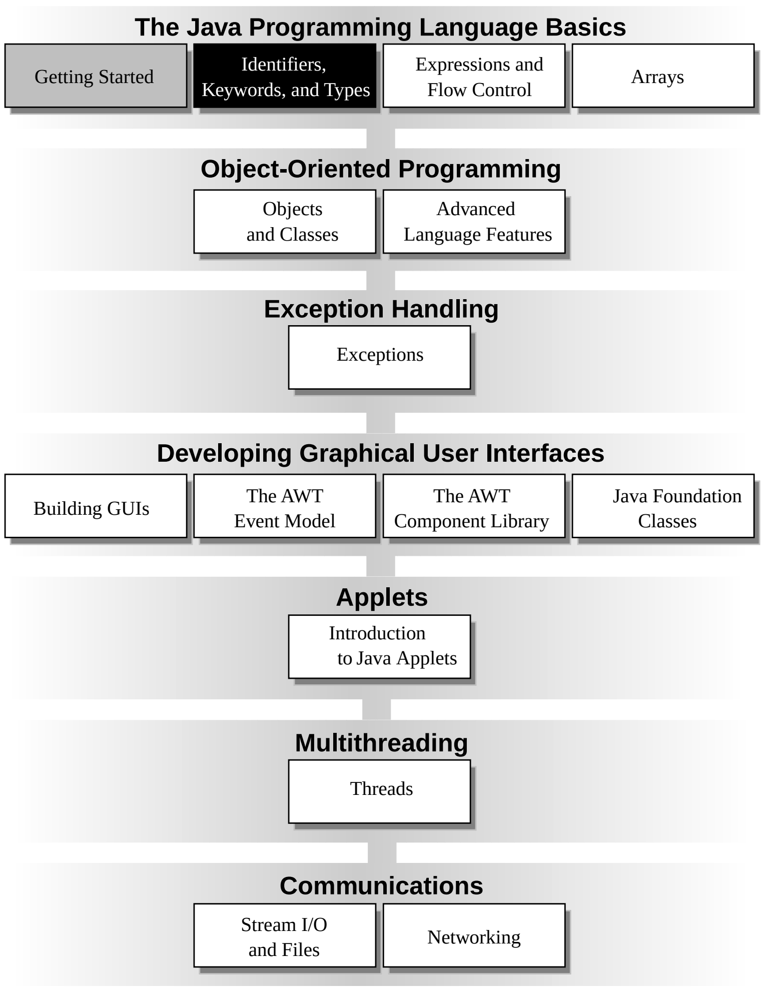
## Think Beyond

How can you benefit from using this programming language in your work environment?

# Identifiers, Keywords, and Types 2

## Course Map

This module covers some of the basic components used in Java technology programs including variables, keywords, primitive types, and class types.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following questions are relevant to the material presented in this module:

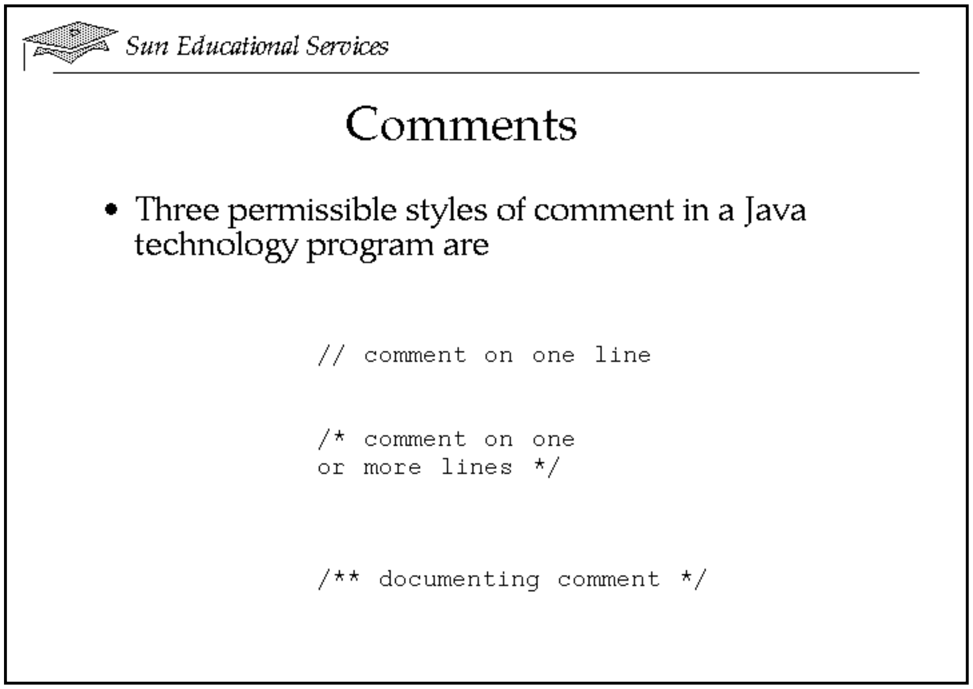
- What is your understanding of a class?
- What is your understanding of an object?

## Objectives

Upon completion of this module, you should be able to

- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *class*, *object*, *member variable*, and *reference variable*
- Create a class definition for a simple class containing primitive member variables
- Declare variables of class type
- Construct an object using `new`
- Describe default initialization
- Access the member variables of an object using the dot notation
- Describe the significance of a reference variable
- State the consequences of assigning variables of class type

✓ ***Documenting comments are explained next. The complete definition of the documentation system used by the JavaSoft team and created by*** `javadoc` ***is defined in "The Design of Distributed Hyperlinked Programming Documentation.", a paper by Lisa Friendly. It is available from*** `http://www.javasoft.com/doc/api_documentation.html`***.***


Sun Educational Services

Comments

- Three permissible styles of comment in a Java technology program are

```
// comment on one line

/* comment on one
or more lines */


/** documenting comment */
```

## Comments

There are three permissible styles for inserting comments:

```
// comment on one line
/* comment on one or more lines */
/** documenting comment */
```

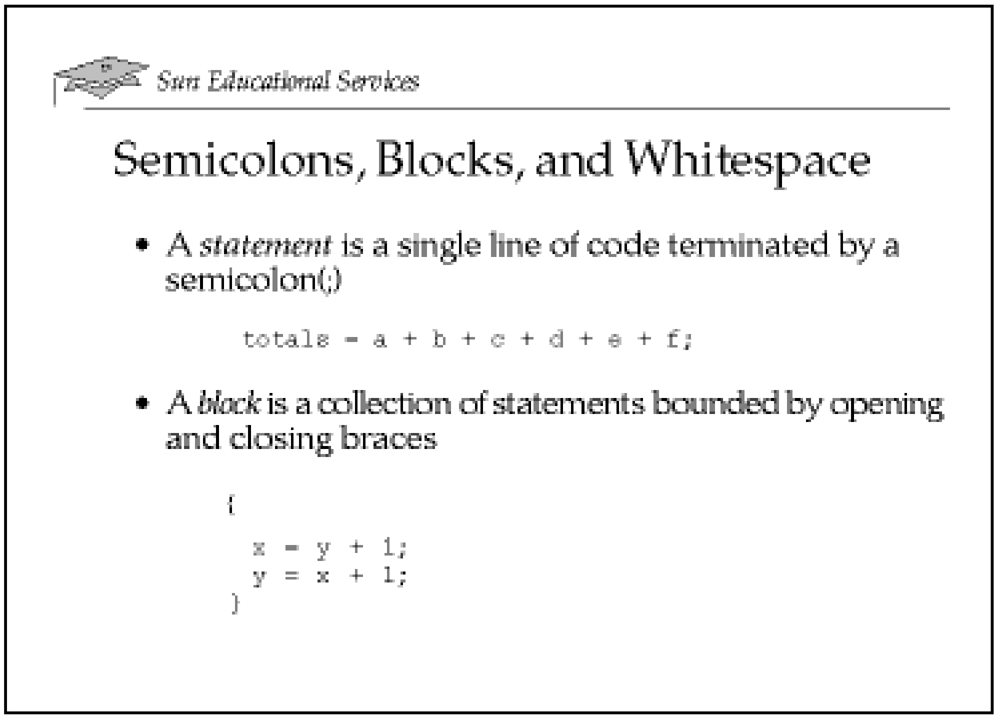
✓ ***The last comment which contains two asterisks at the beginning and one at the end is correct.***

Documenting comments that are placed immediately before a declaration (of a variable, method, or class) indicate that the comment should be included in any automatically generated documentation (the HTML files generated by the `javadoc` command) to serve as a description of the declared item.

---

**Note –** The format of these comments and the use of the `javadoc` tool is discussed in the `docs/tooldocs/solaris` directory of the API documentation for JDK 1.2.

---



Sun Educational Services

## Semicolons, Blocks, and Whitespace

- A *statement* is a single line of code terminated by a semicolon(;)

```
totals = a + b + c + d + e + f;
```

- A *block* is a collection of statements bounded by opening and closing braces

```
{
  x = y + 1;
  y = x + 1;
}
```

## *Semicolons, Blocks, and Whitespace*

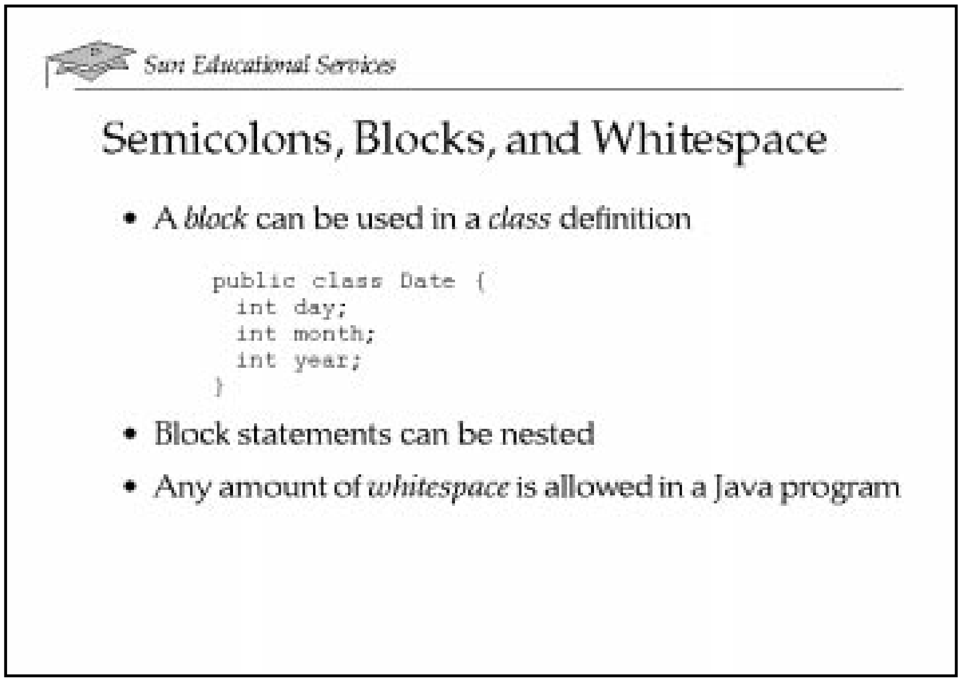
In the Java programming language, a statement is a single line of code terminated with a semicolon (;).

For example,

```
totals = a + b + c + d + e + f;
```

is the same as

```
totals = a + b + c +
    d + e + f;
```



Sun Educational Services

## Semicolons, Blocks, and Whitespace

- A *block* can be used in a *class* definition

```
public class Date {
  int day;
  int month;
  int year;
}
```

- Block statements can be nested
- Any amount of *whitespace* is allowed in a Java program

## *Semicolons, Blocks, and Whitespace*

A *block* or a compound statement is a collection of statements bounded by opening and closing braces ({ }). *Block* statements are also used to group statements belonging to a class.

Blocks of statements can be nested. Consider the `HelloWorldApp` class which consists of the `main` method. This method is a block of statements that is a single unit, the unit itself being one of a group of things in the class `HelloWorldApp` block.

Some other examples of block statements or groupings are

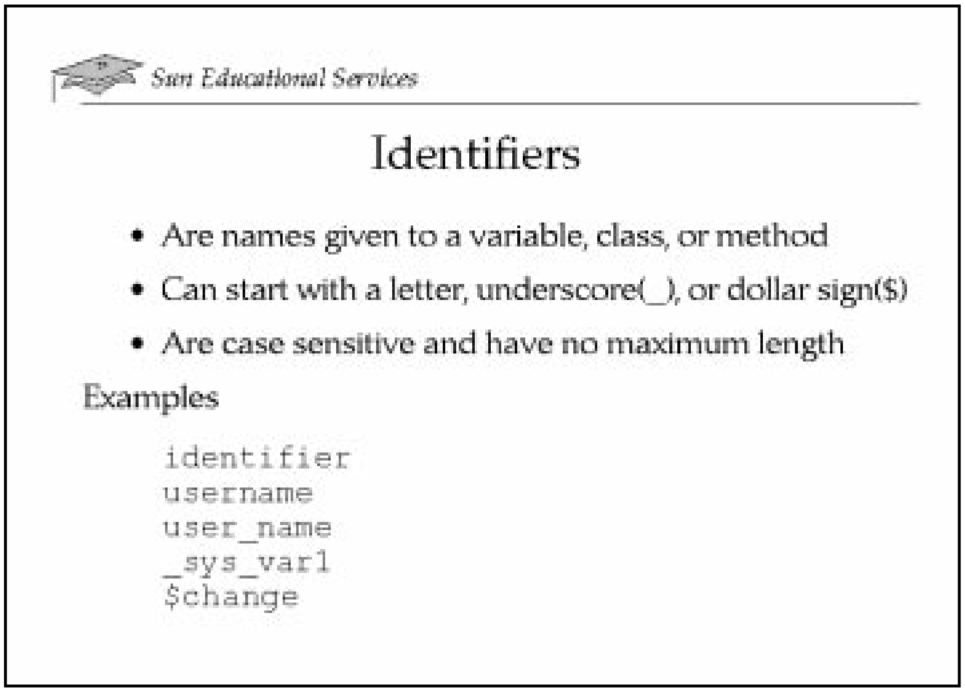
```
// a block statement

{
  x = y + 1;
  y = x + 1;
}
```

## Semicolons, Blocks, and Whitespace

```
// a block used in a class definition
public class MyDate {
  int day;
  int month;
  int year;
}

// an example of a block statement nested within
// another block statement
while ( i < large ) {
  a = a + i;
  // nested block
  if ( a == max ) {
     b = b + a;
     a = 0;
  }
}
```

*Whitespace* is allowed between elements of the source code. Any amount of whitespace is allowed. Whitespace, including spaces, tabs, and newlines, can be used to enhance the visual appearance of your source code.

```
{
   int  x;

   x  =  23  *  54;
}
```



Sun Educational Services

## Identifiers

- Are names given to a variable, class, or method
- Can start with a letter, underscore(_), or dollar sign($)
- Are case sensitive and have no maximum length

Examples

```
identifier
username
user_name
_sys_var1
$change
```

## *Identifiers*

In the Java programming language, an *identifier* is a name given to a variable, class, or method. Identifiers start with a letter, underscore (_), or dollar sign ($). Subsequent characters can be digits. Identifiers are case sensitive and have no maximum length.

✓ ***Specifically, the** Java Specification **states that an identifier starts with a Unicode letter and is followed by any number of Unicode letters or digits. The specification lists the Unicode letters and digits. Unicode represents an extended ASCII set capable of handling international characters.***

Valid identifiers are

- `identifier`
- `userName`
- `User_name`
- `_sys_var1`
- `$change`

## *Identifiers*

Java technology sources are in 16-bit Unicode rather than 8-bit ASCII text, so a letter is a considerably wider definition than just a to z and A to Z.

Beware of non-ASCII characters because Unicode can support *different* characters that look the same.

An identifier cannot be a keyword, but it can contain a keyword as part of its name. For example, `thisOne` is a valid identifier, but `this` is not, because `this` is a Java keyword. Java keywords are discussed next.

---

**Note –** Identifiers containing a dollar sign ($) are generally unusual, although languages such as BASIC, along with VAX/VMS systems, make extensive use of them. Because they are unfamiliar, it is probably best to avoid them unless there is a local convention or other pressing reason for including this symbol in the identifier.

---

## Java Keywords

Table 2-1 lists keywords that are used in the Java programming language.

**Table 2-1** Java Keywords

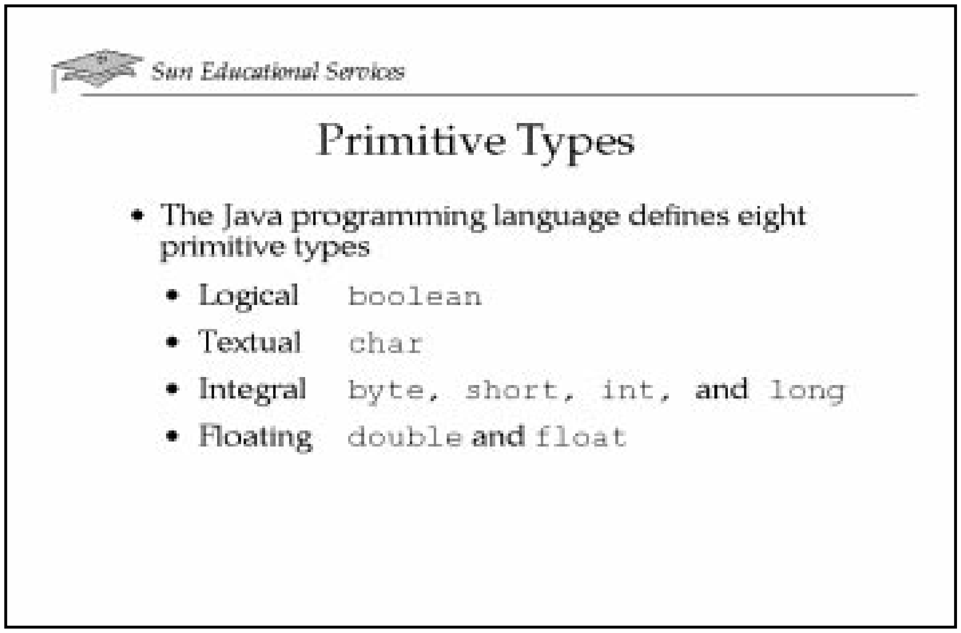
| | | | | |
|---|---|---|---|---|
| `abstract` | `do` | `implements` | `private` | `throw` |
| `boolean` | `double` | `import` | `protected` | `throws` |
| `break` | `else` | `instanceof` | `public` | `transient` |
| `byte` | `extends` | `int` | `return` | `true` |
| `case` | `false` | `interface` | `short` | `try` |
| `catch` | `final` | `long` | `static` | `void` |
| `char` | `finally` | `native` | `super` | `volatile` |
| `class` | `float` | `new` | `switch` | `while` |
| `continue` | `for` | `null` | `synchronized` | |
| `default` | `if` | `package` | `this` | |

Keywords have special meaning to the Java technology compiler. They identify a data type name or program construct name.

The following are important notes about the keywords:

- The literals `true`, `false`, and `null` are lowercase, not uppercase as in the C++ language. Strictly speaking, these are not keywords but literals. The distinction is academic, however.
- There is no `sizeof` operator; the size and representation of all types is fixed and is *not* implementation dependent.
- `goto` and `const` are keywords that are not used in the Java programming language.

✓ ***There is no longer a keyword*** `byvalue`***. The keywords*** `const` ***and*** `goto` ***still exist, but remain unused. While*** `true` ***and*** `false` ***would seem to be keywords, they are actually*** `boolean` ***literals. The word*** `null` ***is also a literal. You can confirm this by reading "The Java Language Specification," ISBN 0-201-63451-1 available from*** `http://www.javasoft.com/doc/language_specification/index.html`


Sun Educational Services

## Primitive Types

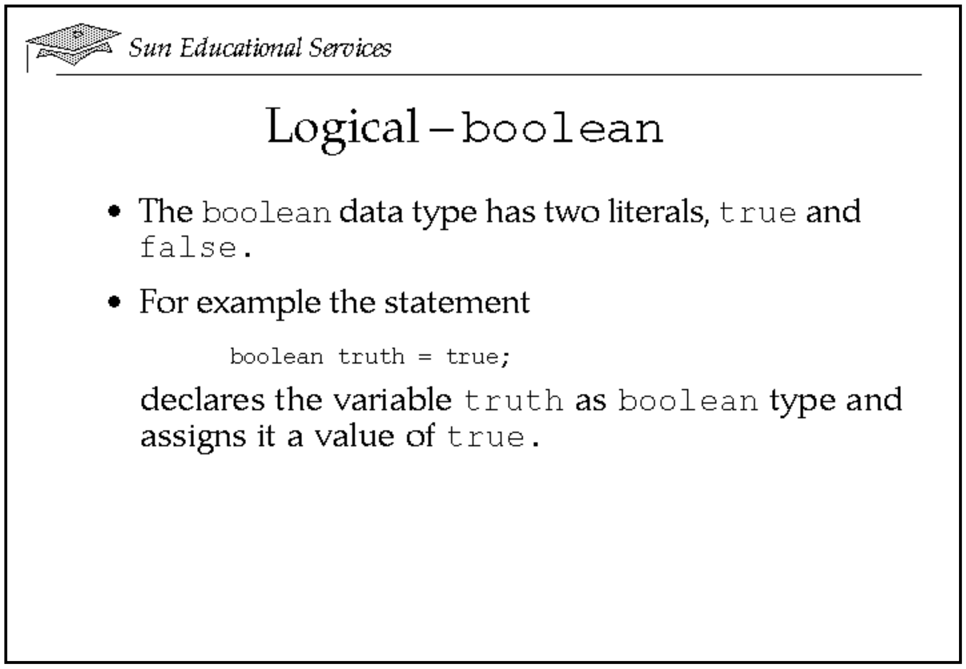
- The Java programming language defines eight primitive types
  - Logical `boolean`
  - Textual `char`
  - Integral `byte, short, int,` and `long`
  - Floating `double` and `float`

## Basic Java Types

### Primitive Types

The Java programming language defines literal values for eight *primitive* data types and one special type. The primitive types can be considered in four categories:

| | |
|---|---|
| ● Logical | `boolean` |
| ● Textual | `char` |
| ● Integral | `byte`, `short`, `int`, and `long` |
| ● Floating point | `double` and `float` |



Sun Educational Services

## Logical – boolean

- The boolean data type has two literals, true and false.
- For example the statement

```
boolean truth = true;
```

declares the variable truth as boolean type and assigns it a value of true.

## *Basic Java Types*

### *Logical –* boolean

Logical values have two states: "on" and "off," "true" and "false," or "yes" and "no." Such a value is represented by the boolean type. The boolean has two literal values: true and false. The following code is an example of the declaration and initialization of a boolean type variable:

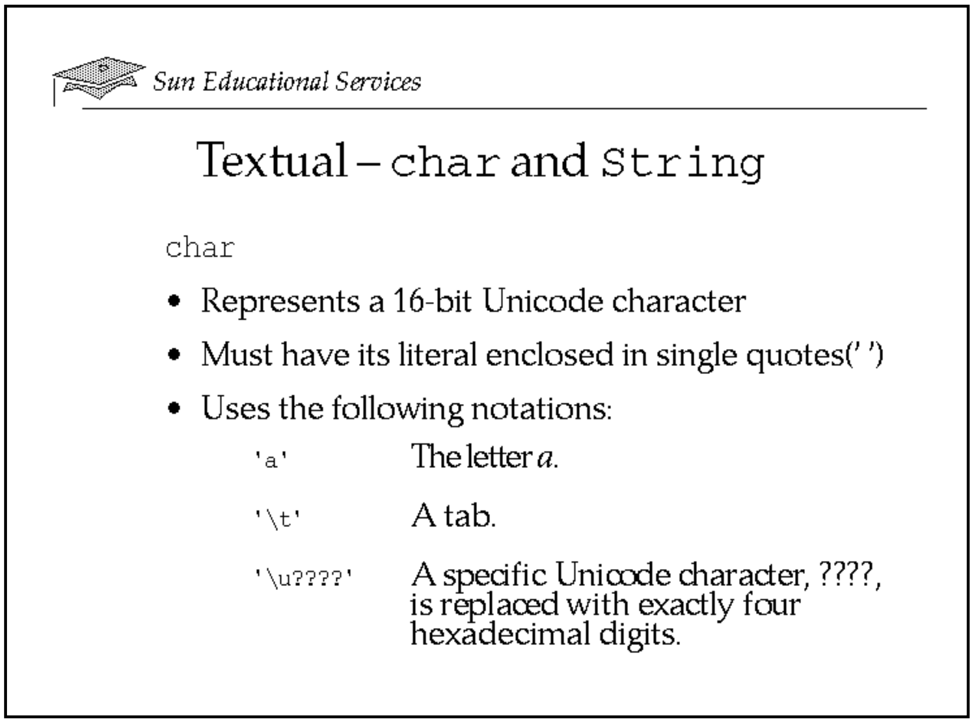
```
// declares the variable truth as boolean and
// assigns it the value true
boolean truth = true;
```

---

**Note –** There are no casts between integer types and the boolean type. Some languages, most notably C and C++, allow numeric values to be interpreted as logical values. This is not permitted in the Java programming language; wherever a boolean is required only boolean values can be used.

---

✓ ***Be sure to explain*** int ***literal formats to students without C experience.***


Sun Educational Services

## Textual – char and String

char

- Represents a 16-bit Unicode character
- Must have its literal enclosed in single quotes(' ')
- Uses the following notations:

| | |
|---|---|
| 'a' | The letter *a*. |
| '\t' | A tab. |
| '\u????' | A specific Unicode character, ????, is replaced with exactly four hexadecimal digits. |

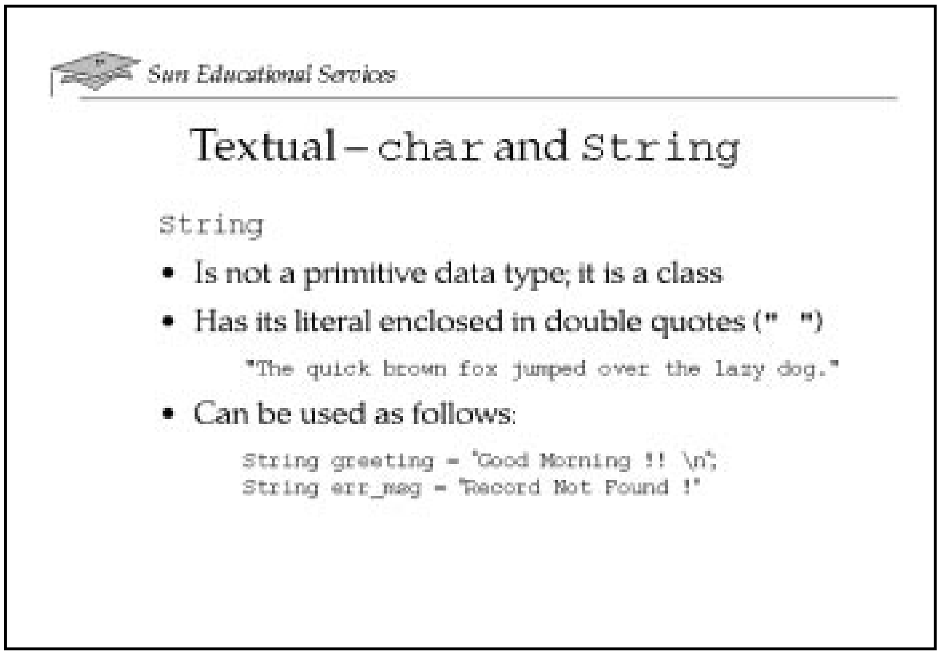
## *Basic Java Types*

### *Textual –* `char` *and* `String`

Single characters are represented by using the `char` type. A `char` represents a 16-bit unsigned Unicode character. A `char` literal must be enclosed in single quotes (' '). For example:

| | |
|---|---|
| ● `'a'` | The letter `a.` |
| ● `'\t'` | A tab. |
| ● `'\u????'` | A specific Unicode character, `????,` is replaced with exactly four hexadecimal digits. |

The `String` type, which is not a primitive but a class, is used to represent sequences of characters. The characters themselves are Unicode, and the backslash notation shown previously for the `char` type also works in a `String`. Unlike C and C++, strings do not end with `\0`.



Sun Educational Services

## Textual – char and String

String

- Is not a primitive data type; it is a class
- Has its literal enclosed in double quotes (" ")

```
"The quick brown fox jumped over the lazy dog."
```

- Can be used as follows:

```
String greeting = "Good Morning !! \n";
String err_msg = "Record Not Found !"
```

## Basic Java Types

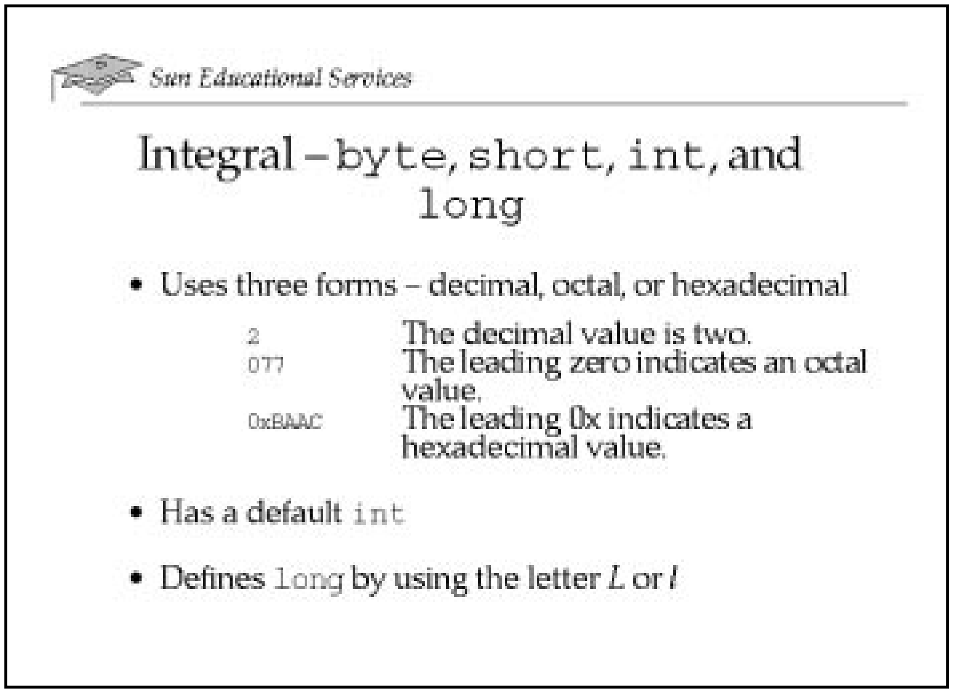
### Textual – `char` and `String` (Continued)

A `String` literal is enclosed in double quote marks like this:

```
"The quick brown fox jumped over the lazy dog."
```

Some examples of the declarations and initialization of `char` and `String` type variables are:

```
// declares and initializes a char variable
char ch = 'A';

// declares two char variables
char ch1,ch2;

// declare two String variables and initialize them
String greeting = "Good Morning !! \n";
String err_msg = "Record Not Found !";

// declare two String variables
String str1,str2;
```



Sun Educational Services

Integral – `byte`, `short`, `int`, and `long`

- Uses three forms – decimal, octal, or hexadecimal

| | |
|---|---|
| `2` | The decimal value is two. |
| `077` | The leading zero indicates an octal value. |
| `0xBAAC` | The leading 0x indicates a hexadecimal value. |

- Has a default `int`
- Defines `long` by using the letter *L* or *l*

## Basic Java Types

### *Integral* – `byte`, `short`, `int`, *and* `long`

There are four integral types in the Java programming language. Each type is declared using one of the keywords `byte`, `short`, `int`, or `long`. Literals of integral type can be represented using decimal, octal, or hexadecimal forms as follows:

| | |
|---|---|
| `2` | The decimal value is two**.** |
| `077` | The leading zero indicates an octal value. |
| `0xBAAC` | The leading 0x indicates a hexadecimal value. |

---

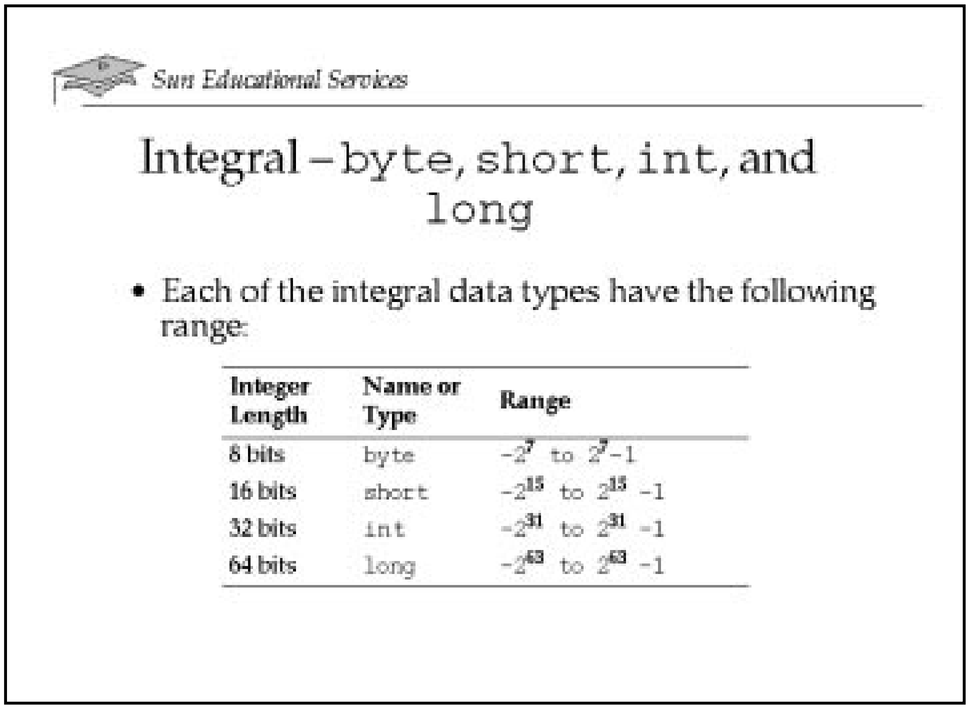
**Note –** All integral types in the Java programming language are signed numbers.

---

## *Integral* – `byte`, `short`, `int`, *and* `long` *(Continued)*

Integral literals are of type `int` unless explicitly followed by the letter "L." The L indicates a `long` value. Note that it is valid in the Java programming language to use either an uppercase or lowercase L, but lowercase is a poor choice since it is usually hard to distinguish it from the digit 1. Long versions of the literals shown previously are:

| | |
|---|---|
| ● `2L` | The `L` indicates the decimal value two is represented as a long. |
| ● `077L` | The leading zero indicates an octal value. |
| ● `0xBAACL` | The `0x` prefix indicates a hexadecimal value. |



Sun Educational Services

## Integral – byte, short, int, and long

- Each of the integral data types have the following range:

| Integer Length | Name or Type | Range |
|---|---|---|
| 8 bits | byte | $-2^{7}$ to $2^{7}-1$ |
| 16 bits | short | $-2^{15}$ to $2^{15}$ -1 |
| 32 bits | int | $-2^{31}$ to $2^{31}$ -1 |
| 64 bits | long | $-2^{63}$ to $2^{63}$ -1 |

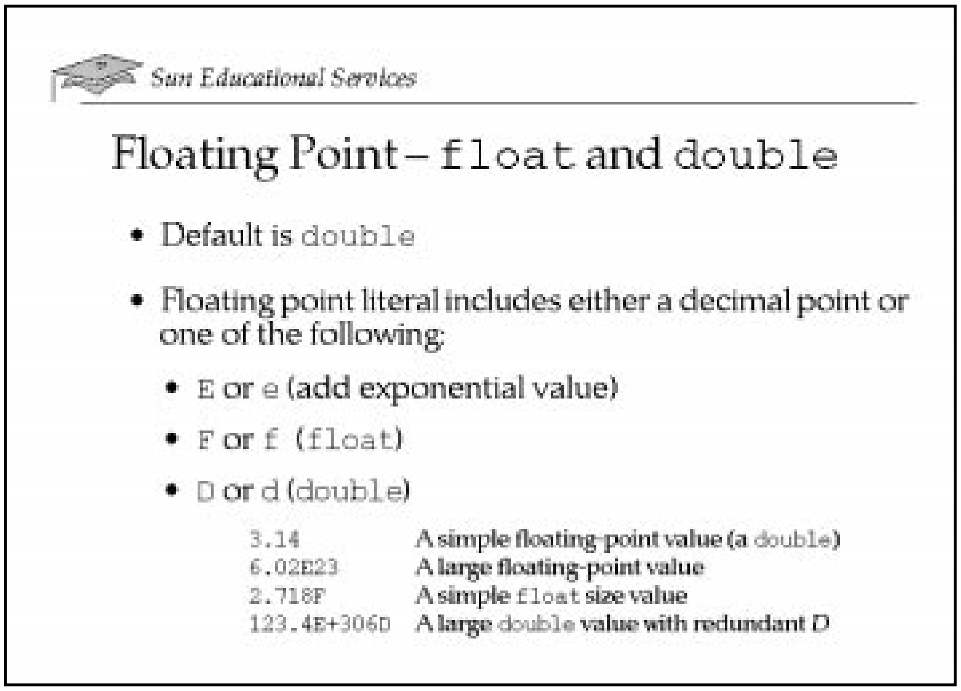
## Basic Java Types

### *Integral* – `byte`, `short`, `int`, *and* `long` *(Continued)*

The size and range for the four integral types are shown in Table 2-2. The range representation is defined by the Java programming language specification as a two's complement and is platform independent.

**Table 2-2** Integral Data Types – Range

| Integer Length | Name or Type | Range |
|---|---|---|
| 8 bits | `byte` | $-2^{7}$ to $2^{7}$ -1 |
| 16 bits | `short` | $-2^{15}$ to $2^{15}$ -1 |
| 32 bits | `int` | $-2^{31}$ to $2^{31}$ -1 |
| 64 bits | `long` | $-2^{63}$ to $2^{63}$ -1 |


Sun Educational Services

## Floating Point – float and double

- Default is double
- Floating point literal includes either a decimal point or one of the following:
  - E or e (add exponential value)
  - F or f (float)
  - D or d (double)

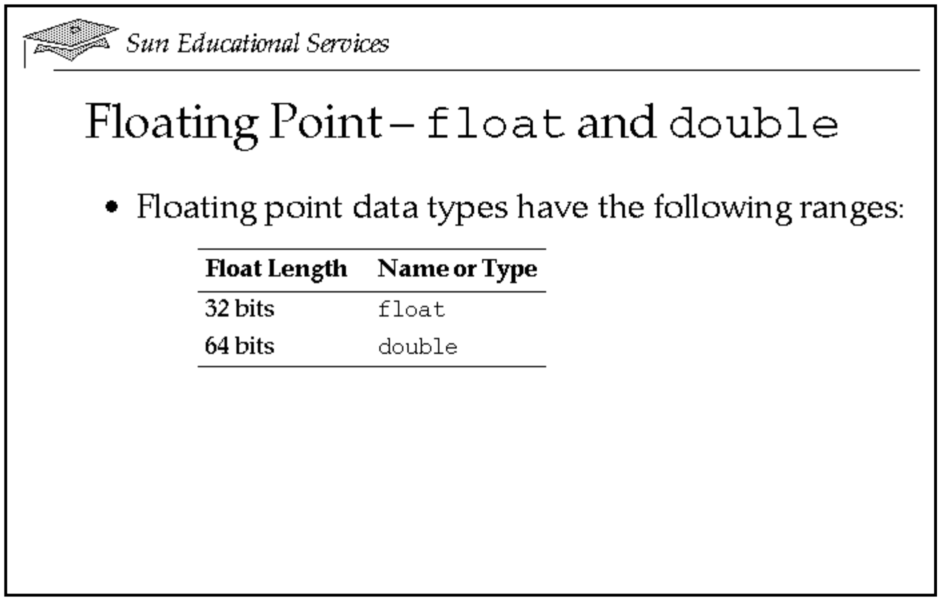
| | |
|---|---|
| 3.14 | A simple floating-point value (a double) |
| 6.02E23 | A large floating-point value |
| 2.718F | A simple float size value |
| 123.4E+306D | A large double value with redundant D |

## *Basic Java Types*

### *Floating Point –* `float` *and* `double`

A floating point variable can be declared using the keyword `float` or `double`. The followin list contains examples of floating point numbers. A numeric literal is a floating point if it includes either a decimal point or an exponent part (the letter *E* or *e*), or is followed by the letter *F* or *f* (float) or the letter *D* or *d* (double).

| | |
|---|---|
| ● 3.14 | A simple floating-point value (a double) |
| ● 6.02E23 | A large floating-point value |
| ● 2.718F | A simple `float` size value |
| ● 123.4E+306D | A large `double` value with redundant D |



Sun Educational Services

## Floating Point – `float` and `double`

- Floating point data types have the following ranges:

| Float Length | Name or Type |
|---|---|
| 32 bits | `float` |
| 64 bits | `double` |

## *Basic Java Types*

### *Floating Point –* `float` *and* `double` *(Continued)*

Floating Point Data Types – RangeThe format of a floating point number is defined by the Java technology specification to be Institute of Electrical and Electronics Engineers (IEEE) 754, using the sizes shown in , and is platform independent.

**Table 2-3**

| Float Length | Name or Type |
|---|---|
| 32 bits | `float` |
| 64 bits | `double` |

**Note –** Floating point literals are `double` unless explicitly declared as `float`.

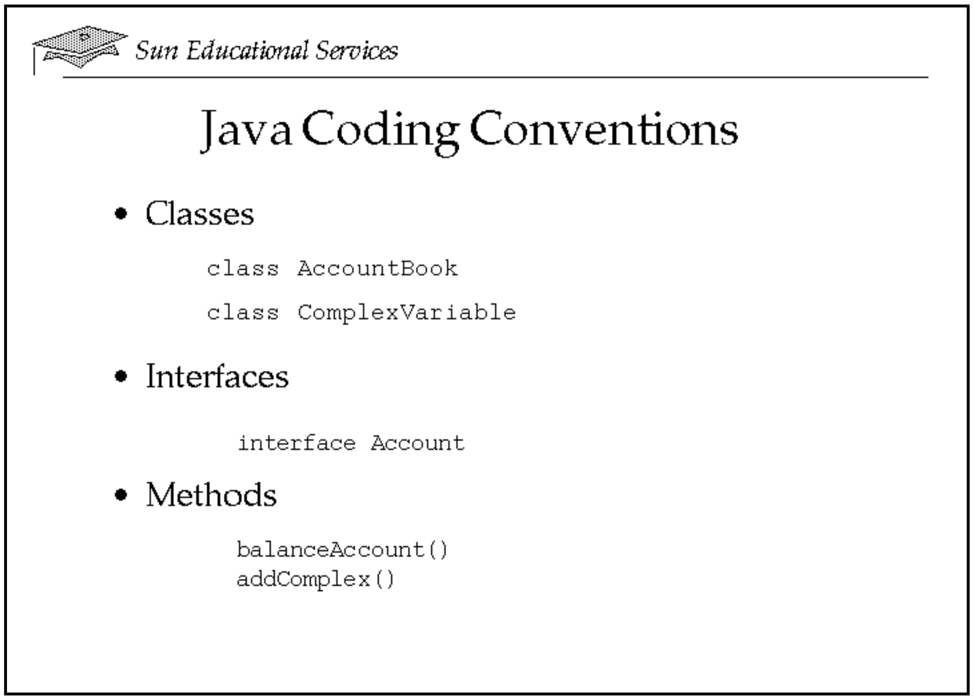
## Variables, Declarations and Assignments

The following program illustrates how to declare and assign values to `int`, `float`, `boolean`, `char`, and `String` type variables:

```
public class Assign {
  public static void main (String args []) {
    // declare integer variables
    int x, y;
    // declare and assign floating point
    float z = 3.414f;
    // declare and assign double
    double w = 3.1415;
    // declare and assign boolean
    boolean truth = true;
    // declare character variable
    char c;
    // declare String variable
    String str;
    // declare and assign String variable
    String str1 = "bye";
    // assign value to char variable
    c = 'A';
    // assign value to String variable
    str = "Hi out there!";
    // assign values to int variables
    x = 6;
    y = 1000;
  }
}
```

Some examples of illegal assignments are

```
y = 3.1415926;        // 3.1415926 is not an int; It
                      // requires casting and decimal will
                      // be truncated.
w = 175,000;          // The comma symbol (,) cannot appear;
truth = 1;            // this is a common mistake made by
                      // ex C / C++ programmers
z = 3.14156;          // Can’t fit double into a
                      // float; This requires casting.
```


Sun Educational Services

## Java Coding Conventions

- Classes

```
class AccountBook
class ComplexVariable
```

- Interfaces

```
interface Account
```

- Methods

```
balanceAccount()
addComplex()
```

## *Java Coding Conventions*

✓ ***The following terms are adapted from the early Java and Oak programming conventions. This is not a complete list and might be updated as JavaSoft updates its programming conventions over time.***

Some coding conventions of the Java programming language are

- *Classes* – Class names should be nouns, in mixed case, with the first letter of each word capitalized.

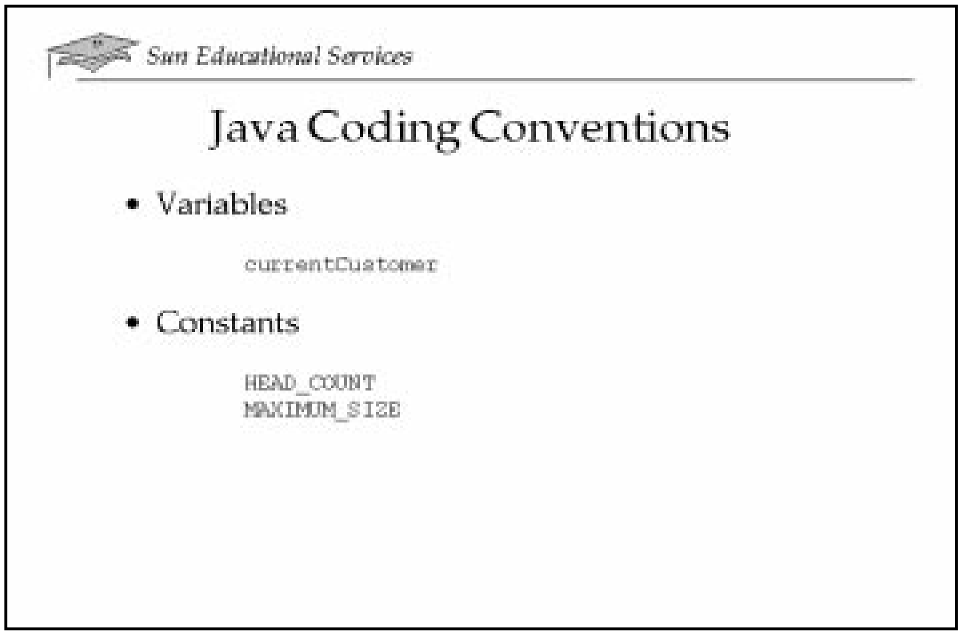
```
class AccountBook
class ComplexVariable
```

- *Interfaces* – Interface names should be capitalized like class names.

```
interface Account
```

- *Methods* – Method names should be verbs, in mixed case, with the first letter in lowercase. Within each method name, capital letters separate words. Limit the use of underscores.

```
balanceAccount()
addComplex()
```



Sun Educational Services

## Java Coding Conventions

- Variables

  currentCustomer

- Constants

  HEAD_COUNT
  MAXIMUM_SIZE

## Java Coding Conventions

✓ ***Native methods use underscores to create complex names; For example,*** `java.lang.String` ***becomes*** `java_lang_String`***.***

- *Variables* – All variables should be in mixed case with a lowercase first letter. Words are separated by capital letters. Limit the use of underscores, and avoid using the dollar sign ($) because this character has special meaning to inner classes.

```
currentCustomer
```

Variables should be meaningful and indicate to the casual reader the intent of their use. Single character names should be avoided except for temporary "throwaway" variables (for example, `i`, `j`, and `k`, used as loop control variables).

- *Constants* – Primitive constants should be all uppercase with the words separated by underscores. Object constants can use mixed-case letters.

```
HEAD_COUNT
MAXIMUM_SIZE
```

## Java Coding Conventions

- *Control structures* – Use braces ({ }) around all statements, even single statements, when they are part of a control structure such as an
  `if-else` or `for` statement.

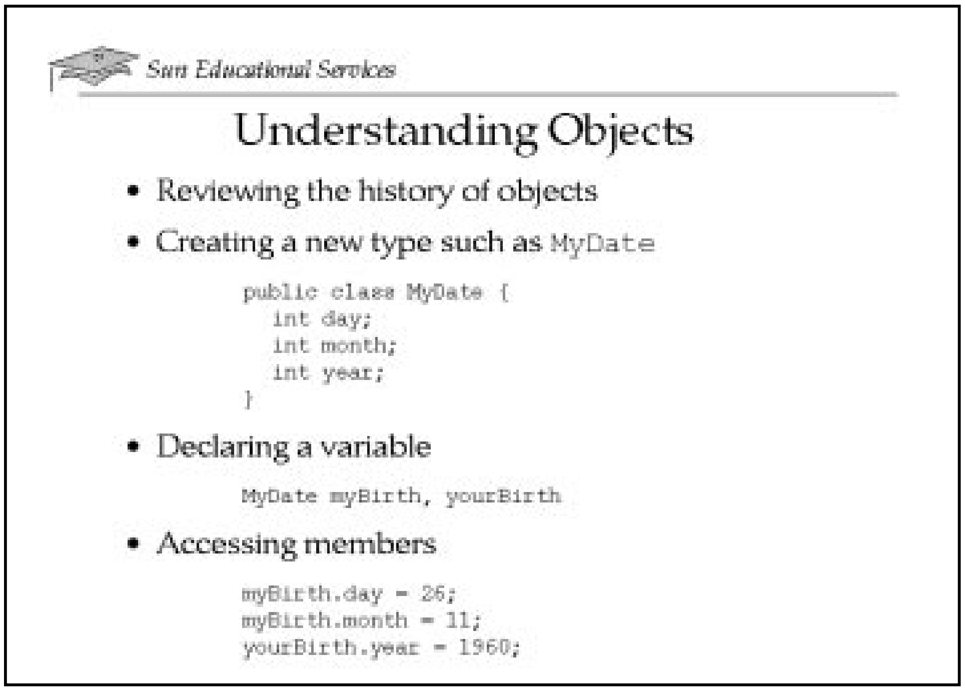
  ```
  if ( condition ) {
    do something
  } else {
    do something else
  }
  ```

- *Spacing* – Place only a single statement on any line, and use two or four-space indentations to make your code readable. The number of spaces may vary depending on whose code standards are used.

- *Comments* – Use comments to explain code segments that are not obvious. Use the // comment delimiter for normal commenting; large sections of code can be commented using the /* . . . */ delimiters. Use the /** . . . */ documenting comment to provide input to `javadoc` for generating HTML documentation for the code.

  ```
  // A comment that takes up only one line.

  /* Comments that continue past one line and take up
            space on multiple lines...*/

  /** A comment for documentation purposes.
   @see Another class for more information
  */
  ```

**Note –** `@see` is a special `javadoc` tag giving the effect of a "see also" link that references a class or method. For more information about `javadoc`, refer to the complete definition of the documentation system in "The Design of Distributed Hyperlinked Programming Documentation.", a paper by Lisa Friendly. It is available from `http://www.javasoft.com/doc/api_documentation.html`.



Sun Educational Services

## Understanding Objects

- Reviewing the history of objects
- Creating a new type such as `MyDate`

```
public class MyDate {
   int day;
   int month;
   int year;
}
```

- Declaring a variable

```
MyDate myBirth, yourBirth
```

- Accessing members

```
myBirth.day = 26;
myBirth.month = 11;
yourBirth.year = 1960;
```

## *Understanding Objects*

### *Reviewing the History of Objects*

Early programming languages and novice programmers tend to treat individual variables as isolated entities. For example, if a program needs to handle a date, three separate integers would be declared:

```
int day, month, year;
```

This statement does two things. It says that when the program refers to a day, month, or year, it will be manipulating an integer. It also allocates the storage for those integers.

Although this approach is simple to understand, it has two significant drawbacks. First, if the program needs to keep track of several dates, then three more declarations are needed for each date. To keep two birthdays, you might use

```
int myBirthDay, myBirthMonth, myBirthYear;
int yourBirthDay, yourBirthMonth, yourBirthYear;
```

## Understanding Objects

### Reviewing the History of Objects (Continued)

This method quickly becomes messy because of the number of names required.

The second weakness is that this scheme ignores the association between a day, a month, and a year and treats each as an independent value. Each variable is part of a single unit (in this case, a date) and should be handled accordingly.

### Creating a New Type

In order to address the two weaknesses, the Java programming language uses a `class` to create new types. Consider the following primitive type declaration:

```
int day;
```

The Java programming language is used to allocate a certain amount of storage and interpret the contents of that storage. So, to define a new type, you must specify how much storage is required and how to interpret the contents. This is done not in terms of numbers of bytes or ordering and meaning of bits, but in terms of other types that are already defined.

For example, to define a type that represents a date, you need enough storage for three integers. Furthermore, the significance of day, month, and year is attributed to these integers. For example:

```
class MyDate {
  int day;
  int month;
  int year;
}
```

The word `class` is a keyword in the Java programming language and must be written entirely in lowercase. The name `MyDate` is capitalized by convention rather than by language requirement.

## Understanding Objects

### Creating a New Type (Continued)

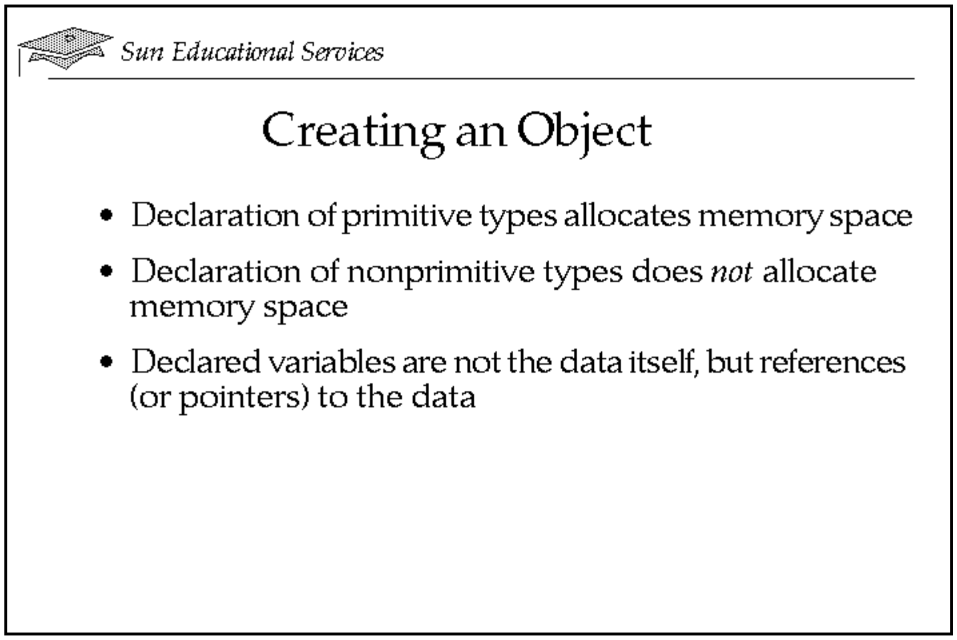
A variable can be declared as being of type `MyDate`, and the day, month, and year parts will be implied by this declaration. For example:

```
MyDate myBirth, yourBirth;
```

Using this declaration, the Java programming language allows the parts (`day`, `month` and `year`) of the variables, called the members, to be accessed using the dot (.) operator. For example:

```
myBirth.day = 26;
myBirth.month = 11;
yourBirth.year = 1960;
```

✓ ***Assess your audience. If many are C or C++ programmers, you could explain this in terms of aggregate data types. Other types of programmers might be comfortable with the terms*** structured types ***or*** record types***.***



Sun Educational Services

## Creating an Object

- Declaration of primitive types allocates memory space
- Declaration of nonprimitive types does *not* allocate memory space
- Declared variables are not the data itself, but references (or pointers) to the data

## Understanding Objects

### Creating an Object

When variables of any primitive type—that is, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, or `double` types—are declared, the memory space is allocated as part of the operation. The declaration of a variable, using a non-primitive type such as `String` or `Date`, does *not* allocate memory space for the object.

In fact, a variable that is declared with a class type is not the data itself, but rather a *reference* to the data.

---

**Note –** You can also think of the reference as a *pointer*—in most implementations it is just that, and if the terminology is meaningful to you, it will help your understanding. It should be noted that the Java programming language does not actually support the pointer data type.

---

# Understanding Objects

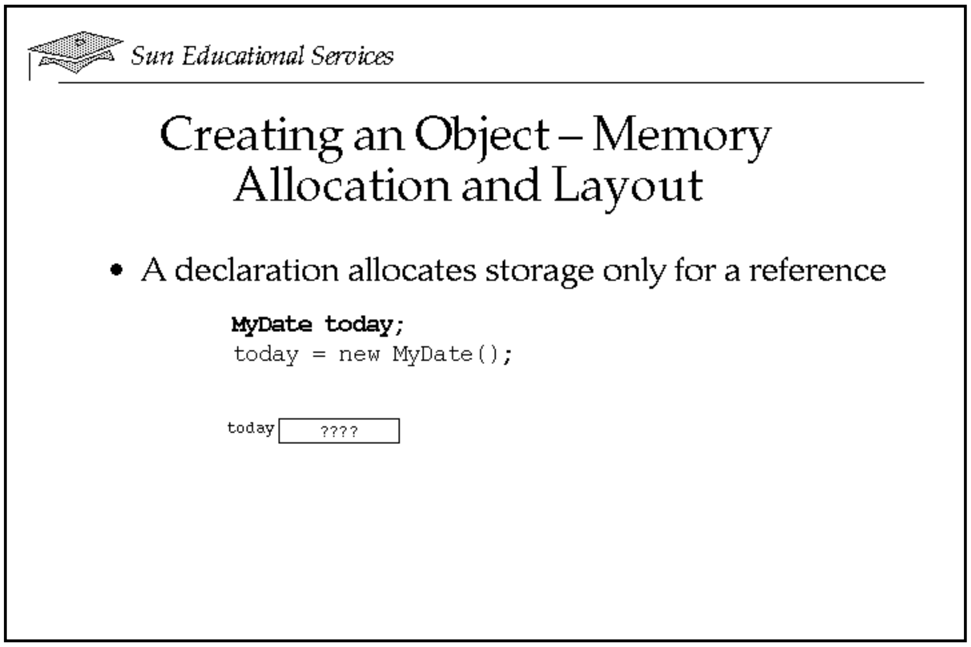
## Creating an Object (Continued)

Before you can use the variable, the actual storage must be allocated. This is done by using the keyword `new` as follows:

```
MyDate myBirth;
myBirth = new MyDate();
```

The first statement, the declaration, allocates just enough space for the *reference*. The second statement allocates the space, called an object, for the three integers used to form `MyDate`. Object assignment sets up the variable `myBirth` to refer correctly to the new *object*. After these two operations have been performed, the contents of the `MyDate` object can be accessed through `myBirth`.

Given a class definition for an arbitrary class `Xxxx`, you can call `new Xxxx()` to create as many objects as you need. Each is separate from the others. A reference to the object can be stored in a variable so that a combination of "variable dot *member*" (such as `myBirth.day`) can be used to access the individual members of each of the objects. Note that it is possible to make use of objects without having a reference to them. Such objects are called *anonymous* objects.



Sun Educational Services

## Creating an Object – Memory Allocation and Layout

- A declaration allocates storage only for a reference

```
MyDate today;
today = new MyDate();
```

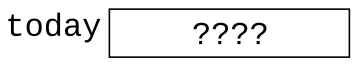
## Understanding Objects

### Creating an Object – Memory Allocation and Layout

In a method body, the declaration

```
MyDate today;
today = new MyDate();
```

allocates storage only for the reference

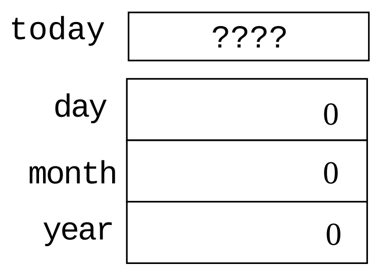


## Understanding Objects

### Creating an Object – Memory Allocation and Layout (Continued)

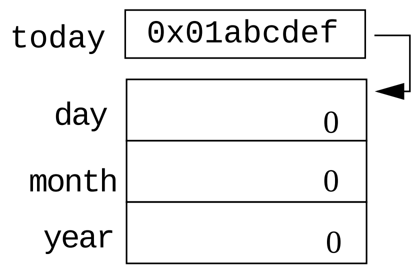
The keyword `new` implies allocation and initialization of storage.

```
MyDate today;
today = new MyDate();
```



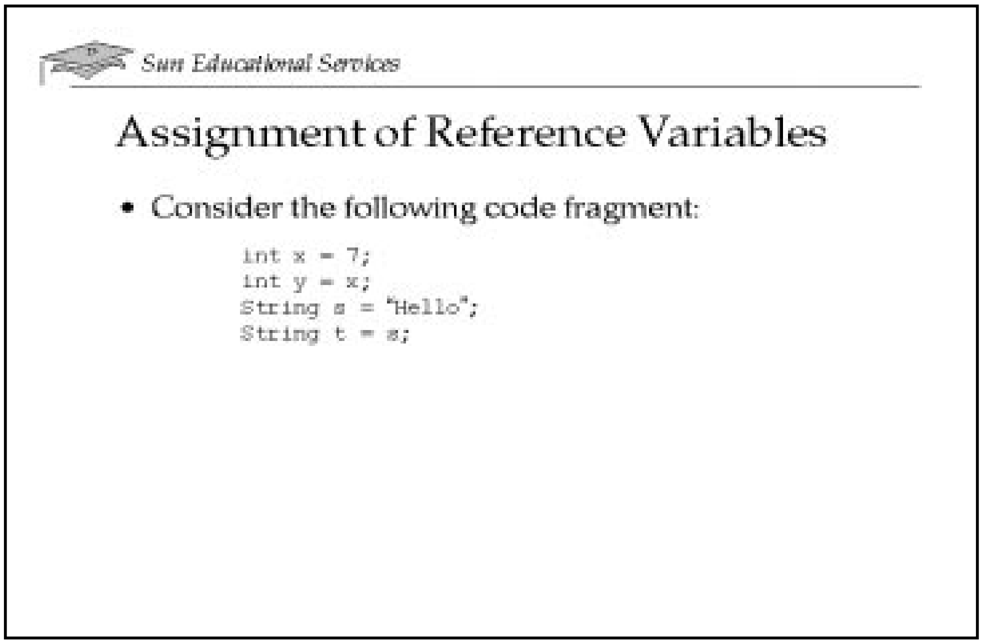
The assignment then sets up the reference variable so that it refers properly to the newly created object.

```
MyDate today;
today = new MyDate();
```



It is also possible to allocate space for both the reference `today` and the object referred to by the reference `today` using one statement.

```
MyDate today = new MyDate();
```



Sun Educational Services

## Assignment of Reference Variables

- Consider the following code fragment:

```
int x = 7;
int y = x;
String s = "Hello";
String t = s;
```

## *Understanding Objects*

### *Assignment of Reference Types*

In the Java programming language, a variable declared with a type of class is referred to as a reference type. This is because it is refers to a non-primitive type. This has consequences for the meaning of assignment. Consider this code fragment:

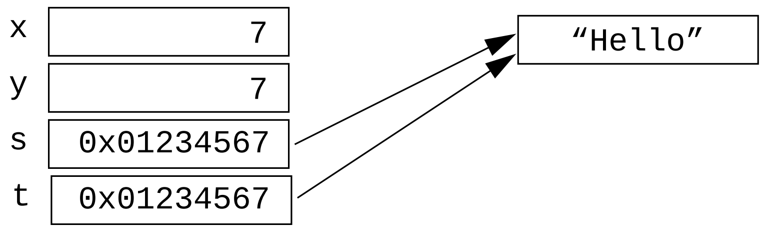
```
int x = 7;
int y = x;
String s = "Hello";
String t = s;
```

Four variables are created: two primitives of type `int` and two references of type `String`. The value of *x* is seven, and this value is copied into *y*. Both *x* and *y* are independent variables and further changes to either do not affect the other.

# Understanding Objects

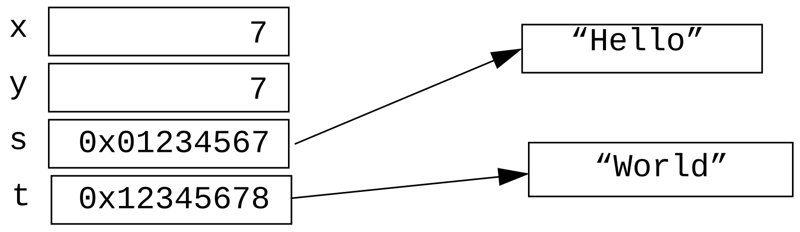
## Assignment of Reference Types (Continued)

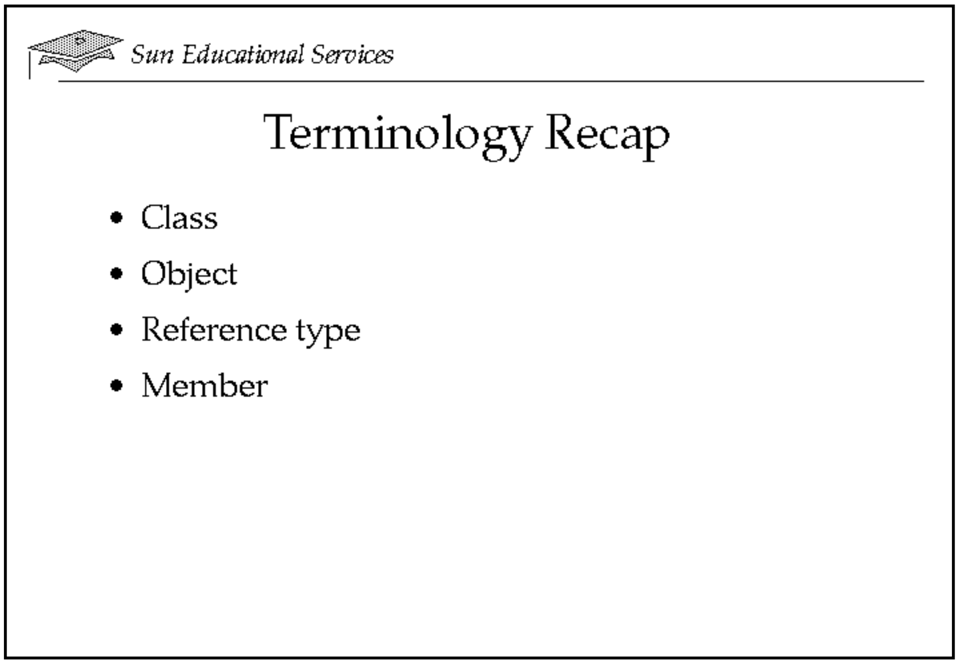
With the variables *s* and *t*, only one `String` object exists and it contains the text "Hello". Both *s* and *t* refer to that single object.



With a reassignment of the variable *t*, the new object `World` is created and *t* refers to this object. This scenario is depicted as

```
t = "World";       // reassign the variable
```





Sun Educational Services

## Terminology Recap

- Class
- Object
- Reference type
- Member

## *Understanding Objects*

### *Terminology Recap*

Some of the terms introduced in this module are:

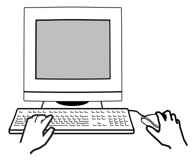
- Class – A way to define new types in the Java programming language. The class declaration defines new types and describes how those types are implemented. There are many additional features to a class that have not yet been discussed.

## Understanding Objects

### Terminology Recap (Continued)

- Object – An actual instance of a class. The class can be considered as a template—a model of the object you are describing. An object is what you get each time you create an instance of a class using `new`.
- Reference type – A user defined type that refers to an object of a class, interface, or array.
- Member – One of the elements that makes up an object. The term is also used for elements of the defining class. The terms *member variable, instance variable*, and *field* are used interchangeably.

## Exercise: Using Objects



**Exercise objective –** Using the correct Java keywords, write a program to create a class and an object from the class. Compile and run the program; then, verify that the references are assigned and manipulated as described in this module.

### Preparation

In order to successfully complete this lab, you must be able to compile and run a Java program. You also need to be familiar with the object-oriented concepts of classes and objects, and with the concepts of references.

### Tasks

#### *Level 1 Lab: Create a Class and Corresponding Objects*

Complete the following steps:

1. A point can be characterized by an x and y coordinate. Define a class called `MyPoint` that represents this idea. What should you call the file?
2. Write a `main()` method *inside* your class; below that, declare two variables of type `MyPoint`. Call the variables `start` and `end`. Create the objects using `new MyPoint()` and assign the reference values into the variables `start` and `end`, respectively.
3. Assign the value 10 to the `x` and `y` members of the object `start`.
4. Assign the value 20 to the `x` value of the `end` object, and the value 30 to the `y` value of the `end` object.

## Exercise: Using Objects

### Tasks

#### *Level 1 Lab: Create a Class and Corresponding Objects (Continued)*

5. Print out the values of both members (`x` and `y`) of each of the `MyPoint` objects (`start` and `end`).

---

**Note –** To complete step 5, you need to know more about the `System` class. `System.out.println()` with a `String` argument outputs a string and starts a new line. `System.out.print()` does not start a new line. If you use `System.out.print()`, call either `System.out.println()` or `System.out.flush()` before the application terminates, otherwise, you might find the last line of output is not displayed.

---

To display numbers, you can use the following form (which will be described later in this course):

```
System.out.println("Start MyPoint = X: " +
                   start.x + " Y: " + start.y);
```

---

**Note –** The plus symbol (+) converts one operand into a `String` if the other is already a `String`.

---

6. Compile and run the program.

✓ ***C and C++ users: There is no simple equivalent of `printf()` and its formatting facilities in the Java programming language.***

## Exercise: Using Objects

### Tasks

#### Level 2 Lab: Investigate Reference Assignment

Using the `MyPoint` class you created in the previous exercise, add code to the `main()` method to do the following:

1. Declare a new variable of type `MyPoint` and call it `stray`. Assign `stray` the reference value of the existing variable `end`.

2. Print out the values of the `x` and `y` members for both `end` and `stray` variables.

3. Assign new values to both the `x` and `y` members of the variable `stray`.

4. Print out the values of the members of both `end` and `stray`. Compile and run the `MyPoint` class. The values reported by `end` reflect the change made in `stray`, indicating that both variables refer to the same `MyPoint` object.

5. Assign new values to both the `x` and `y` members of the `start` variable.

6. Print out the values of the members of both `start` and `end`. Compile and run the `MyPoint` class again. The values of `start` are still independent of those in `stray` and `end`, indicating that the `start` variable still refers to a `MyPoint` object that is different from the one referred to by `stray` and `end`.

## Exercise: Using Objects

### Exercise Summary

**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Use comments in a source program
- ❑ Distinguish between valid and invalid identifiers
- ❑ Recognize Java technology keywords
- ❑ List the eight primitive types
- ❑ Define literal values for numeric and textual types
- ❑ Define the terms *class, object*, *member variable,* and *reference variable*
- ❑ Create a class definition for a simple class containing primitive member variables
- ❑ Declare variables of class type
- ❑ Construct an object using `new`
- ❑ Describe default initialization
- ❑ Access the member variables of an object using the dot notation
- ❑ Describe the significance of a reference variable
- ❑ State the consequences of assigning variables of class type

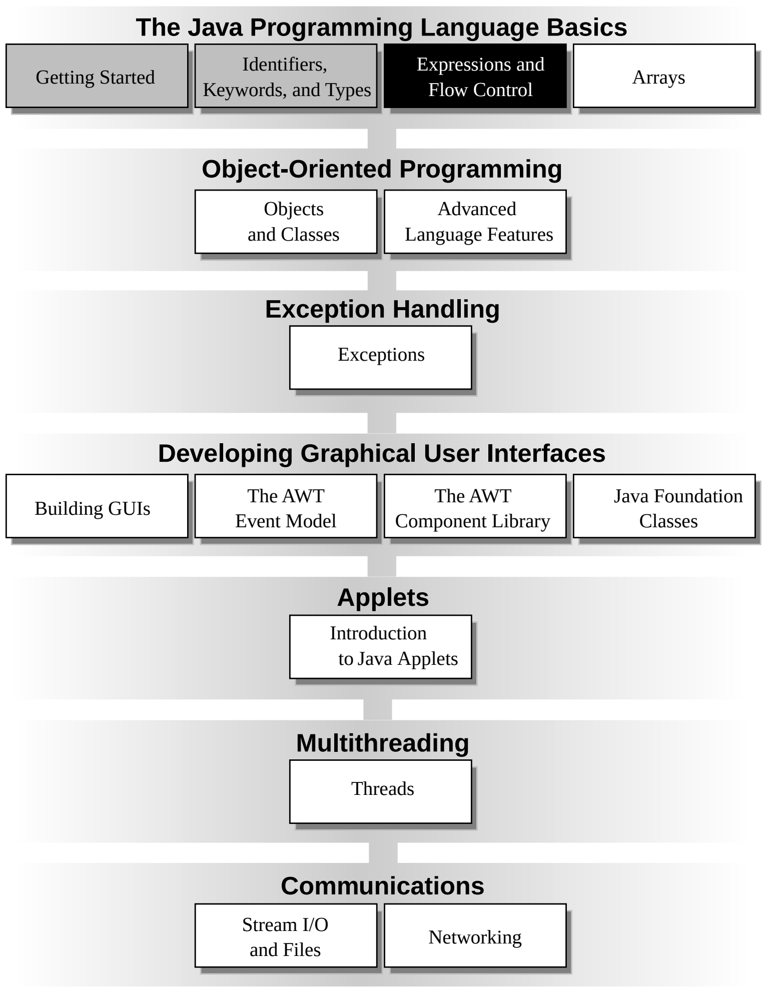
## Think Beyond

Can you think of examples of classes and objects in your existing applications?

# Expressions and Flow Control 3

## Course Map

This module discusses variables, operators, and arithmetic expressions; and lays out the different control structures governing the path of execution.



The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following questions are relevant to the material presented in this module:

- What types of variables are useful to programmers (for instance, programmers of other languages will want to know how the Java programming language defines and handles global and local variables)?
- Can multiple classes have variables with the same name and, if so, what are their scope?

✓ ***Good use of variables greatly improves a computer program all around, from execution efficiency to maintainability. This module discusses the implications of where and how variables are declared, and the implications of how they are used in expressions.***

- List the types of control structures used in other languages. What methods do languages in general employ for flow control and for discontinuing the flow (such as in a loop or switch)?

✓ ***Control structures are what govern the path of program execution. Like variables, good use of control structures yields efficient, straightforward programs. In*** `while` ***loops, certain conditions have to be met in order to exit. In switch controls, a break is used to exit. Older languages used the*** `goto` ***statement, for lack of a better way, to control the program flow. The Java programming language does not use*** `goto`.

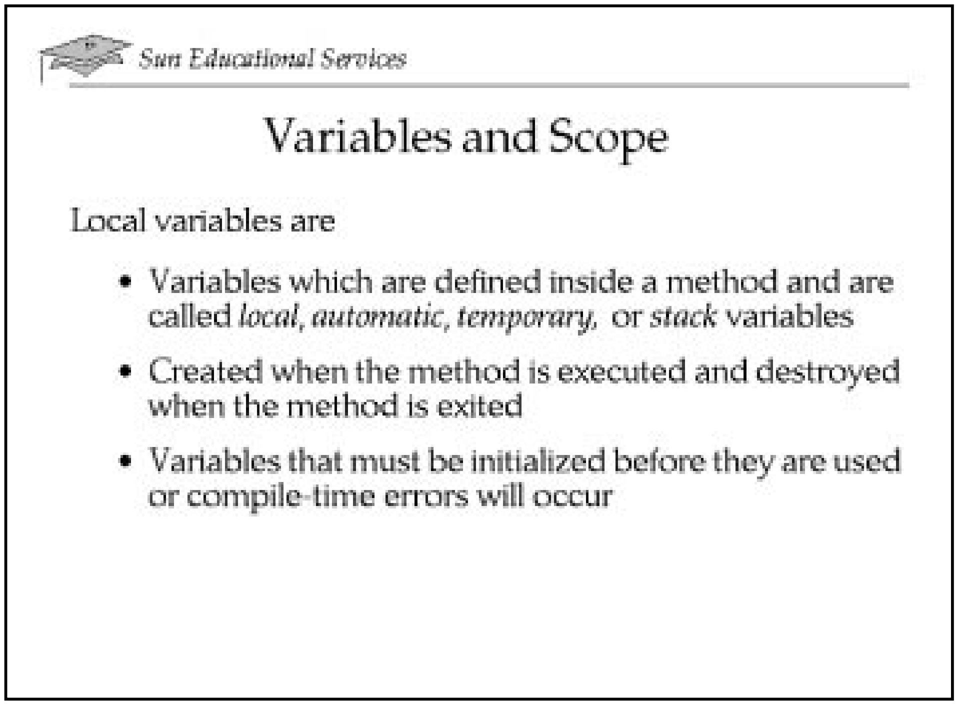
✓ ***Focus the students' attention on the following areas:***

- ***Use variables effectively.***
  - ***Determine the implications of how and where variables are declared.***
  - ***Learn the implications of how variables are used.***
  - ***Learn about operator usage and expressions in a Java software program.***
  - ***Control a Java software program's execution path with control structures.***

## Objectives

Upon completion of this module, you should be able to

- Distinguish between instance and local variables
- Describe how instance variables are initialized
- Identify and correct a `Possible reference before assignment` compiler error
- Recognize, describe, and use Java software operators
- Distinguish between legal and illegal assignments of primitive types
- Identify `boolean` expressions and their requirements in control constructs
- Recognize assignment compatibility and required casts in fundamental types
- Use `if`, `switch`, `for`, `while`, and `do` constructions and the labeled forms of `break` and `continue` as flow control structures in a program



Sun Educational Services

## Variables and Scope

Local variables are

- Variables which are defined inside a method and are called *local, automatic, temporary,* or *stack* variables
- Created when the method is executed and destroyed when the method is exited
- Variables that must be initialized before they are used or compile-time errors will occur

## Expressions

### Variables and Scope

You have seen two ways that variables can be described: variables of primitive type or variables of reference type. You have also seen two places variables can be declared: inside a method (a *method* is an object-oriented term which refers to a function or subroutine such as `main()`) or outside a method but within a *class* definition. Variables can also be defined as method parameters or constructor parameters.

✓ ***Methods are covered more thoroughly in Module 5, "Objects and Classes."***

Variables defined inside a method are called *local* variables, but are sometimes referred to as *automatic*, *temporary*, or *stack* variables.

## Expressions

### Variables and Scope

Variables defined outside a method are created when the object is constructed using the `new Xxxx()` call. There are two possible kinds of variables. The first kind is a class variable which is declared using the `static` keyword. This is done when the class is loaded. Class variables continue to exist for as long as the class exists. The second kind is an instance variable which is declared without the `static` keyword. Instance variables continue to exist for as long as the object is referenced. Instance variables are sometimes referred to as member variables, as they are members of the class. The `static` variable will be discussed later in this course in more detail.

Method parameter variables define arguments passed in a method call. Each time the method is called, a new variable is created and itlasts only until the method is exited.

## Variables and Scope (Continued)

Local variables are created when execution enters the method, and are destroyed when the method is exited. This is why local variables are sometimes referred to as “temporary or automatic.” Variables that are defined within a member function are local to that member function, so you can use the same variable name in several member functions to refer to different variables. This is illustrated in the following example:

```
class OurClass {

  // instance variable of class OurClass
  int i;

  int firstMethod() {

    // local variable
    int j = 0;

    // Both i and j are accessable here

    return 1;
  } // end of firstMethod

  int secondMethod() {

    // local variable, but different than j defined in
    // firstMethod().
    int j = 0;

    // Scope is limited to the body of secondMethod().
    // Both i (instance variable of the class OurClass)
    // and j (local variable of this method) are
    // accessible here

    return 2;
  } // end of secondMethod

} // end of OurClass
```

## Expressions

### Variable Initialization

No variable in a Java program may be used before being initialized. When an object is created, instance variables are initialized with the following values at the time the storage is allocated:

| | |
|---|---|
| `byte` | `0` |
| `short` | `0` |
| `int` | `0` |
| `long` | `0L` |
| `float` | `0.0F` |
| `double` | `0.0D` |
| `char` | `'\u0000' (NULL)` |
| `boolean` | `false` |
| All reference types | `null` |

**Note –** A reference that has the null value refers to no object. An attempt to use the object it refers to will cause an exception. Exceptions are errors that occur at runtime and are discussed in a later module.

While variables defined outside of a method are initialized automatically, local variables *must* be initialized manually before use. The compiler flags an error if it can determine a condition where a variable can be used before being initialized.

```
public void doComputation() {
   int x = (int)(Math.random() * 100);
   int y;
   int z;
   if (x > 50) {
      y = 9;
   }
   z = y + x;  // Possible use before initialization
}
```

# Expressions

## Operators

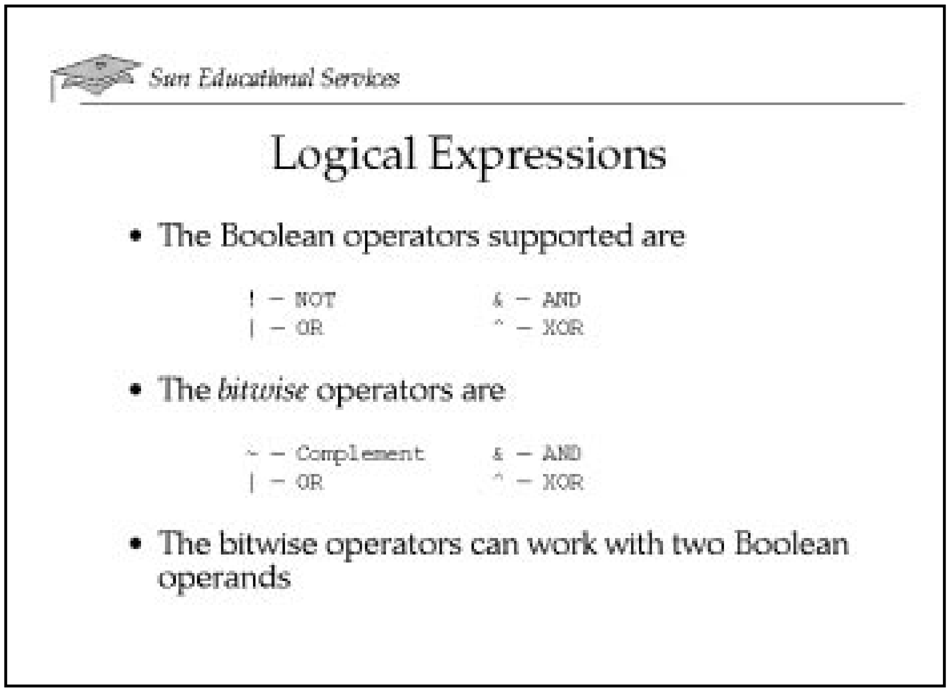
The Java software operators are very similar in style and function to those of C and C++. Table 3-1 lists the operators in order of precedence (L to R means left-to-right associative; R to L means right-to-left associative):

**Table 3-1** Operators in Order of Precedence

| | |
|---|---|
| Separator | `.   []   ()   ;   ,` |

| | |
|---|---|
| R to L | `++  -- + - ~ ! (data type)` |
| L to R | `*  /  %` |
| L to R | `+  -` |
| L to R | `<<  >>  >>>` |
| L to R | `<  >  <=  >= instanceof` |
| L to R | `==  !=` |
| L to R | `&` |
| L to R | `^` |
| L to R | `\|` |
| L to R | `&&` |
| L to R | `\|\|` |
| R to L | `?:` |
| R to L | `=   *=   /=   %=   +=   -=   <<=`<br>`>>=   >>>=   &=   ^=   \|=` |

**Note –** The `instanceof` operator is unique to the Java programming language and will be discussed in the Module 5, “Objects and Classes."


Sun Educational Services

## Logical Expressions

- The Boolean operators supported are

```
! – NOT          & – AND
| – OR           ^ – XOR
```

- The *bitwise* operators are

```
~ – Complement   & – AND
| – OR           ^ – XOR
```

- The bitwise operators can work with two Boolean operands

## *Expressions*

### *Logical Expressions*

Most Java operators are taken from other languages and behave as expected.

Relational and logical operators return a `boolean` result. There is no automatic conversion of `int` to `boolean`.

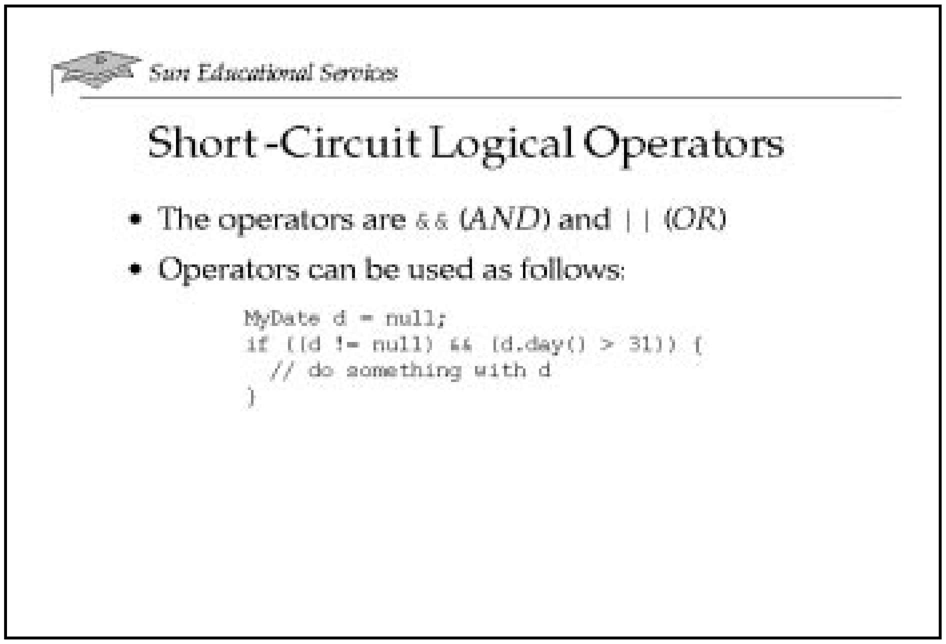
```
int i = 1;
if (i) // generates a compile error
if (i != 0)// Correct
```

The `boolean` operators supported are ! , & , ^ , and | for the algebraic Boolean operations NOT, AND, XOR, and OR, respectively. Each of these returns a `boolean` result. The operators && and `||` are the short circuit equivalents of the operators & and `|`. Short circuit logical operators are discussed on the following page.

# Expressions

## Bitwise Operations

The Java programming language supports bitwise operations on integral data types. These are represented as the operators ~ , & , ^ , and | for the bitwise operations of NOT (bitwise complement), bitwise AND, bitwise XOR, and bitwise OR, respectively. The bit shift operators are discussed later in this course.



Sun Educational Services

## Short -Circuit Logical Operators

- The operators are && (*AND*) and || (*OR*)
- Operators can be used as follows:

```
MyDate d = null;
if ((d != null) && (d.day() > 31)) {
  // do something with d
}
```

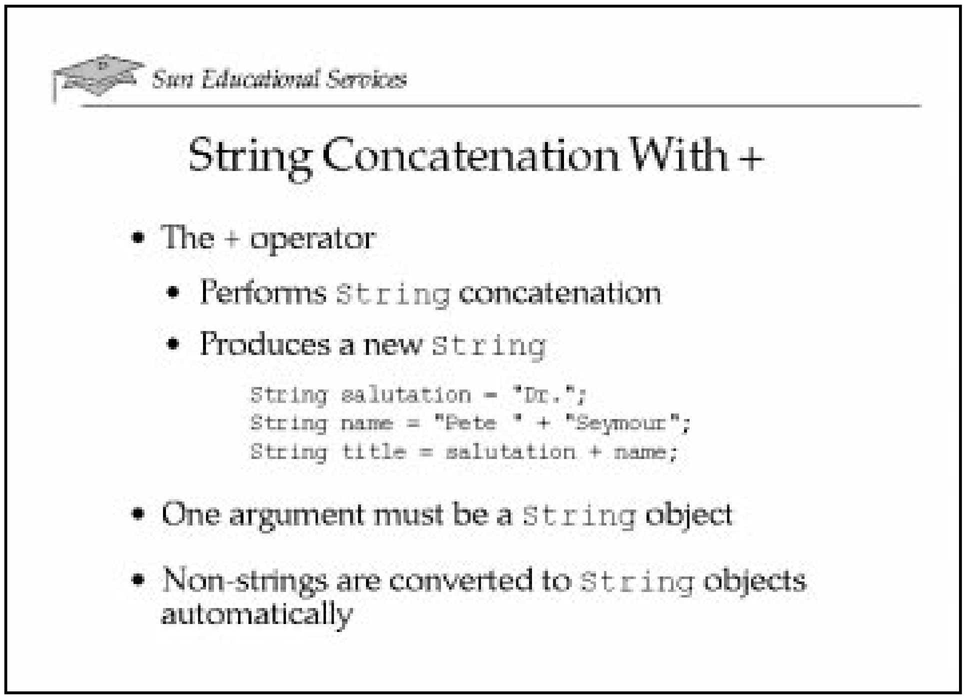
## *Expressions*

### *Short-Circuit Logical Operators*

The operators && (defined as AND) and || (defined as OR) perform *short-circuit* logical expressions. Consider this example:

```
MyDate d = null;
if ((d != null) && (d.day() > 31)) {
  // do something with d
}
```

The `boolean` expression that forms the argument to the `if ()` statement is legal and entirely safe. This is because the second subexpression is skipped when the first subexpression is false, since the entire expression will always be false when the first subexpression is false, regardless of how the second subexpression would evaluate. Similarly, if the || operator is used and the first expression returns true, the second expression would not be evaluated because the whole expression would already be known to be true.


Sun Educational Services

## String Concatenation With +

- The + operator
  - Performs String concatenation
  - Produces a new String

```
String salutation = "Dr.";
String name = "Pete " + "Seymour";
String title = salutation + name;
```

- One argument must be a String object
- Non-strings are converted to String objects automatically

## Expressions

### String Concatenation With +

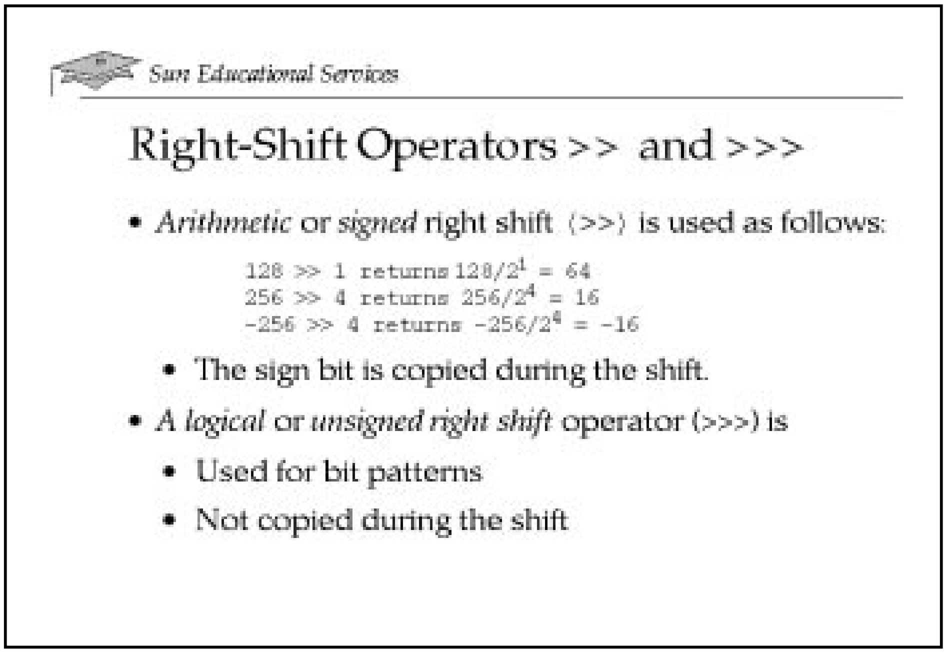
The + operator performs concatenation of String objects, producing a new String.

```
String salutation = "Dr. ";
String name = "Pete " + "Seymour";
String title = salutation + name;
```

The result of the last line is

```
Dr. Pete Seymour
```

If either argument of the + operator is a String object, then the other is converted to a String. All objects can be converted to a String automatically, although the result might be rather cryptic. The object that is not a string is converted to a string equivalent using the toString() member function.


Sun Educational Services

## Right-Shift Operators >> and >>>

- *Arithmetic* or *signed* right shift (>>) is used as follows:

```
128 >> 1 returns 128/2^1 = 64
256 >> 4 returns 256/2^4 = 16
-256 >> 4 returns -256/2^4 = -16
```

  - The sign bit is copied during the shift.
- *A logical* or *unsigned right shift* operator (>>>) is
  - Used for bit patterns
  - Not copied during the shift

## *Expressions*

### *Right-Shift Operators >> and >>>*

The Java programming language provides two right-shift operators.

The operator >> performs an *arithmetic* or *signed* right shift. The result of this shift is that the first operand is divided by two raised to the number of times specified by the second operand. For example:

```
128 >> 1 returns 128/2^1 = 64
256 >> 4 returns 256/2^4 = 16
-256 >> 4 returns -256/2^4 = -16
```

The >> operator results in the sign bit being copied during the shift.

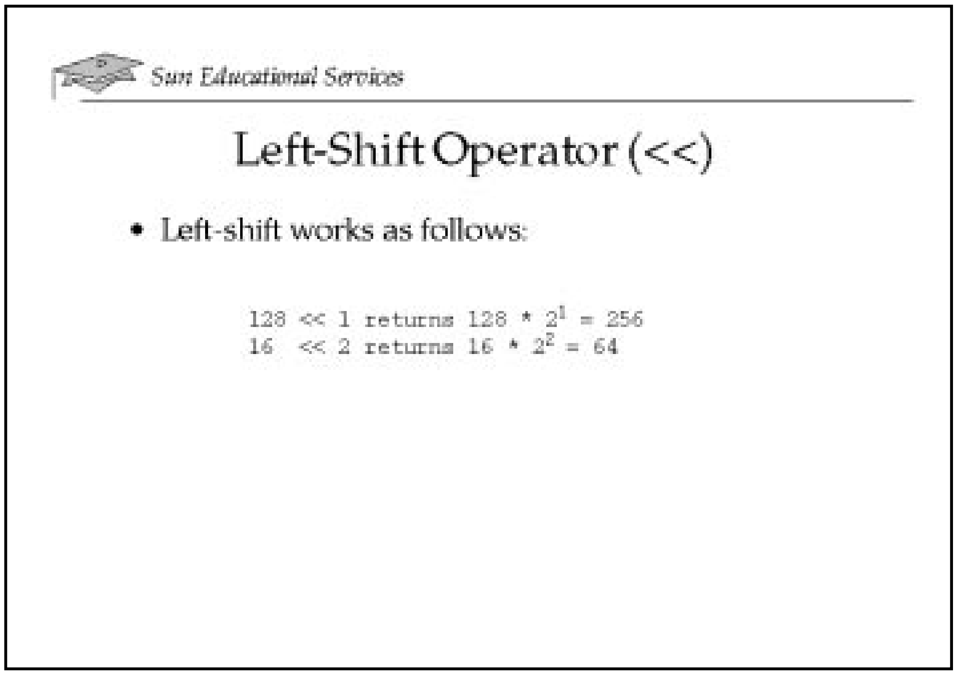
The *logical* or *unsigned* right shift operator >>> works on the bit pattern rather than the arithmetic meaning of a value and always places zeros in the most significant bits. For example:

```
1010 ... >> 2 gives 111010 ...
1010 ... >>> 2 gives 001010 ...
```

## Right-Shift Operators >> and >>> (Continued)

**Note –** The shift operators reduce their right-hand operand modulo 32 for an `int` type left-hand operand and modulo 64 for a `long` type right-hand operand. Therefore, for any `int x`, `x >>> 32` results in `x` being unchanged, not zero as you might expect.

**Note –** It is important to appreciate that the >>> operator is only permitted on integral types, and *is only effective on* `int` *or* `long` values. If used on a `short` or `byte` value, the value is promoted, with sign extension, to an `int` before >>> is applied. By this point, the unsigned shift has usually become a signed shift.



Sun Educational Services

## Left-Shift Operator (<<)

- Left-shift works as follows:

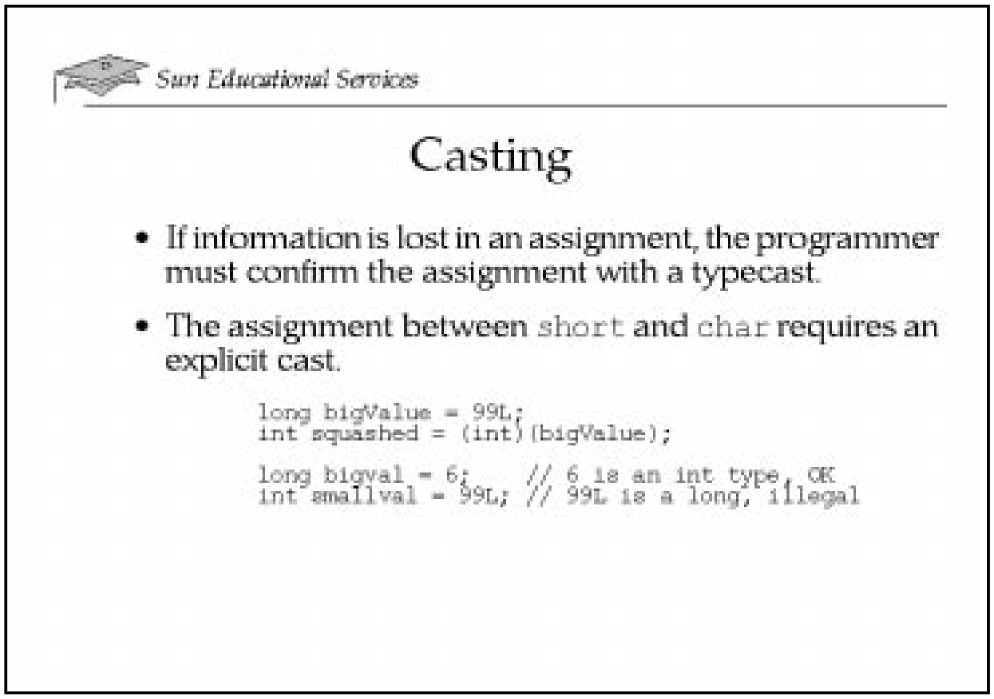
```
128 << 1 returns 128 * 2^1 = 256
16  << 2 returns 16 * 2^2 = 64
```

## Expressions

### Left-Shift Operator (<<)

The operator << performs *a* left shift. The result of this shift is that the first operand is multiplied by two raised to the number specified by the second operand. For example:

128 << 1 returns $128*2^1 = 256$
16 << 2 returns $16*2^2 = 64$


Sun Educational Services

## Casting

- If information is lost in an assignment, the programmer must confirm the assignment with a typecast.
- The assignment between `short` and `char` requires an explicit cast.

```
long bigValue = 99L;
int squashed = (int)(bigValue);

long bigval = 6;    // 6 is an int type, OK
int smallval = 99L; // 99L is a long, illegal
```

# Expressions

## Casting

Where information would be lost in an assignment, the compiler requires the programmer to confirm the assignment with a typecast. This can be done, for example, by "squeezing" a `long` value into an `int` variable. Explicit casting is done like this:

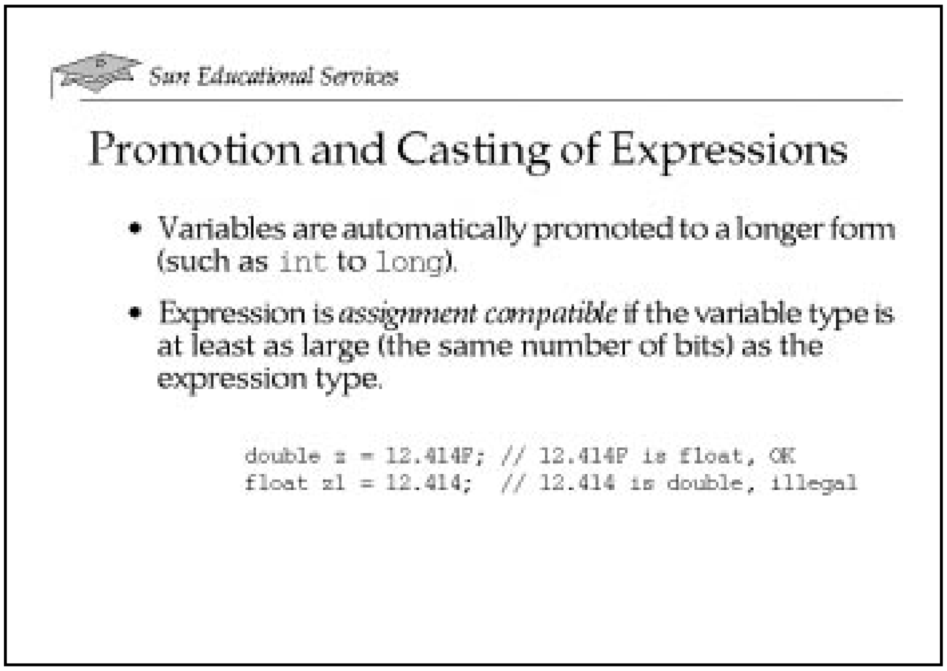
```
long bigValue = 99L;
int squashed = (int)(bigValue);
```

Observe that the desired target type is placed in parentheses and used as a prefix to the expression that must be modified. It might not be necessary, but is generally advisable to enclose the entire expression to be cast in parentheses. Otherwise, the precedence of the cast operation can cause problems.

---

**Note –** Recall that the range of `short` is $-2^{15}$ to $2^{15}$ -1, and the range of `char` is 0 to $2^{16}$ -1. Hence, assignment between `short` and `char` always requires an explicit cast.

---



Sun Educational Services

## Promotion and Casting of Expressions

- Variables are automatically promoted to a longer form (such as `int` to `long`).
- Expression is *assignment compatible* if the variable type is at least as large (the same number of bits) as the expression type.

```
double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414;  // 12.414 is double, illegal
```

## *Expressions*

### *Promotion and Casting of Expressions*

Variables can be automatically promoted to a longer form (such as `int` to `long`), when there would be no loss of information.

```
long bigval = 6;    // 6 is an int type, OK
int smallval = 99L; // 99L is a long, illegal

double z = 12.414F; // 12.414F is float, OK
float z1 = 12.414;  // 12.414 is double, illegal
```

In general, you can think of an expression as being *assignment compatible* if the variable type is at least as large (the number of bits) as the expression type.

# Expressions

## Promotion and Casting of Expressions (Continued)

For the + operator, when the two operands are of primitive numeric types, the result is at least an `int` and has a value calculated by promoting the operands to the result type or promoting the result to the wider type of the operands. This might result in overflow or loss of precision.

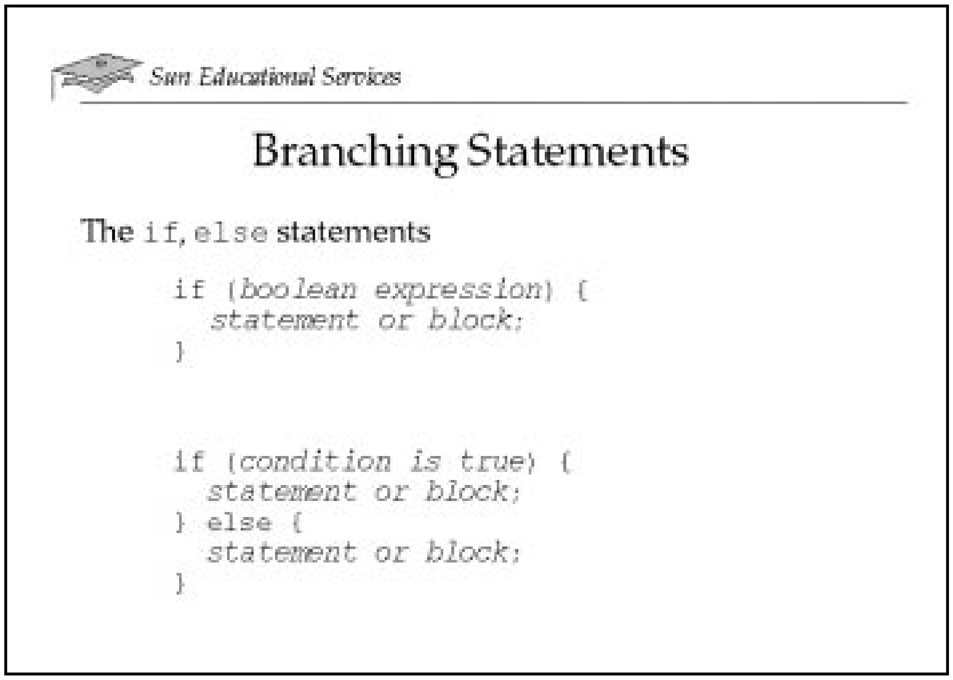
For example, the following code fragment:

```
short a, b, c;
a = 1;
b = 2;
c = a + b;
```

will cause an error because it raises each `short` to an `int` before operating on them. However, if `c` is declared as an `int`, or a typecast is done as

```
c = (short)(a + b);
```

then the code works.



Sun Educational Services

Branching Statements

The `if`, `else` statements

```
if (boolean expression) {
  statement or block;
}


if (condition is true) {
  statement or block;
} else {
  statement or block;
}
```

## *Branching Statements*

Conditional statements allow for the selective execution of portions of the program according to the value of some expressions. The Java programming language supports the `if` and `switch` statements for two-way and multiple-way branching, respectively.

### `if`, `else` *Statements*

The basic syntax for `if`, `else` statements is

```
if (boolean expression) {
    statement or block;
} else {
    statement or block;
}
```

## Branching Statements

### `if, else` *Statements (Continued)*

*Example*

```
int count;
count = getCount(); // a method defined in the program
if (count < 0) {
   System.out.println("Error: count value is negative.");
} else {
   System.out.println("There will be " + count +
                            " people for lunch today.");
}
```

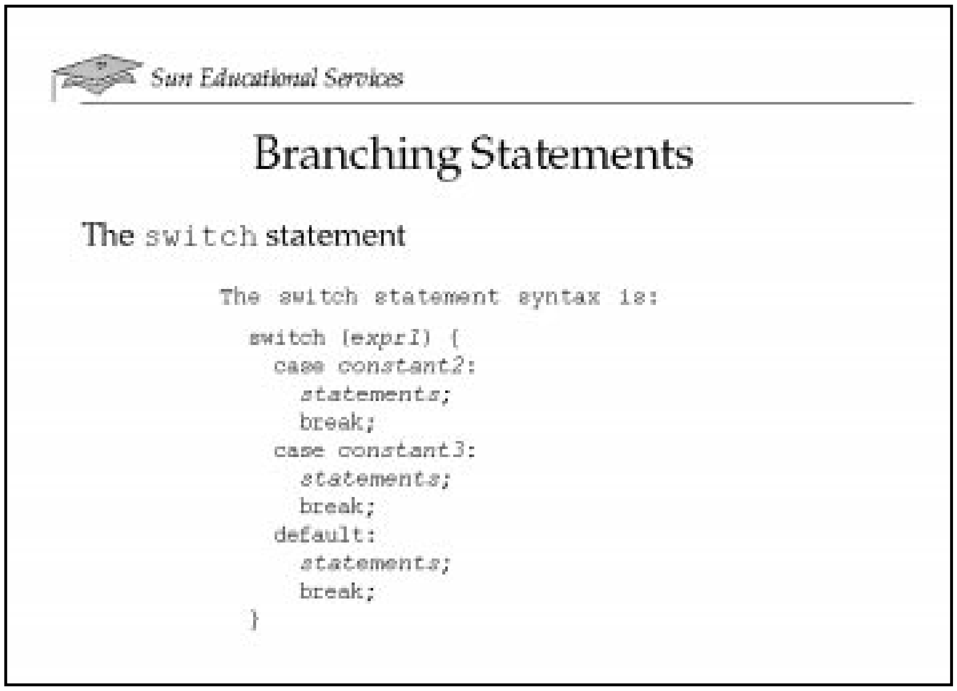
The Java programming language differs from C/C++ because an `if()` takes a `boolean` expression, not a numeric value. Recall that `boolean` types and numeric types cannot be converted or cast. Thus, if you have

```
if (x) // x is int
```

use

```
if (x != 0)
```

The entire `else` part is optional and can be omitted if no action is to be taken when the tested condition is false.



Sun Educational Services

## Branching Statements

The switch statement

The switch statement syntax is:

```
switch (expr1) {
  case constant2:
    statements;
    break;
  case constant3:
    statements;
    break;
  default:
    statements;
    break;
}
```

## *Branching Statements*

### `switch` *Statement*

The `switch` statement syntax is

```
switch (expr1) {
  case constant2:
    statements;
    break;
  case constant3:
    statements;
    break;
  default:
    statements;
    break;
}
```

## Branching Statements

### `switch` *Statement (Continued)*

---

**Note –** In the `switch (expr1)` statement, `expr1` must be assignment compatible with an `int` type. Promotion occurs with `byte`, `short`, or `char` types. Floating point, `long` expressions, or class references are not permitted.

---

The optional `default` label is used to specify the code segment to be executed when the value of the variable or expression cannot match any of the `case` values. If there is no `break` statement as the last statement in the code segment for a certain `case`, the execution will continue into the code segment for the next `case` without checking the `case` expression's value.

*Example 1*

```
int colorNum = 0;

switch (colorNum) {
   case 0:
      setBackground(Color.red);
      break;
   case 1:
      setBackground(Color.green);
      break;
   default:
      setBackground(Color.black);
      break;
}
```

Example 1 sets the background color based on the value of `colorNum`. If the first two cases are not satisfied the color will be set to black.

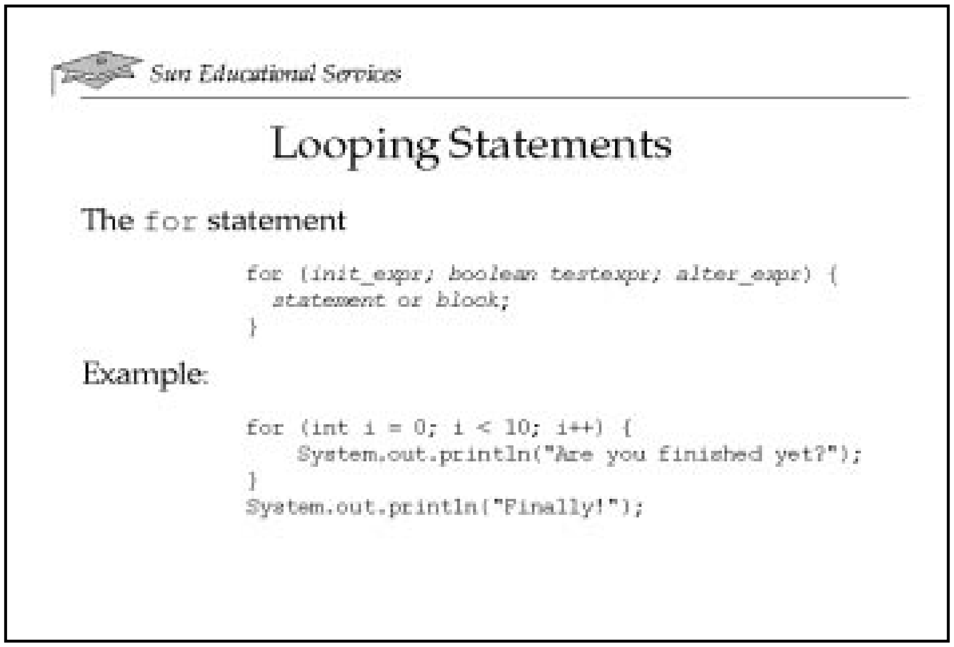
## Branching Statements

### switch Statements (Continued)

*Example 2*

```
switch (colorNum) {
   case 0:
      setBackground(Color.red);
   case 1:
      setBackground(Color.green);
   default:
      setBackground(Color.black);
      break;
}
```

Example 2 sets the background color to black irrespective of the value of the case variable `colorNum`. If `colorNum` is 0, the background color would first be set to red, then green, and then black.



Sun Educational Services

## Looping Statements

The for statement

```
for (init_expr; boolean testexpr; alter_expr) {
  statement or block;
}
```

Example:

```
for (int i = 0; i < 10; i++) {
    System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
```

## *Looping Statements*

Loop statements allow for the repeated execution of blocks of statements. The Java programming language supports three types of loop constructs: `for`, `while`, and `do` loops. `for` and `while` loops test the loop condition before executing the loop body whereas `do` loops check the loop condition after executing the loop body. This implies that the `for` and `while` loops might not execute the loop body even once, whereas `do` loops will execute the loop body at least once.

### `for` *Loops*

The `for` loop syntax is

```
for (init_expr; boolean testexpr; alter_expr3) {
    statement or block;
}
```

## Looping Statements

### `for` *Loops (Continued)*

*Example*

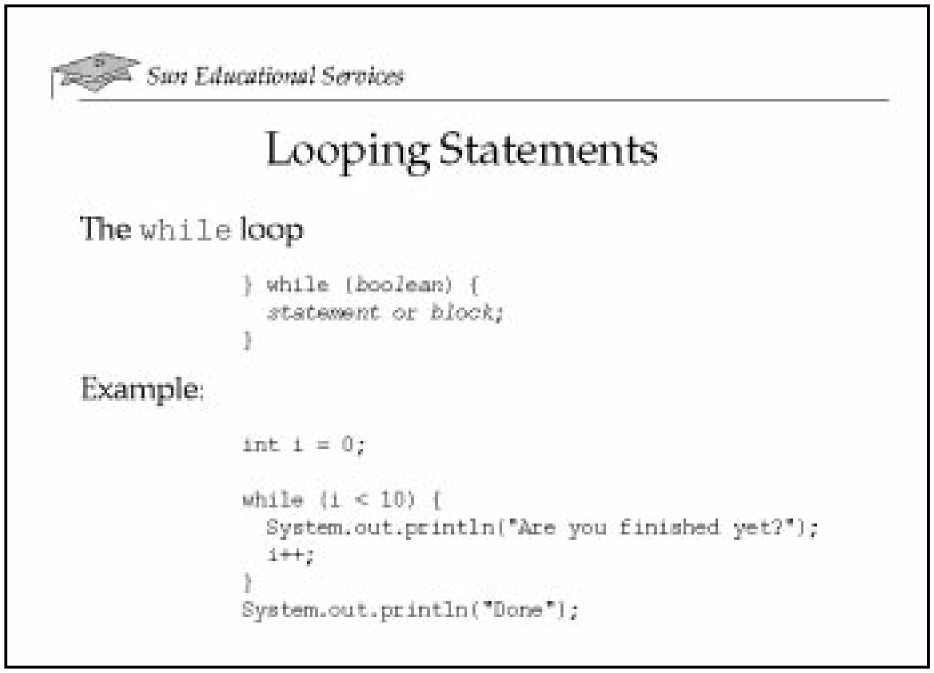
```
for (int i = 0; i < 10; i++) {
    System.out.println("Are you finished yet?");
}
System.out.println("Finally!");
```

---

**Note –** The Java programming language allows the comma separator in a `for()` loop structure. For example,
`for (i = 0, j = 0; j < 10; i++, j++) { }` is legal, and it initializes both `i` and `j` to 0, and increments both i and j after executing the loop body.

---

In the previous example, `int i` is declared and defined within the `for` block. The variable `i` is only accessible within the scope of this particular `for` block.



Sun Educational Services

## Looping Statements

The while loop

```
} while (boolean) {
  statement or block;
}
```

Example:

```
int i = 0;

while (i < 10) {
  System.out.println("Are you finished yet?");
  i++;
}
System.out.println("Done");
```

## Looping Statements

### while Loops

The while loop syntax is:

```
while (boolean) {
  statement or block;
}
```

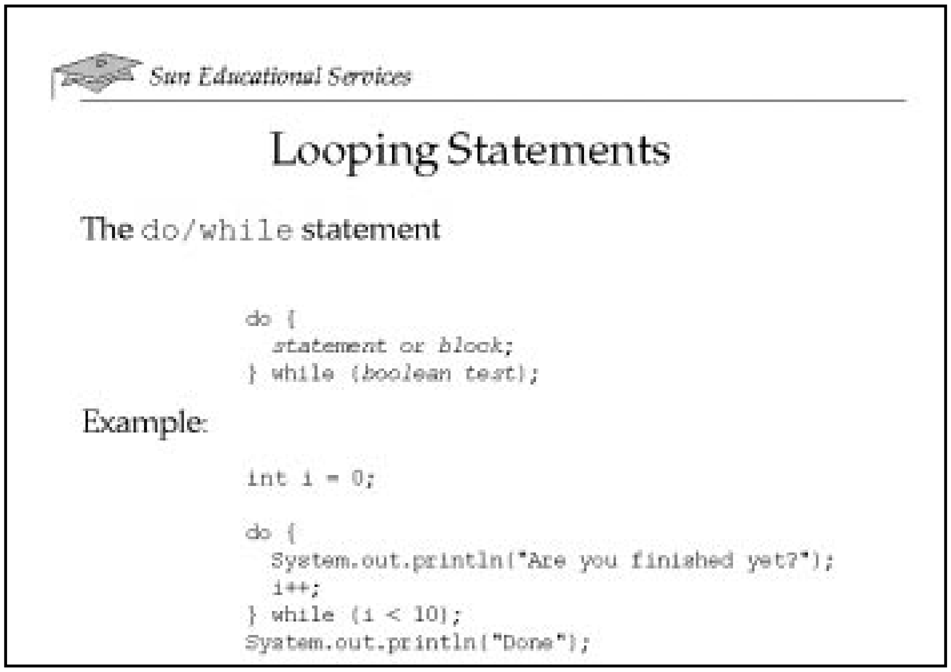
*Example*

```
int i = 0;

while (i < 10) {
   System.out.println("Are you finished yet?");
   i++;
}
System.out.println("Done");
```

## Looping Statements

### `while` *Loops (Continued)*

Ensure that the loop control variable is appropriately initialized before the loop body begins execution, and ensure that the loop condition is true to begin with. The control variable must be updated appropriately to prevent an infinite loop.



Sun Educational Services

## Looping Statements

The do/while statement

```
do {
  statement or block;
} while (boolean test);
```

Example:

```
int i = 0;

do {
  System.out.println("Are you finished yet?");
  i++;
} while (i < 10);
System.out.println("Done");
```

## *Looping Statements*

### do *Loops*

The syntax for the do loop is

```
do {
   statement or block;
} while (boolean test);
```

*Example*

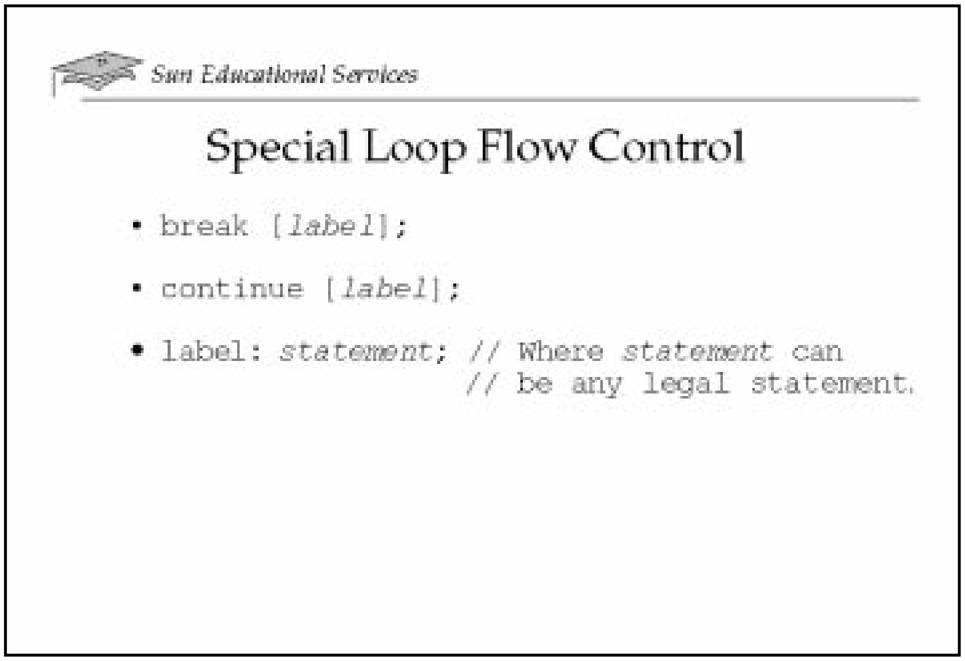
```
int i = 0;
do {
   System.out.println("Are you finished yet?");
   i++;
} while (i < 10);
System.out.println("Done");
```

As with the `while` loops, ensure that the loop control variable is appropriately initialized, updated in the body of the loop, and properly tested.

## Looping Statements

### do *Loops*

Use the for loop in cases where the loop is to be executed a predetermined number of times. Use the while and do loops in cases where this is not determined beforehand.



Sun Educational Services

## Special Loop Flow Control

- break [*label*];
- continue [*label*];
- label: *statement*; // Where *statement* can
  // be any legal statement.

## *Special Loop Flow Control*

The following statements can be used to further control loop statements:

- `break` [*label*]`;`
- `continue` [*label*]`;`
- `label:` *statement*`;// Where` *statement* `should be a loop`

The `break` statement is used to exit from `switch` statements, loop statements, and labeled blocks prematurely.

The `continue` statement is used to skip over and jump to the end of the loop body.

`label` identifies any valid statement to which control needs to be transferred. It is used to identify a compound statement that is a loop construct.

## *Special Loop Flow Control*

The break, continue, and label statements can be used follows:

- The break statement

```
do {
  statement or block;
  if (condition is true)
    break;
} while (boolean expression);
```

- The continue statement

```
do {
  statement or block;
  if (boolean expression)
    continue;
} while (boolean expression);
```

- The break statement with a label named loop

```
loop:
  do {
    statement;
    do {
      statement;
      statement;
      if (boolean expression)
        break loop;
    } while (boolean expression)
      statement;
  } while (boolean expression);
```

## Special Loop Flow Control

- The `continue` statement with a label named `test`

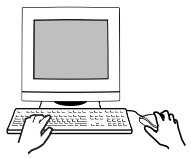
```
test:
  do {
     statement;
    do {
       statement;
       statement;
       if (condition is true)
          continue test;
    } while (condition is true)
       statement;
  } while (condition is true);
```

✓ ***Labeled*** `break` ***and*** `continue` ***statements can be used to jump directly out of nested loops. This facility removes one of the legitimate reasons for using*** `goto`***. The Java programming language does not use*** `goto`***, although it is a reserved word.***

### Example

```
1  loop:while ( true ) {
2    for ( int i = 0; i < 100; i++ ) {
3      switch ( c = System.in.read() ) {
4        case -1:
5        case '\n':
6          // jumps out of while loop to line 13
7          break loop;
8        ...
9      }
10   }
11 }
12
13 test:for ( ... ) {
14   ...
15   while ( ... ) {
16     if ( j > 10 ) {
17       // jumps to the increment portion of for loop
18       // at line 13
19       continue test;
20     }
21   }
22 }
```

## Exercise: Using Expressions



**Exercise objective –** You will write, compile, and run two arithmetic programs that uses identifiers, expressions, and control structures.

### Preparation

In order to successfully complete this lab, you must be able to compile and run a Java program, and have a familiarity with flow control constructs.

### Tasks

#### *Level 1 Lab: Create a Factorial Application*

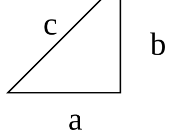
A factorial of a number X (often written as X!) is equal to X*(X-1)*(X-2)*....*1. For example, 4! is equal to 4 x 3 x 2 x 1 = 24.

Create an application called `Factor` that will print the factorials of 2, 4, 6, and 10.

#### *Level 2 Lab: Solve a Geometry Program*

For any given right triangle, the length of the hypotenuse (the longest side) is given by the formula:

$$c = \sqrt{a^2 + b^2}$$



Write a Java software program called `Hypotenuse` that calculates the longest side given the other two sides of a right triangle.

## Exercise: Using Expressions

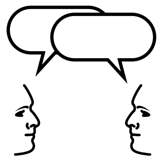
### Tasks

#### Level 1 Lab: Create a Factorial Application (Continued)

Hint – Start with the template solution provided in the `mod03/templates` directory which take input from the command line. Also, look at the `java.lang.Math` class, specifically `Math.sqrt()`;

## Exercise: Using Expressions

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Distinguish between instance and local variables
- ❑ Describe how instance variables are initialized
- ❑ Identify and correct a `Possible reference before assignment` compiler error
- ❑ Recognize, describe, and use Java software operators
- ❑ Distinguish between legal and illegal assignments of primitive types
- ❑ Identify `boolean` expressions and their requirements in control constructs
- ❑ Recognize assignment compatibility and required casts in fundamental types
- ❑ Use `if`, `switch`, `for`, `while`, and `do` constructions and the labeled forms of `break` and `continue` as flow control structures in a program

## Notes

## Think Beyond

What data types do most programming languages use to group similar data elements together?

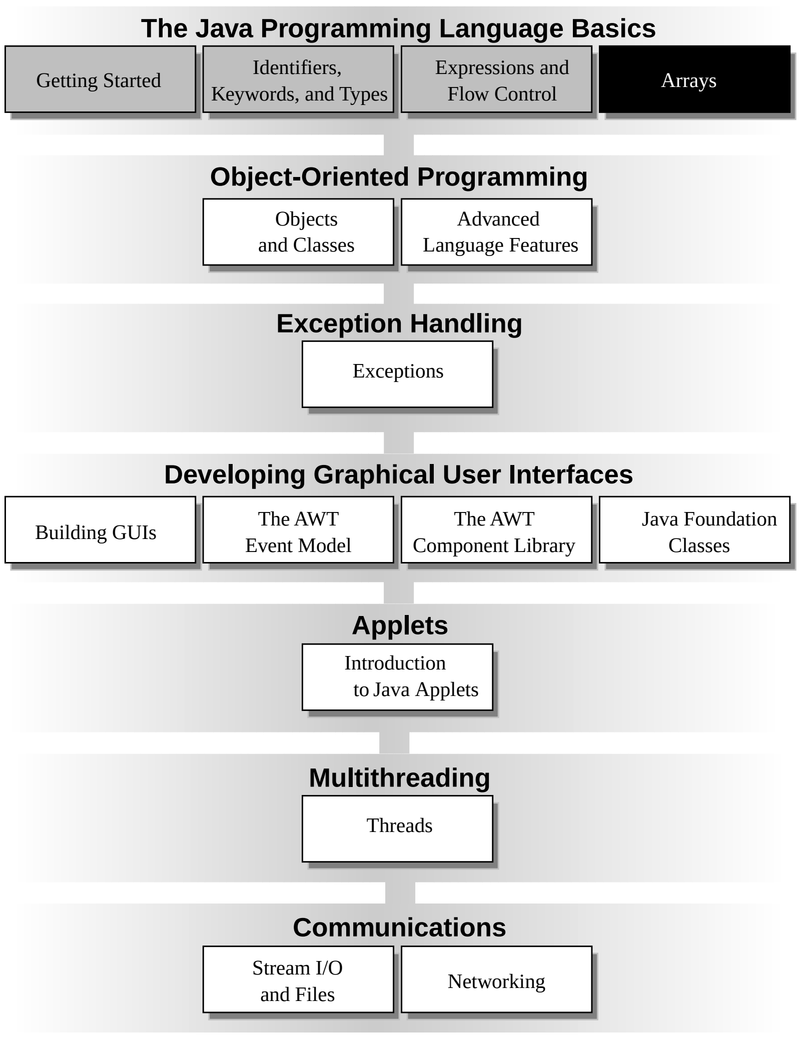
How do you perform the same operation on all elements of a group (for example, a matrix)?

What data types does the Java programming language use?

# Arrays 4

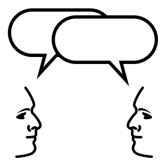
## Course Map

This module describes how to define, initialize, and use arrays in the Java programming language.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

- What is the purpose of an array?

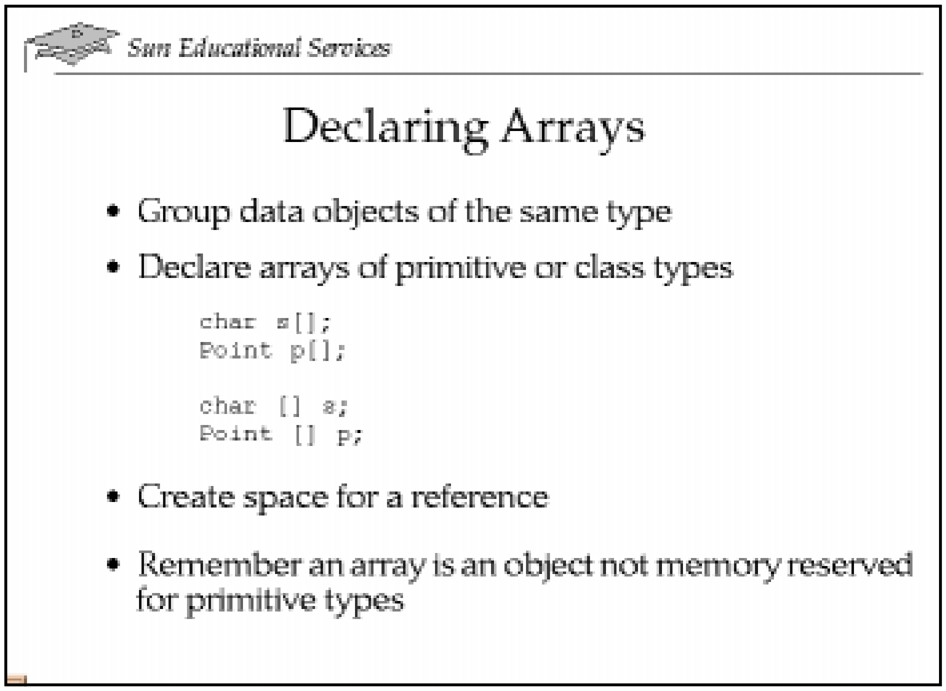
✓ ***An array is often used to group similar data elements together, and perform the same operation on all elements of the group (for example, a matrix). This module discusses the Java programming language implementation of arrays, or groupings of similar elements.***

✓ ***This module covers declaration and initialization of arrays, as well as their use. Implementation of a multi-dimensional array as an array of arrays is examined. Use of arrays' built-in*** `length` ***attribute for bounds checking is also discussed.***

## Objectives

Upon completion of this module, you should be able to

- Declare and create arrays of primitive, class, or array types
- Explain why elements of an array are initialized
- Given an array definition, initialize the elements of an array
- Determine the number of elements in an array
- Create a multi-dimensional array
- Write code to copy array values from one array type to another



Sun Educational Services

## Declaring Arrays

- Group data objects of the same type
- Declare arrays of primitive or class types

```
char s[];
Point p[];

char [] s;
Point [] p;
```

- Create space for a reference
- Remember an array is an object not memory reserved for primitive types

## *Declaring Arrays*

Arrays are typically used to group objects of the same type and refer to them by a common name.

You can declare arrays of any type, either primitive or class:

```
char s[];
Point p[]; // where Point is a class
```

In the Java programming language, an array is an object even when the array is made up of primitive types, and as with other class types, the declaration does not create the object itself. Instead, the declaration of an array creates a reference that *can be used* to refer to an array. The actual memory used by the array elements is dynamically allocated either by a `new` statement or by an array initializer.

In the next section, you will see how to create and initialize the actual array.

## Declaring Arrays

The code shown on page 4-4, with square brackets after the variable name, is standard for C, C++, and the Java programming language. This format leads to complex forms of declaration that can be difficult to read. Therefore, the Java programming language allows an alternative form with the square brackets on the left:

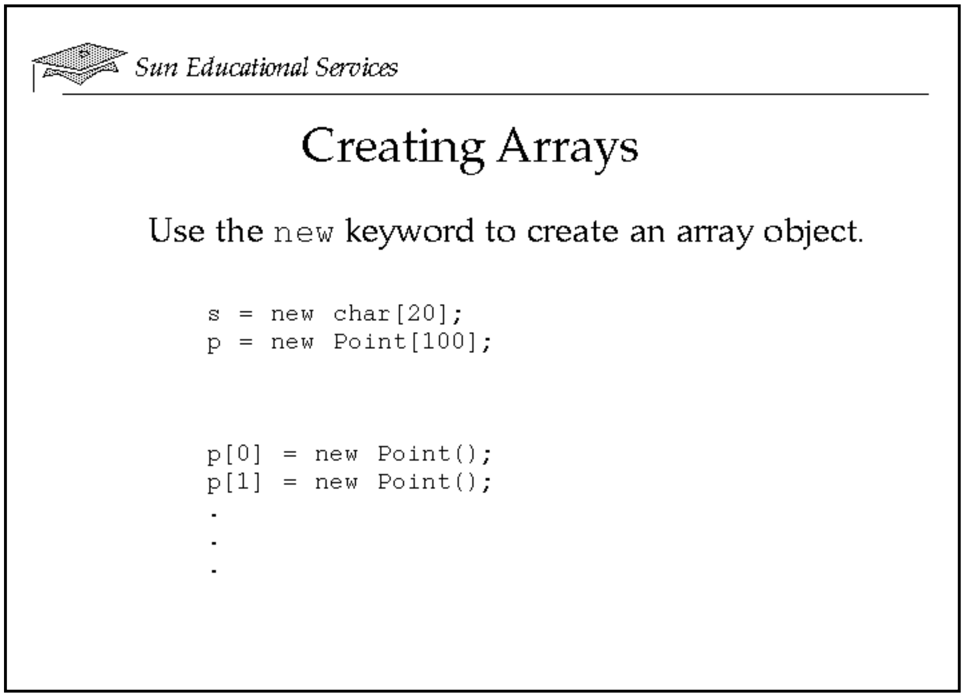
```
char [] s;
Point [] p;
```

The result is that you can consider a declaration as having the type part at the left, and the variable name at the right. You will see both formats used, but should decide on one or the other for your own use. The declarations do not specify the actual size of the array.

---

**Note –** When declaring arrays with the brackets to the left, the brackets apply to all variables to the right of the brackets.

---



Sun Educational Services

## Creating Arrays

Use the `new` keyword to create an array object.

```
s = new char[20];
p = new Point[100];


p[0] = new Point();
p[1] = new Point();
.
.
.
```

## *Creating Arrays*

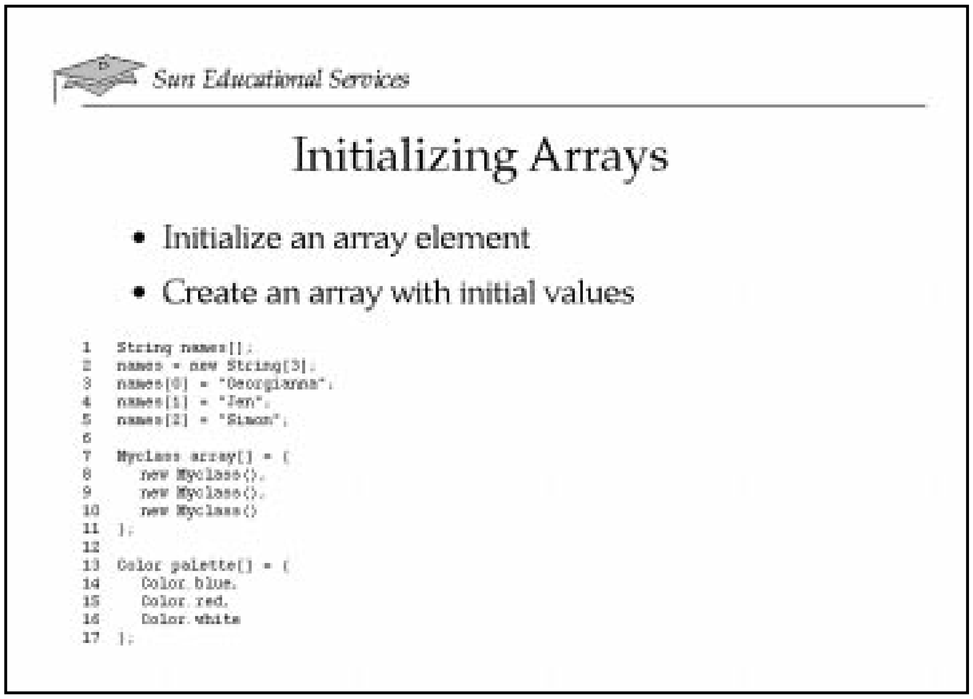
You can create arrays, like all objects, using the `new` keyword

```
s = new char[20];
p = new Point[100];
```

The first line creates an array of 20 `char` values. The second line creates an array of 100 variables of type `Point`. It does not, however, create 100 `Point` objects. These must be created separately as follows:

```
p[0] = new Point();
p[1] = new Point();
.
.
.
```

The subscript used to index the individual array elements will always begin from 0, and will have to be maintained in the legal range – greater than or equal to zero and less than the array length. Any attempt made to access an array element outside these bounds will cause a runtime error. There are more elegant ways to initialize arrays, which will be introduced shortly.



Sun Educational Services

## Initializing Arrays

- Initialize an array element
- Create an array with initial values

```
String names[];
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";

Myclass array[] = {
   new Myclass(),
   new Myclass(),
   new Myclass()
};

Color palette[] = {
   Color.blue,
   Color.red,
   Color.white
};
```

## *Initializing Arrays*

When you create an array, every element is initialized. In the case of the `char` array `s` in the previous section, each value is initialized to the zero (`'\u0000'` - null) character. In the case of the array `p`, each value is initialized to `null`, indicating that it does not (yet) refer to a `Point` object. After the assignment `p[0] = new Point()`, the first element of the array refers to a real `Point` object.

---

**Note –** Initialization of all variables, including elements of arrays, is essential to the security of the system. Variables must *not* be used in an uninitialized state.

---

The Java programming language allows a shorthand that will create arrays with initial values

```
String names[] = {
   "Georgianna",
   "Jen",
   "Simon"
};
```

## *Initializing Arrays*

✓ ***Make sure students understand that in this example the* `names` *array will have exactly three elements.***

This code is equivalent to

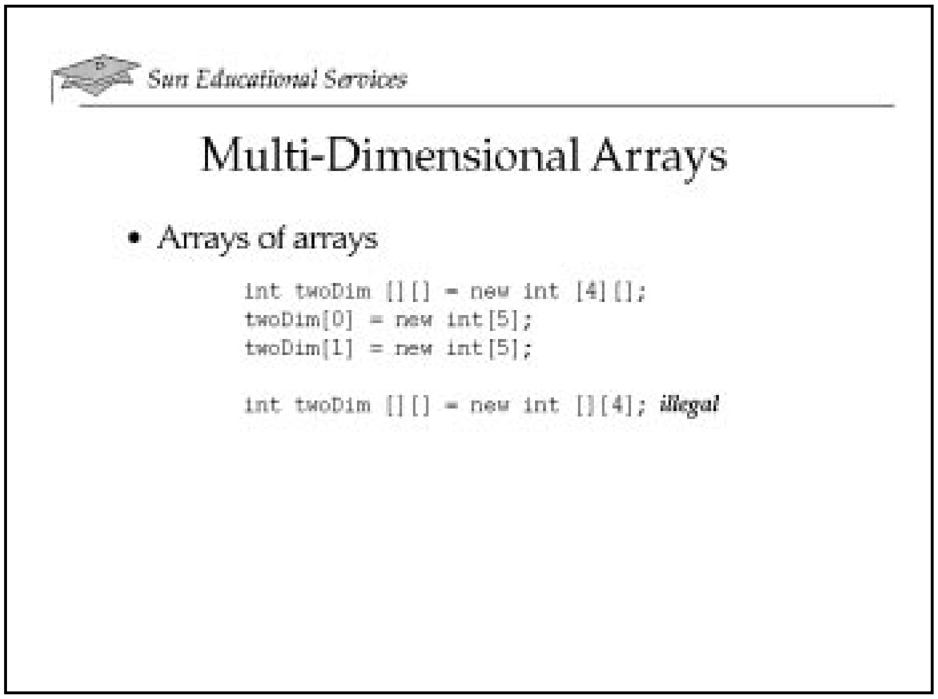
```
String names[];
names = new String[3];
names[0] = "Georgianna";
names[1] = "Jen";
names[2] = "Simon";
```

This shorthand can be used for any element type. For example:

```
Myclass array[] = {
   new Myclass(),
   new Myclass(),
   new Myclass()
};
```

Constant values of the appropriate class type can also be used

```
Color palette[] = {
   Color.blue,
   Color.red,
   Color.white
};
```


Sun Educational Services

## Multi-Dimensional Arrays

- Arrays of arrays

```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];

int twoDim [][] = new int [][4]; illegal
```

## *Multi-Dimensional Arrays*

The Java programming language does not provide multi-dimensional arrays in the same way that other languages do. Since an array can be declared to have any base type, you can create arrays of arrays (and arrays of arrays of arrays, and so on). A two-dimensional array is shown in the following example:

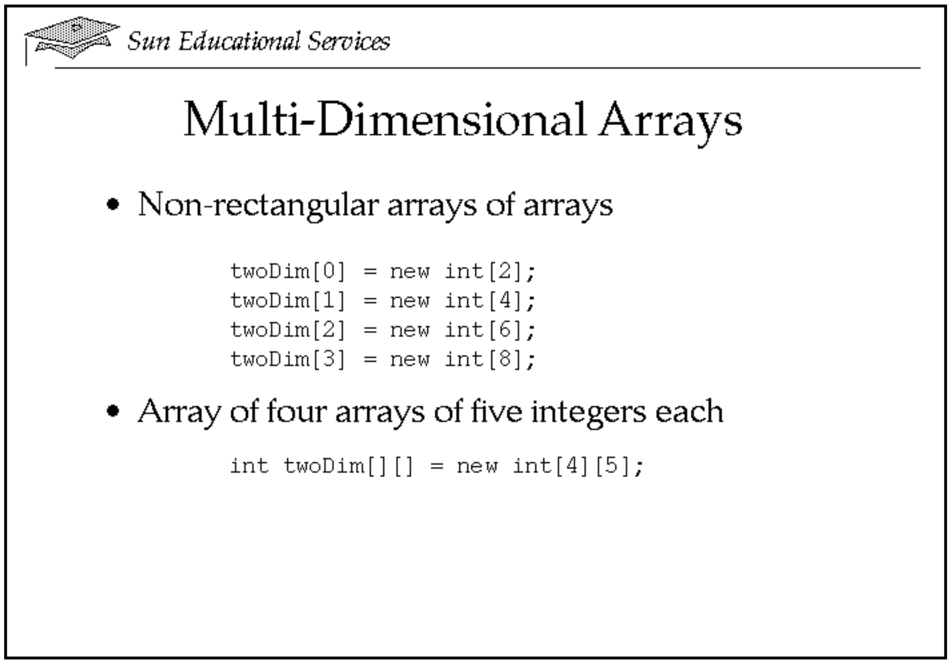
```
int twoDim [][] = new int [4][];
twoDim[0] = new int[5];
twoDim[1] = new int[5];
```

The object that is created by the first call to `new` is an array which contains four elements. Each element is a `null` reference to an element of type `array of int` and must be initialized separately to each point to its array.

---

**Note –** Although the declaration format allows the square brackets to be at the left or right of the variable name, this flexibility does not carry over to other aspects of array syntax. For example, `new int [][4]` is not legal.

---



Sun Educational Services

## Multi-Dimensional Arrays

- Non-rectangular arrays of arrays

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

- Array of four arrays of five integers each

```
int twoDim[][] = new int[4][5];
```

## *Multi-Dimensional Arrays*

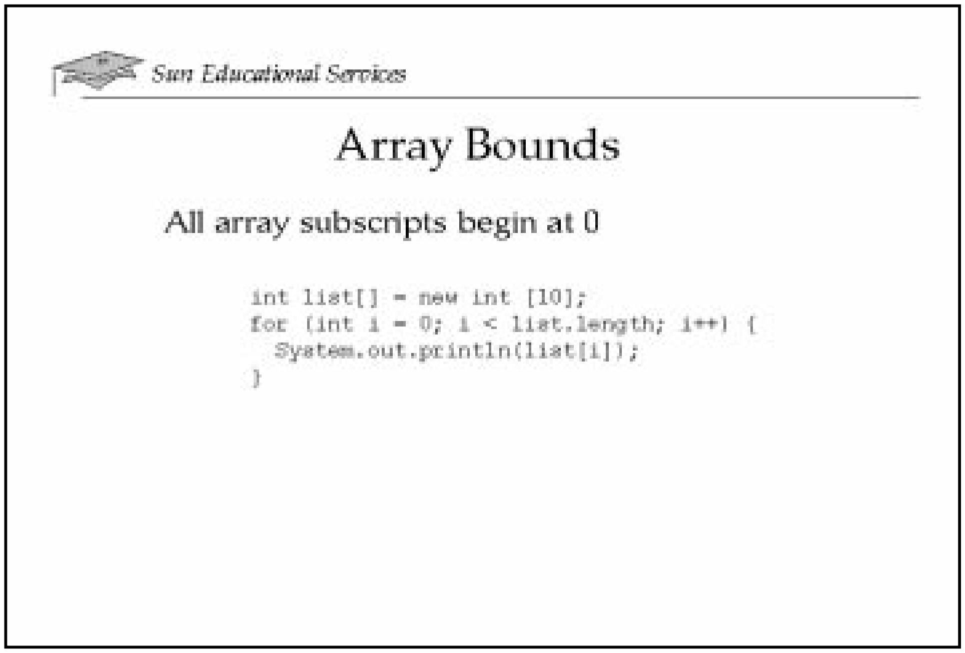
Because of this separation, it is possible to create non-rectangular arrays of arrays. That is, the elements of `twoDim` can be initialized like this:

```
twoDim[0] = new int[2];
twoDim[1] = new int[4];
twoDim[2] = new int[6];
twoDim[3] = new int[8];
```

Since this type of initialization is tedious, and the rectangular array of arrays is the most common form, there is a shorthand to create two-dimensional arrays. For example,

```
int twoDim[][] = new int[4][5];
```

can be used to create an array of four arrays of five integers each.



Sun Educational Services

## Array Bounds

All array subscripts begin at 0

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++) {
  System.out.println(list[i]);
}
```

## *Array Bounds*

In the Java programming language, all array subscripts begin at zero. The number of elements in an array is stored as part of the array object, as the `length` attribute. This value is used to perform bounds checking of all runtime accesses. If an out-of-bounds access occurs, then a runtime error occurs.

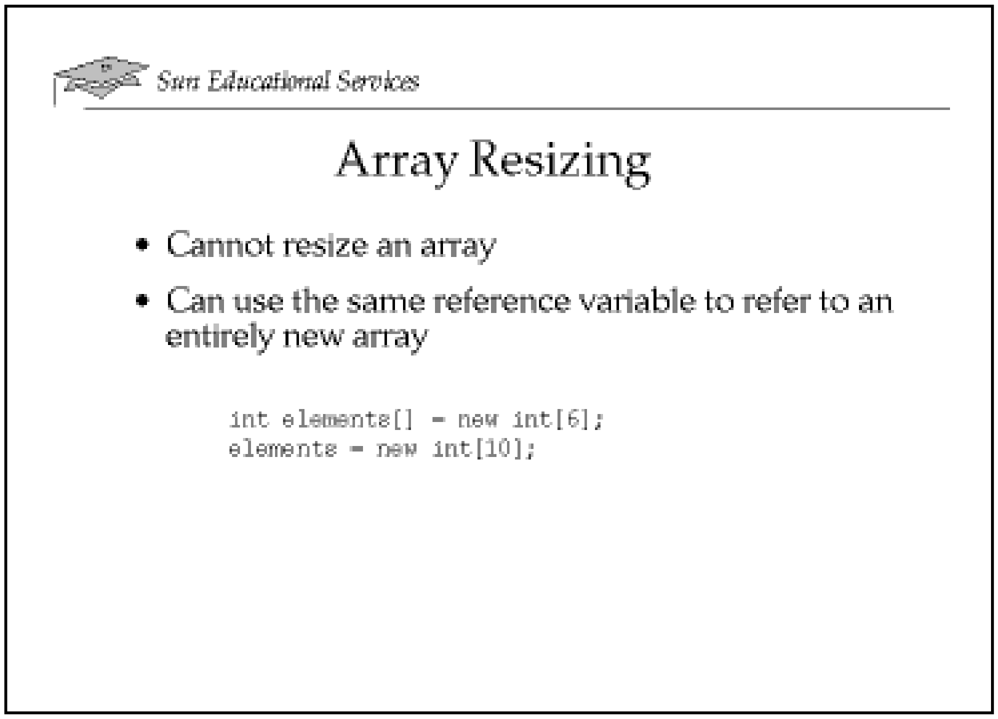
Use the `length` attribute to iterate on an array as follows:

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++) {
   System.out.println(list[i]);
}
```

Using the `length` attribute makes program maintenance easier.

✓ ***Ask the students about the multi-dimensional example***

```
int multiDim[][] = new int [10][5];

System.out.println(“multiDim.length is “ + multiDim.length);

System.out.println(“multiDim[0].length is “ + multiDim[0].length);
```


Sun Educational Services

## Array Resizing

- Cannot resize an array
- Can use the same reference variable to refer to an entirely new array

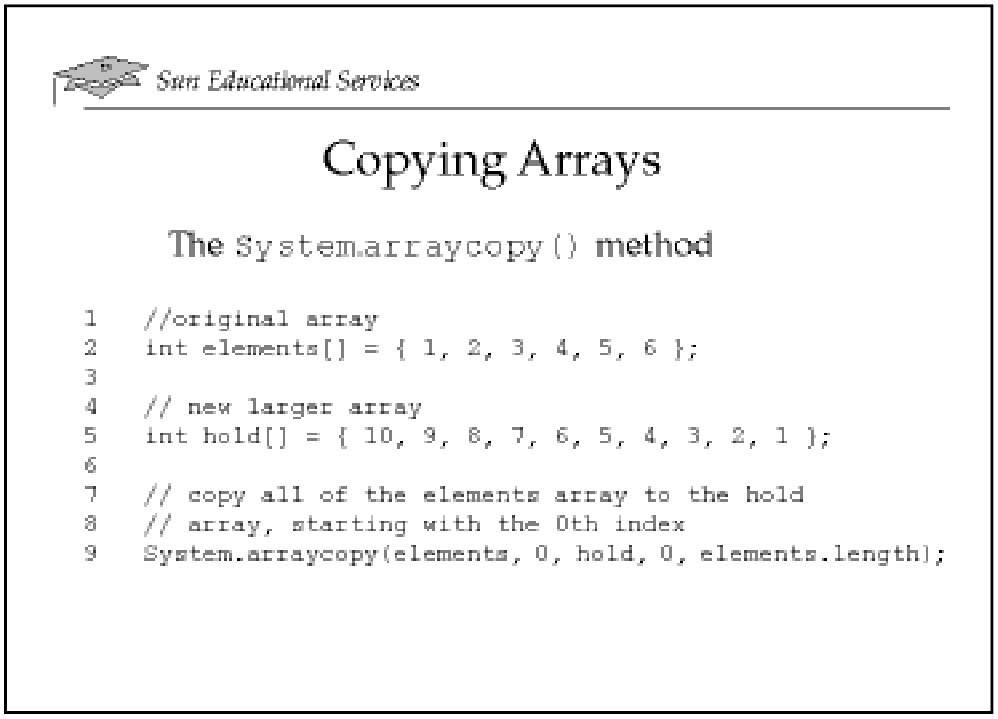
```
int elements[] = new int[6];
elements = new int[10];
```

## *Array Resizing*

Once created, an array cannot be resized. However, you can use the same reference variable to refer to an entirely new array

```
int myArray[] = new int[6];
myArray = new int[10];
```

In such a case, the first array is effectively lost unless some other reference to it is retained elsewhere.


Sun Educational Services

## Copying Arrays

The System.arraycopy() method

```
//original array
int elements[] = { 1, 2, 3, 4, 5, 6 };

// new larger array
int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copy all of the elements array to the hold
// array, starting with the 0th index
System.arraycopy(elements, 0, hold, 0, elements.length);
```

## *Copying Arrays*

The Java programming language provides a special method in the `System` class, `arraycopy()`, to copy arrays. For example, `arraycopy()`can be used as follows:

```
// original array
int myArray[] = { 1, 2, 3, 4, 5, 6 };

// new larger array
int hold[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

// copy all of the myArray array to the hold
// array, starting with the 0th index
System.arraycopy(myArray, 0, hold, 0,
                  myArray.length);
```

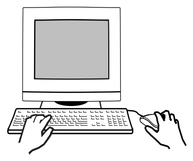
At this point, the array `hold` has the following contents: 1, 2, 3, 4, 5, 6, 4, 3, 2, 1.

---

**Note –** `System.arraycopy()` will copy references, not objects, when dealing with arrays of objects. The objects themselves do not change.

---

## Exercise: Using Arrays



**Exercise objective –** After defining and initializing arrays, you will use them in a program.

### Preparation

In order to successful complete this lab, you must understand basic matrix concepts and know how to index an array to obtain its value.

### Tasks

#### Level 1 Lab: Create a Basic Array

Complete the following steps:

1. Create a class called `BasicArray`. In the `main()` method, declare two variables called `thisArray` and `thatArray`. They should be of type `array of int`.
2. Create an array of 10 `int` values that range from 1 to 10. Assign the reference of this third array to the variable *`thisArray`*.
3. Use a `for()` loop to print out all values of *`thisArray`*. How do you control the limit on the loop?
4. Compile and run the program. How many values are printed? What are the values?
5. For each element of *`thisArray`*, set the value to be the factorial of the index value. Print out the values of the array.
6. Compile and run the program.
7. Assign the reference of *`thisArray`* to the variable *`thatArray`*. Print out all the elements of *`thatArray`*.

## Exercise: Using Arrays

### Tasks

#### Level 1 Lab: Basic Array Use (Continued)

8. Compile and run the program. How many values are displayed for `thatArray`? What are these values and where do they come from?

9. Modify some of the elements of `thisArray`. Print out the values of `thatArray`.

10. Compile and run the program. What do you notice about the values of `thatArray`?

11. Create an array of 20 `int` values. Assign the reference of the new array to the variable `thatArray`. Print out the values of `thatArray`.

12. Compile and run the program. How many values are shown for each of the arrays? What are those values?

13. Copy the values of `thisArray` into `thatArray`. What method call will you use to do this? How will you limit the number of elements copied? What will happen to elements 10 to 19 of `thatArray`?

14. Print out the values of `thatArray`.

15. Compile and run the program. Were you right about the values shown? If not, do you understand what has happened and what you misunderstood?

16. Change some values of `thatArray`. Print out both `thisArray` and `thatArray`.

17. Compile and run the program. Are the values as you expected?

## *Exercise: Using Arrays*

### *Tasks*

#### *Level 2 Lab: CreateArrays of Arrays*

Complete the following steps:

1. Create a class called `Array2D`. In the `main()` method declare a variable called *`twoD`*, of type `array` of `array` of `int`. Make the first dimension equal to 4 (`[4][]`).

2. Create an array of element type `int`. The array should have four elements and be assigned to element [0] of the variable, *`twoD`*.

3. Write two nested `for()` loops that print out all the values of *`twoD`*. Arrange the output in a matrix format. (The `System.out.print()` method is helpful here.)

4. Compile and run the program. You should find that it generates a runtime error (called a null pointer exception) since the elements [1] to [3] of *`twoD`* are uninitialized.

5. Create further arrays of `ints` containing five, six, and seven elements, respectively. Assign the references to these to the elements [1], [2], and [3] of *`twoD,`* respectively. Ensure that the code for this is inserted *before* the nested `for()` loops described in step 3.

6. Compile and run the program. This time you should see a non-rectangular layout of zero values.

7. Assign to each element of the *`twoD`* array a distinct nonzero value. (Hint – Use `Math.random()` to obtain random values.)

8. Declare a variable called *`oneD`* of type `array` of `int`. Then, create an array of `int` that will hold four elements, and assign the reference to this array to the first element of array *`twoD`* and array *`oneD`*. Print out both the *`oneD`* and *`twoD`* arrays, after making the assignment.

9. Compile and run the program. Observe that the individual arrays shown by printing the values in *`oneD`* are the same as the elements of the array in *`twoD`*.

## *Exercise: Using Arrays*

### *Tasks*

#### *Level 3 Lab: Develop a Anagram Game*

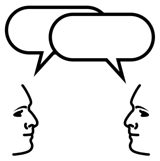
Complete the following steps:

1. Create an application called `WordScrambler` that has an array of words (up to eight characters in length) that it will scramble (rearrange the order) of the letters and present to the user.

2. Allow the user to see the word anagrams and take up to five guesses to unscramble each.

## Exercise: Using Arrays

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Declare and create arrays of primitive, class, or array types
- ❑ Explain why elements of an array are initialized
- ❑ Explain how to initialize the elements of an array
- ❑ Determine the number of elements in any array
- ❑ Create a multi-dimensional array
- ❑ Write code to copy array values from one array type to another

## Think Beyond

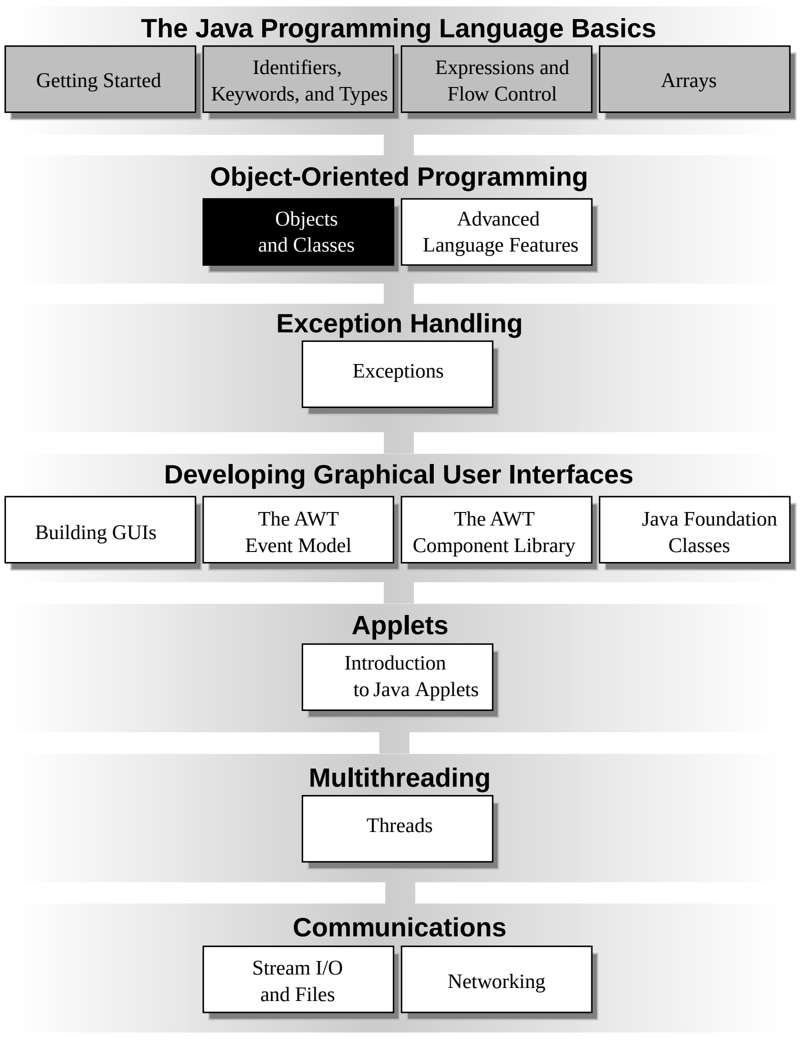
How can you create a three-dimensional array?

What is a disadvantage of using arrays?

# *Objects and Classes* 5

## *Course Map*

This module is the first of two discussing the object-oriented paradigm and the object-oriented features of the Java programming language.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following questions are relevant to the material presented in this module:

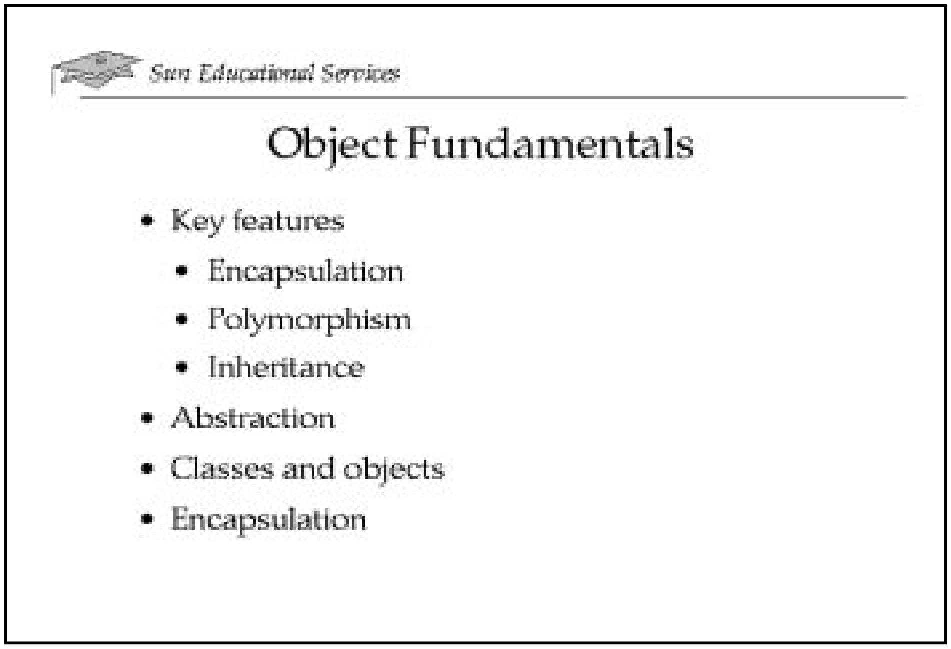
- The elements of the Java programming language studied up to this point exist in most languages whether object-oriented or not.
  - ▼ What features does the Java programming language possess to make it an object-oriented language?
  - ▼ What does the term *object-oriented* really mean?

✓ ***This module is the first of two which discusses the object-oriented features of the language. The object-oriented features help make development and maintenance of a Java software program an easier task than if the programs were written in another language. For example, data elements in the form of objects are easier to conceptualize as they model real-world objects better; this helps to cut a project's development time. Other object-oriented ideas such as data hiding help make future efforts such as maintenance and upgrading a more efficient and reliable procedure.***

## Objectives

Upon completion of this module, you should be able to

- Define *encapsulation*, *polymorphism*, and *inheritance*
- Use the access modifiers `private` and `public`
- Develop a program segment to create and initialize an object
- Invoke a method on a particular object
- Describe constructor and method overloading
- Describe what the `this` reference is used for
- Discuss why Java application code is reusable
- In a Java program, identify the following:
    - The `package` statement
    - The `import` statement
    - Classes, member methods, and variables
    - Constructors
    - Overloaded methods
    - Overridden methods
    - Parent class constructors



Sun Educational Services

### Object Fundamentals

- Key features
  - Encapsulation
  - Polymorphism
  - Inheritance
- Abstraction
- Classes and objects
- Encapsulation

## Object Fundamentals

The object-oriented programming (OOP) paradigm enables real-world concepts to be modeled in a computer program. It includes the features of structured programming and the mechanism to organize data and algorithms. There are three key features of OOP languages: encapsulation, polymorphism, and inheritance. All these features are tied to the concept of a *class*.

### Abstract Data Type

When data types are composed of data items, you can define a number of program pieces or methods that operate specifically on that type of data. When a programming language defines a primitive type such as an integer, it also defines a number of operations (such as add, subtract, multiply, and divide) that it can perform on instances of that type.

## Object Fundamentals

### Abstract Data Type (Continued)

In many programming languages, once an aggregate data type has been defined, the programmer defines utility functions to operate on variables of that type without any particular association between the code and the aggregate type (except perhaps in the naming convention).

Some programming languages, including Java, allow a tighter association between the declaration of a data type and the declaration of the code that is intended to operate on variables of that type. Such an association is often described as an *abstract data type*.

### Classes and Objects

The notion of an abstract data type in the Java programming language is realized as a *class*. A class provides a definition for a particular type of object. It defines the data inside an object, the specifics of that object's creation, and the functionality provided by that object to act on its own data. A class is therefore a template. *Objects* are constructed based on their class model, much like a building is constructed from an architectural drawing. Many buildings can be built from the same blueprint, yet each building is an object on its own.

It should be noted that the class defines what an object is, but is not itself an object, per se. There is only one copy of a class definition in a program, but there can be several objects that are instances of that class. To instantiate an object in the Java programming language, use the `new` operator.

Data types defined in classes are not of much use unless you can do things with them. Methods define the operations that can be carried out on objects; in short, methods define what the class does. Thus all methods in the Java programming language must belong to a class. Unlike C++ programs, Java software programs cannot have a method outside a class in global space.

# Object Fundamentals

## Classes and Objects (Continued)

Consider the following example of a class:

```
class EmpInfo {
  String name;
  String designation;
  String department;
}
```

These variables (`name`, `designation`, and `department`) are called the members of the class `EmpInfo`.

To instantiate an object, create it and then assign values to the members as follows:

```
// Create instance
EmpInfo employee = new EmpInfo();

// Initializes the three members
employee.name = "Robert Javaman";
employee.designation = "Manager";
employee.department = "Coffee Shop";
```

The object `employee` of class `EmpInfo` can now be used. For example,

```
System.out.println(employee.name + " is " +
                   employee.designation + " at " +
                   employee.department);
```

prints the following:

```
Robert Javaman is Manager at Coffee Shop
```

# Object Fundamentals

## Classes and Objects (Continued)

You can now define a method `print()` embedded within the class to print the data shown in the following example. This is a fundamental feature of object- oriented languages where data and code are *encapsulated* into a single entity. The piece of code named `print()` can be referred to as a method, which is the object-oriented term for a "function."

```
class EmpInfo {
  String name;
  String designation;
  String department;

  void print() {
    System.out.println(name + " is " +
       designation + " at " + department);
  }
}
```

This method prints the data of the class members once the object has been created and instantiated. This is done as follows:

```
// Creates instance
EmpInfo employee = new EmpInfo();

// Initializes the three members
employee.name = "Robert Javaman";
employee.designation = "Manager";
employee.department = "Coffee Shop";

// Prints the details
employee.print();
```

## Object Fundamentals

### Classes and Objects (Continued)

Consider an aggregate data type `MyDate` and a function called `tomorrow()` that evaluates the next date.

You can create an association between the `MyDate` type and the `tomorrow()` method as follows:

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    // code to increment day
  }
}
```

A method does not operate on a piece of data; rather, the data operates on itself.

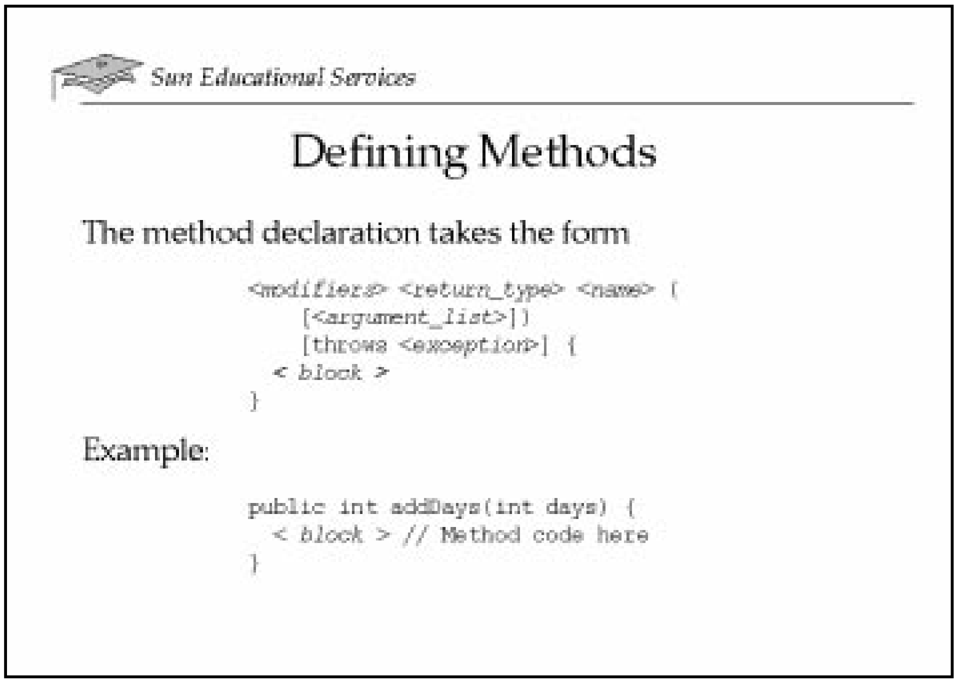
```
MyDate d = new MyDate();
d.tomorrow();
```

This notation reflects the idea that the behavior is carried out by the object rather than on the object. Recall that you can refer to the fields of the `MyDate` class by using the dot notation

```
d.day
```

and that this meant “the `day` field of the `MyDate` object referred to by the variable `d`.” Thus, the previous example reads “the `tomorrow` behavior of the `MyDate` object referred to by the variable `d`.” In other words, the `d` object performs the `tomorrow()` operation on itself.

The idea that methods are a property of an object and can interact intimately with their object’s data as part of a single unit is a key object-oriented concept. (This differs from the idea that methods are separate entities, brought in from the outside, to act on the data.) The term *message passing* is often used to convey the notion of instructing an object to do something to its own data; an object’s methods define what that object can do to its own data.


Sun Educational Services

## Defining Methods

The method declaration takes the form

```
<modifiers> <return_type> <name> (
    [<argument_list>])
    [throws <exception>] {
  < block >
}
```

Example:

```
public int addDays(int days) {
  < block > // Method code here
}
```

## *Object Fundamentals*

### *Defining Methods*

The Java programming language uses an approach that is very similar to other languages, particularly C and C++, to define methods. The declaration takes the following form:

```
<modifiers> <return_type> <name> ([<argument_list>])
    [throws <exception>] { .... }
```

The `<name>` can be any legal identifier, with some restrictions based on the names that are already in use.

The `<return_type>` indicates the type of value returned by the method. If the method does not return any value, it should be declared void. Java technology is rigorous about returned values, and if the declaration states that the method returns an `int`, for example, then the method must return an `int` from all possible return paths (and can only be invoked in contexts that expect an `int` to be returned). Use the `return` command within a method to pass back a value.

# Object Fundamentals

## Defining Methods

The <*modifiers*> segment can carry a number of different modifiers, including `public`, `protected`, and `private`. The `public` access modifier indicates that the method can be called from any other code. `private` indicates that a method can be called only by the other methods in the class. `protected` will be discussed later in this course.

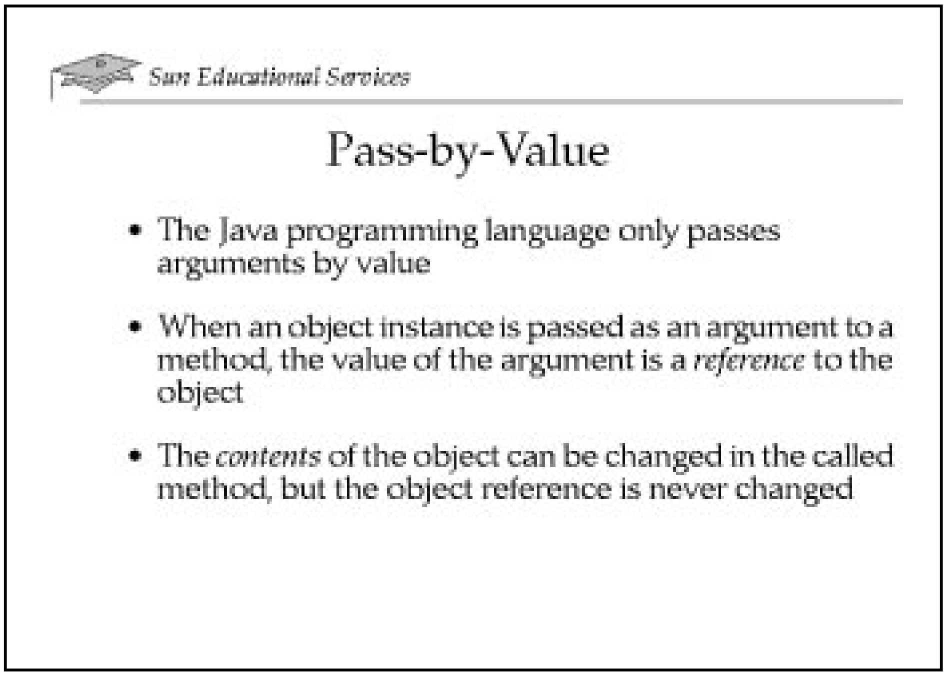
The <*argument_list*> allows argument values to be passed into a method. Elements of the list are separated by commas, while each element consists of a type and an identifier.

The `throws` <*exception*> clause causes a runtime error (exception) to be reported to the calling method so as to handle it in a suitable manner. The abnormal conditions are specified in `<exception>`.

For example,

```
public void addDays(int days) {
  < block > // Method code here
}
```

instructs the body of the method, `<block>`, to receive an argument indicating the number of days to add to the current date. In this method, the value is referenced by the identifier `days`.



Sun Educational Services

## Pass-by-Value

- The Java programming language only passes arguments by value
- When an object instance is passed as an argument to a method, the value of the argument is a *reference* to the object
- The *contents* of the object can be changed in the called method, but the object reference is never changed

## Object Fundamentals

### Pass-by-Value

The Java programming language passes arguments only “by value;” that is, the argument *cannot be changed* by the method called. When an object instance is passed as an argument to a method, the value of the argument is a reference to the object. The *contents* of the object can be changed in the called method, but the object reference is never changed.

The following code example illustrates this point:

```
1  public class PassTest {
2
3     float ptValue;
4
5     // Methods to change the current values
6     public void changeInt (int value) {
7        value = 55;
8     }
9
```

## Pass-by-Value (Continued)

```
public void changeStr (String value) {
   value = new String (“different”);
}

public void changeObjValue (PassTest ref) {
   ref.ptValue = 99.0F;
}

public static void main (String args[]) {

   String str;
   int val;

   // Create an instance of the class
   PassTest pt = new PassTest ();

   // Assign the int
   val = 11;

   // Try to change it
   pt.changeInt (val);

   // What is the current value?
   System.out.println (“Int value is: ” + val);

   // Assign the string
   str = new String (“hello”);

   // Try to change it
   pt.changeStr (str);

   // What is the current value?
   System.out.println (“Str value is: ” + str);

   // Now set the ptValue
   pt.ptValue = 101.0f;


```

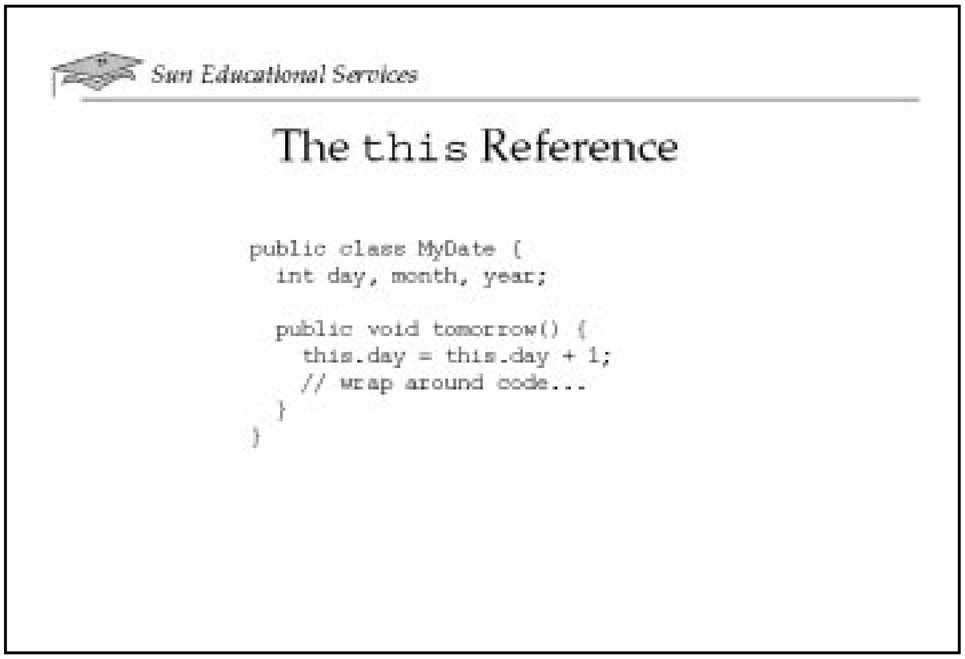
## *Pass-by-Value (Continued)*

```
      // Now change the value of the float
      // through the object reference
      pt.changeObjValue (pt);

      // What is the current value?
      System.out.println (“Current ptValue is: ” +
                           pt.ptValue);
   }
}
```

This code outputs the following:

```
java PassTest

Int value is: 11
Str value is: hello
Current ptValue is: 99.0
```

The `String` object is not changed by `changeStr()`; however, the contents of the `PassTest` object are changed.



Sun Educational Services

## The this Reference

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    this.day = this.day + 1;
    // wrap around code...
  }
}
```

## *Object Fundamentals*

### *The* `this` *Reference*

The keyword `this` is used to refer to the current object or class instance. Here, `this.day` refers to the day field of the current object.

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    this.day = this.day + 1;
    // wrap around code...
  }
}
```

## Object Fundamentals

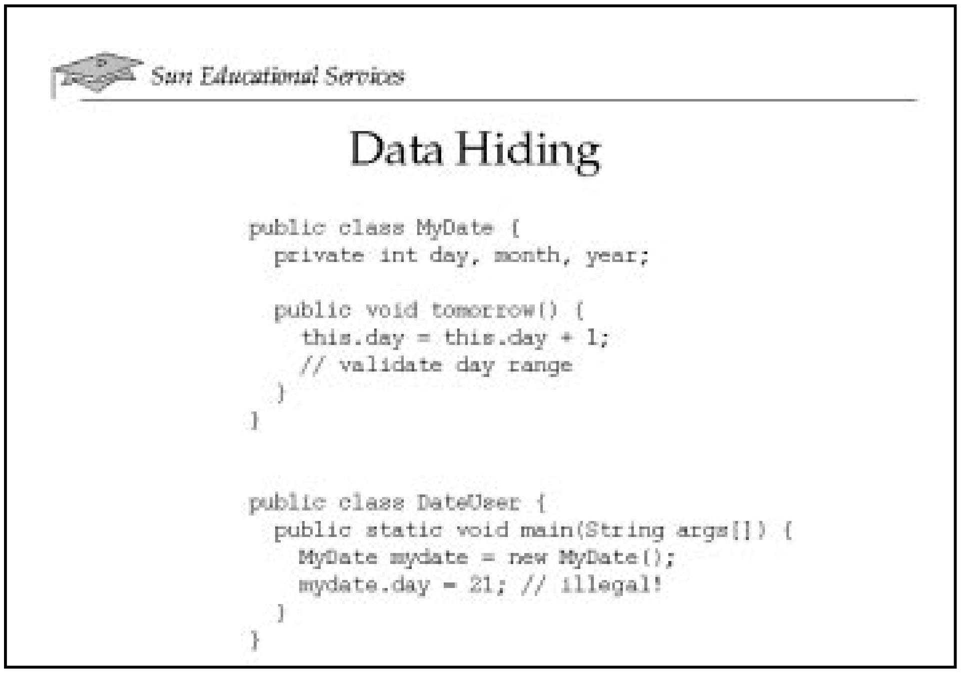
### The `this` Reference (Continued)

The Java programming language automatically associates all instance variable and method references with the `this` keyword, so using the keyword is redundant in certain circumstances. The following code is equivalent to the previous code:

```
public class MyDate {
  int day, month, year;

  public void tomorrow() {
    day = day + 1; // no 'this.' before 'day'
    // wrap around code...
  }
}
```

There are occasions when the keyword `this` is not redundant. For example, you might want to call a method in some entirely separate class and pass a reference to the current object as an argument. For example:

```
public class MyDate {
  int day, month, year;

  public void born() {
    Birthday bDay = new Birthday (this);
    sendAnnouncements();
  }
}
```


Sun Educational Services

Data Hiding

```
public class MyDate {
  private int day, month, year;

  public void tomorrow() {
    this.day = this.day + 1;
    // validate day range
  }
}


public class DateUser {
  public static void main(String args[]) {
    MyDate mydate = new MyDate();
    mydate.day = 21; // illegal!
  }
}
```

## Object Fundamentals

### Data Hiding

Using the `private` keyword in the declaration of `day`, `month`, and `year` in the `MyDate` class makes it impossible to access these members from any code except methods in the `MyDate` class itself. So given declaration for the `MyDate` class, the following code is illegal:

```
public class DateUser {
  public static void main(String args[]) {
    MyDate d = new MyDate();
    d.day = 21; // illegal!
  }
}
```

Preventing access to the data variables directly might seem odd, but it actually has great advantages for the quality of the program that uses the `MyDate` class. Since the individual items of data are inaccessible, the only way to read or write them is through methods. So if your program requires internal consistency of the members of the class, this can be managed by the methods of the class itself.

## Object Fundamentals

### Data Hiding (Continued)

Consider a `MyDate` class that allows arbitrary access to its members from outside. It would be very easy for code to do any of the following:

```
MyDate d = new MyDate();
d.day = 32; // invalid day
d.month = 2; d.day = 30; // plausible but wrong
d.month = d.month + 1; // no check for wrap around
```

**Caution –** These and other assignments result in invalid or inconsistent values in the fields of the `MyDate` object. Such a situation is unlikely to show up as a problem immediately, but will undoubtedly halt the program at some stage.

## Data Hiding (Continued)

If the data members of a class are not exposed, the user of the class is forced to modify the member variables by using methods. These methods can perform validity checks. Consider the following method as part of the `MyDate` class:

```
// part of MyDate class
public void setDay(int targetDay) {
   if (targetDay > this.daysInMonth()) {
      System.err.println("invalid day " + targetDay);
   }
   else {
      this.day = targetDay;
   }
}
```

The method checks to see if the day that it has been asked to set is valid. If the day is not valid, the method ignores the request and prints a message. You will see later that the Java programming language provides a much more effective mechanism for handling out-of-range method arguments. For now, however, you can see that the `MyDate` class is effectively protected against invalid dates.

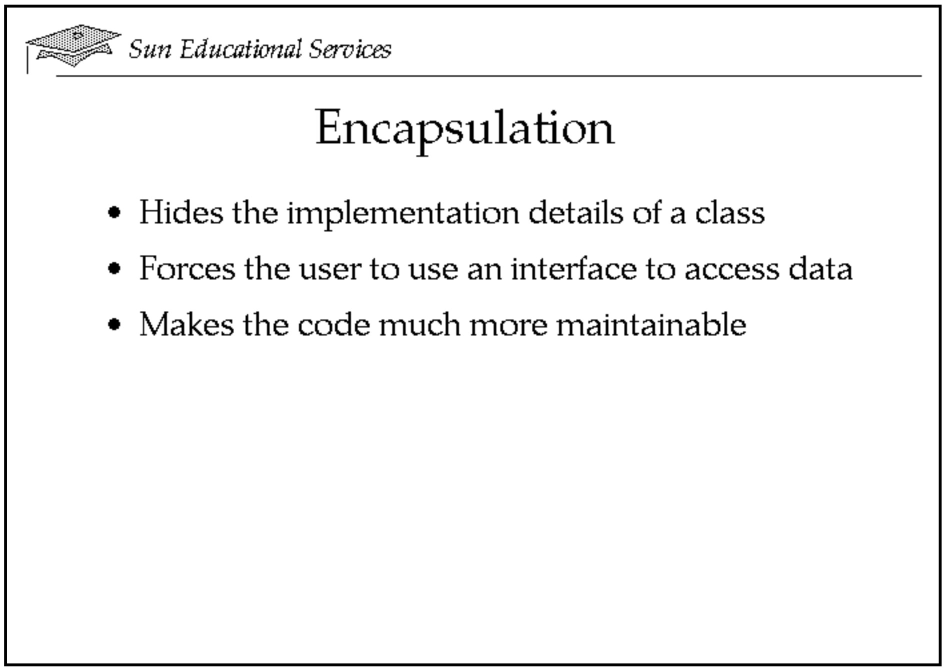
✓ ***What other checks could or should be added to this method?***

---

**Note –** In the previous example, `daysInMonth()` is assumed to be a method in the `MyDate` class. It therefore has a `this` value from which it can extract the `month` and `year` that are required to answer the query. As with the member variables, the use of `this` to qualify the `daysInMonth()` method is redundant.

---

Rules about how a method can properly be called such as "the value of the argument must be within the valid range of day numbers for the month in the object" are called *invariants (*or *pre-* and *post conditions).* Careful use of precondition tests can make a class much easier to reuse, and much more reliable in reuse, since the programmer using the class will find out immediately if any method has been misused.


Sun Educational Services

Encapsulation

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code much more maintainable

## Object Fundamentals

### Encapsulation

In addition to protecting the data of an object from improper modification, forcing the user to access the data through a method enables a class to be reused more simply by ensuring that required side effects are properly handled. In the case of the `MyDate` class, for example, consider how the `tomorrow()` method should be constructed.

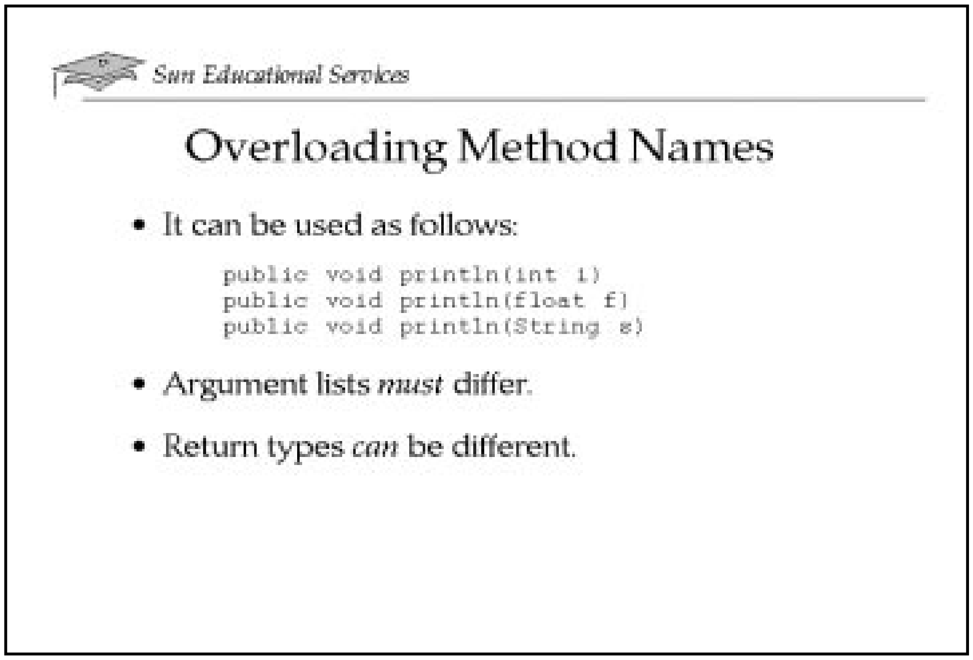
If the data were entirely accessible, each user of the class would have to increment the `day` value, test it against the number of days in the current month, and handle month-end (and possibly year-end) conditions. This is tedious and error prone. Although these needs are well understood for dates, other data types might have constraints that are not well known. By forcing the user of the class to use the supplied `tomorrow()` method, everyone can be assured that the necessary side effects will be handled consistently and correctly every time.

## Object Fundamentals

### Encapsulation (Continued)

Data hiding is generally referred to as *encapsulation*. It distinguishes the outside interface from the class from its implementation, hiding the implementation details of the class. Forcing the user to use the outside interface ensures that the calling code will still work even if the implementation changes, assuming the functionality through the interface remains the same. This makes the code much more maintainable.

✓ ***An optional group exercise is to model objects in your classroom as high-level Java software classes, noting their attributes and behaviors, and how they interact with each other. This brings the object-oriented paradigm into a real-world perspective.***



Sun Educational Services

## Overloading Method Names

- It can be used as follows:

```
public void println(int i)
public void println(float f)
public void println(String s)
```

- Argument lists *must* differ.
- Return types *can* be different.

## *Overloading Method Names*

In some circumstances, you might want to write several methods in the same class that do basically the same job with different arguments. Consider a simple method that is intended to output a textual representation of its argument. This method could be called `println()`.

Now suppose that you need a different print method for printing each of the `int`, `float`, and `String` types. This is reasonable, since the various data types require different formatting, and probably varied handling. You could create three methods, called `printInt()`, `printFloat()`, and `printString()`, respectively. However, this is tedious.

Java, along with several other programming languages, allows you to reuse a method name for more than one method. Clearly, this can work only if there is something in the circumstances under which the call is made that will distinguish which method is actually needed. In the case of the three print methods, it is possible to make this distinction based on the number and type of the arguments.

## Overloading Method Names

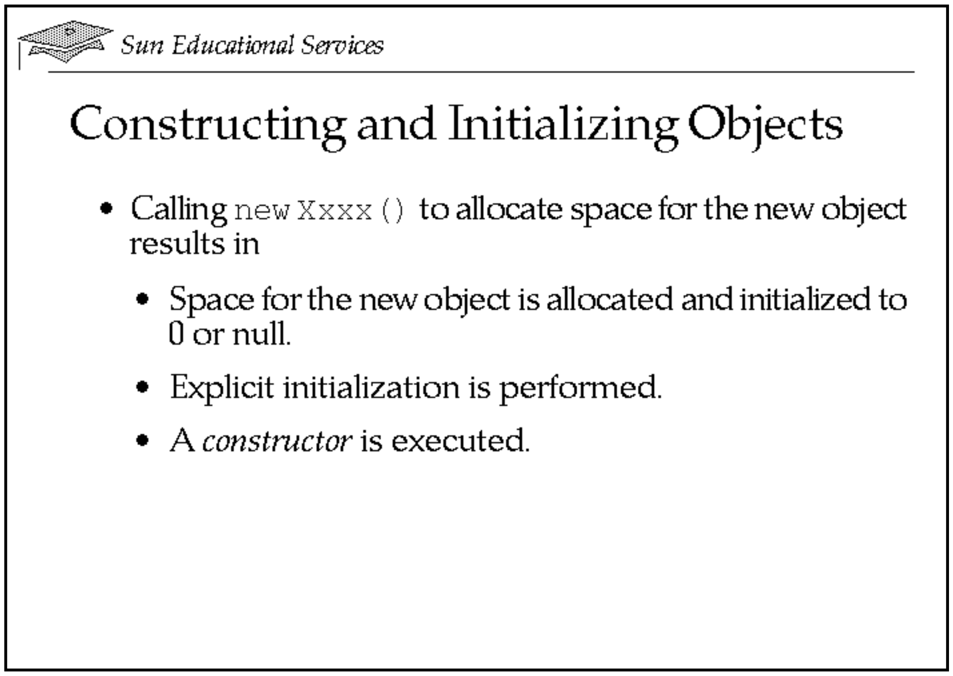
By reusing the method name, you end up with the following methods:

```
public void println(int i)
public void println(float f)
public void println()
```

When you write code to call one of these methods, the appropriate one is chosen according to the type of argument or arguments you supply.

Two rules apply to overloaded methods:

- The argument lists of the calling statement must differ enough to allow unambiguous determination of the proper method to call. Normal widening promotions (for example, `float` to `double`) might be applied; this can cause confusion under some conditions.
- The return type of the methods can be different, but it is not sufficient for the return type to be the only difference. The argument lists of overloaded methods *must* differ.



Sun Educational Services

## Constructing and Initializing Objects

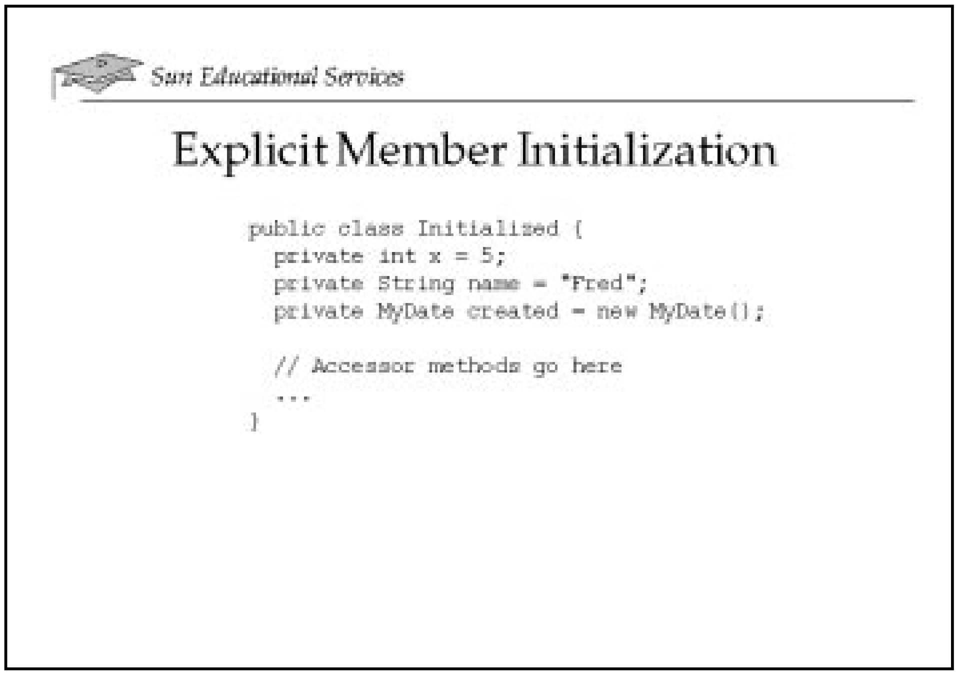
- Calling `new Xxxx()` to allocate space for the new object results in
  - Space for the new object is allocated and initialized to 0 or null.
  - Explicit initialization is performed.
  - A *constructor* is executed.

## *Constructing and Initializing Objects*

You have seen how you must execute a call to `new Xxx()` in order to allocate space for a new object. You will see that sometimes arguments can be placed in the parentheses, for example; `new Button(″Press me″)`. Using the keyword `new` causes the following things to happen:

- First, the space for the new object is allocated and initialized to the form of zero or null. In the Java programming language, this phase is indivisible to ensure that you cannot have an object with random values in it.
- Second, any explicit initialization is performed.
- Third, a *constructor*, which is a special method, is executed. Arguments passed in the parentheses to `new` are passed to the constructor (“`Press me`”).

This section investigates these last two phases.



Sun Educational Services

### Explicit Member Initialization

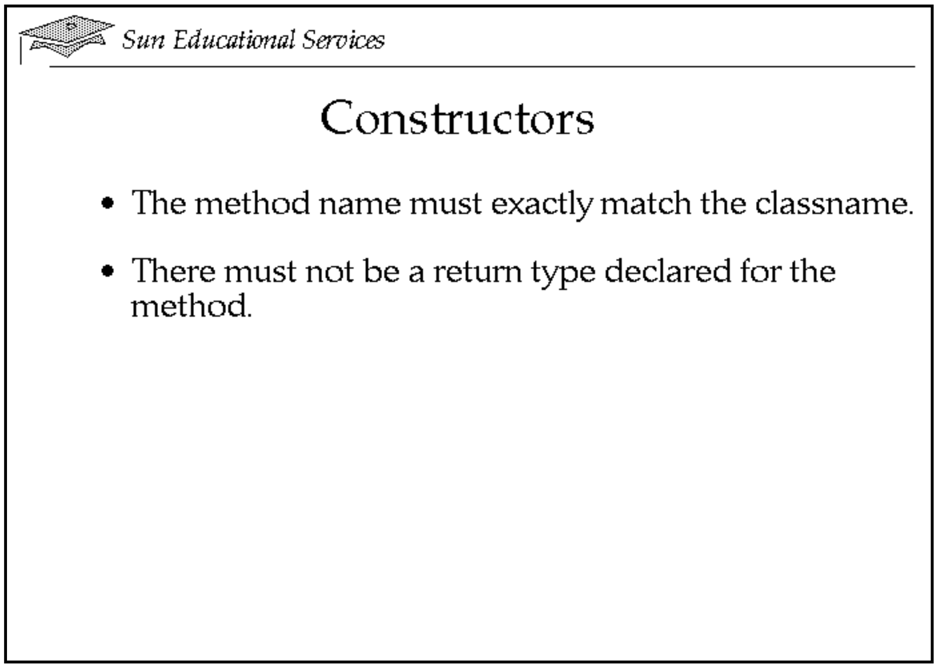
```
public class Initialized {
  private int x = 5;
  private String name = "Fred";
  private MyDate created = new MyDate();

  // Accessor methods go here
  ...
}
```

## Constructing and Initializing Objects

### Explicit Member Initialization

If you place simple assignment expressions in your member declarations, you can perform explicit member initialization during construction of your object.

```
public class Initialized {
  private int x = 5;
  private String name = “Fred”;
  private MyDate created = new MyDate();

  // Methods go here
  ...
}
```



Sun Educational Services

## Constructors

- The method name must exactly match the classname.
- There must not be a return type declared for the method.

## *Constructing and Initializing Objects*

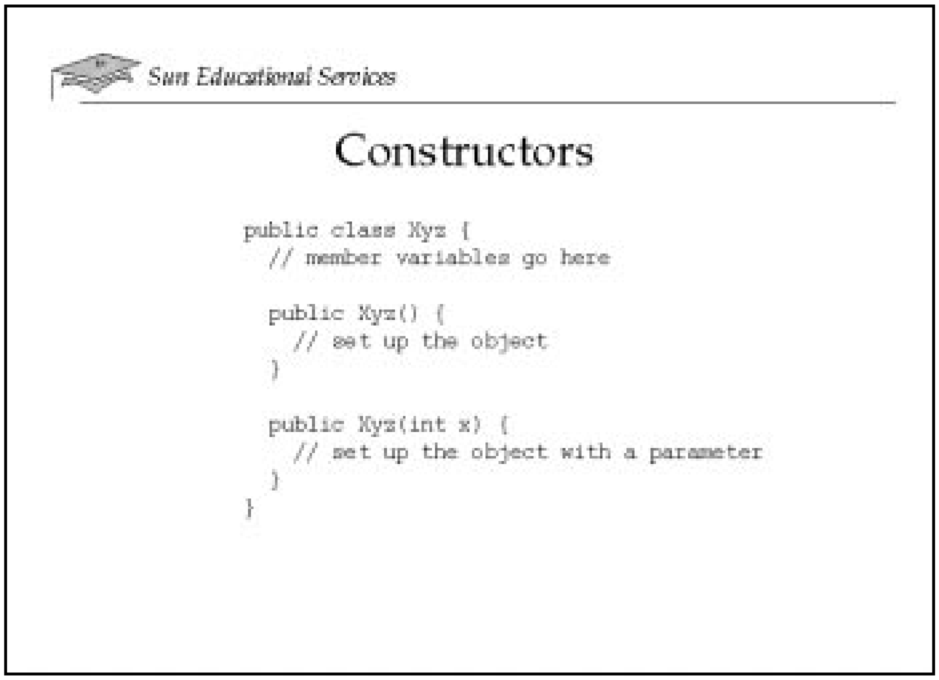
### *Constructors*

The explicit initialization mechanism just described provides a simple way to set the starting values of fields in an object. Sometimes, however, you need to execute a method to perform the initialization. or handle possible errors. You can use loops or conditionals to determine the initialization. You can also pass arguments into the construction process so that the code that requests the construction of the new object can control the object it creates.

The final stage of initialization of a new object is to call a method called a *constructor.*

Constructors are identified by the two following rules:

- The method name must exactly match the classname.
- There must not be a return type declared for the method.



Sun Educational Services

## Constructors

```
public class Xyz {
  // member variables go here

  public Xyz() {
    // set up the object
  }

  public Xyz(int x) {
    // set up the object with a parameter
  }
}
```

## *Constructing and Initializing Objects*

### *Constructors (Continued)*

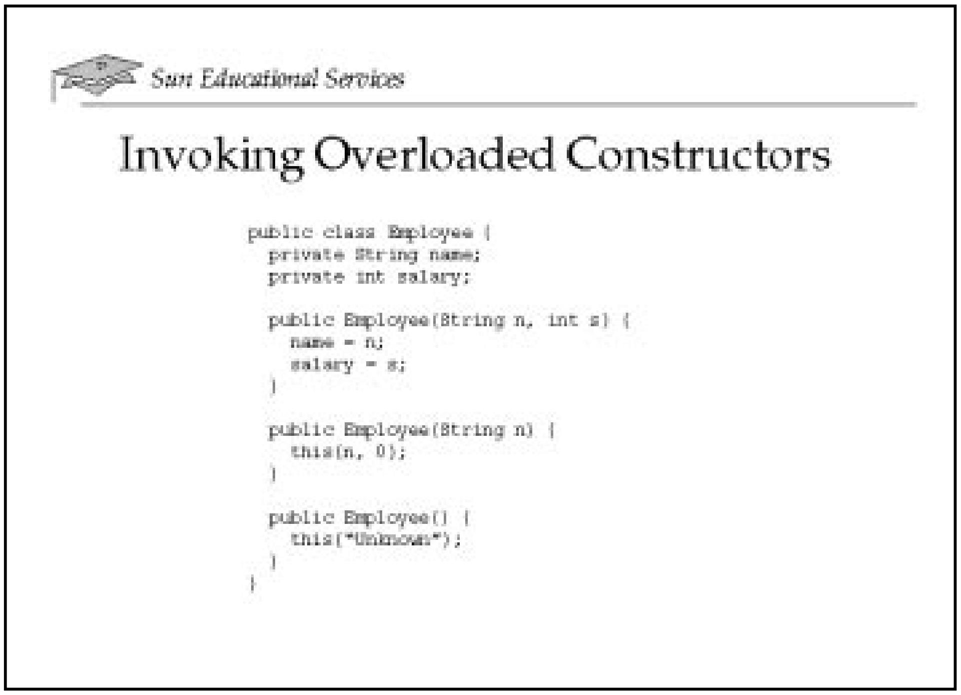
✓ ***Declaring a return type causes the constructor to appear as a method that never gets called, thus no compile error occurs.***

```
public class Xyz {
   // member variables
   public Xyz() {           // No-arg constructor
      // set up the object.
   }
   public Xyz(int x) {       //int-arg constructor
      // set up the object using the parameter x.
   }
}
```

---

**Note –** As with methods, you can overload the constructor name by providing several constructors with different argument lists. When you issue a `new Xyz(argument_list)` call, the argument list passed in the `new` statement determines which constructor is used.

---



Sun Educational Services

## Invoking Overloaded Constructors

```
public class Employee {
  private String name;
  private int salary;

  public Employee(String n, int s) {
    name = n;
    salary = s;
  }

  public Employee(String n) {
    this(n, 0);
  }

  public Employee() {
    this("Unknown");
  }
}
```

## *Constructing and Initializing Objects*

### *Invoking Overloaded Constructors*

If you have a class with several constructors, you might want to duplicate the effect of one inside a different one. This can be achieved using the keyword `this` as a method call. For example:

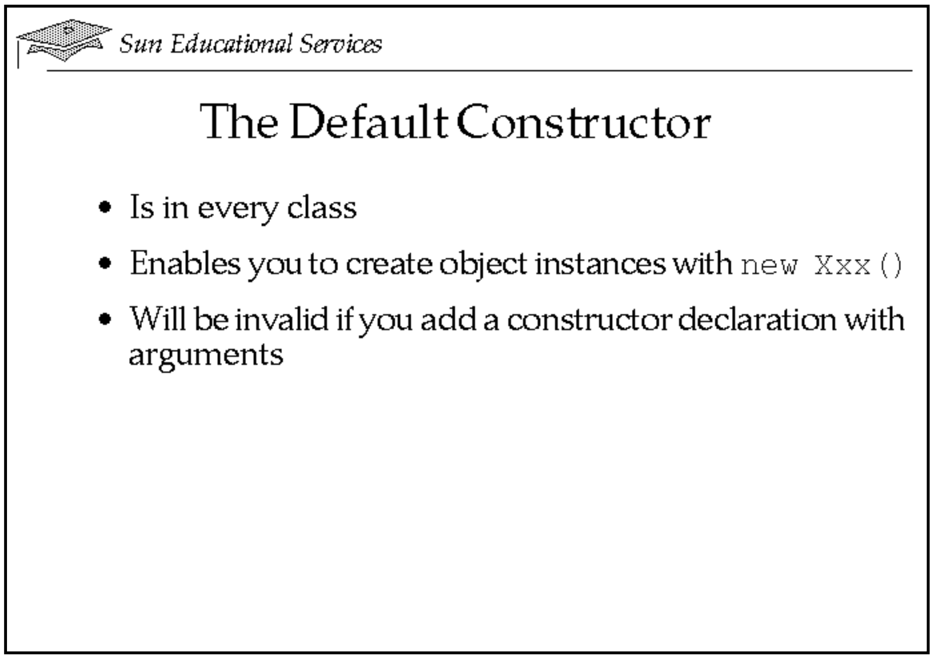
## *Invoking Overloaded Constructors (Continued)*

```
public class Employee {
  private String name;
  private int salary;

  public Employee(String n, int s) {
    name = n;
    salary = s;
  }

  public Employee(String n) {
    this(n, 0);
  }

  public Employee() {
    this("Unknown");
  }
}
```

In the second constructor, which has one `String` argument, the call `this(n,0)` passes control to the version of the constructor that takes one `String` argument and one `int` argument.

In the third constructor, which has no arguments, the call `this(″Unknown″)` passes control to the version of constructor that takes one `String` argument.

**Note –** Any call to `this`, if present, must be the first statement in any constructor.



Sun Educational Services

## The Default Constructor

- Is in every class
- Enables you to create object instances with `new Xxx()`
- Will be invalid if you add a constructor declaration with arguments

## *Constructing and Initializing Objects*

### *The Default Constructor*

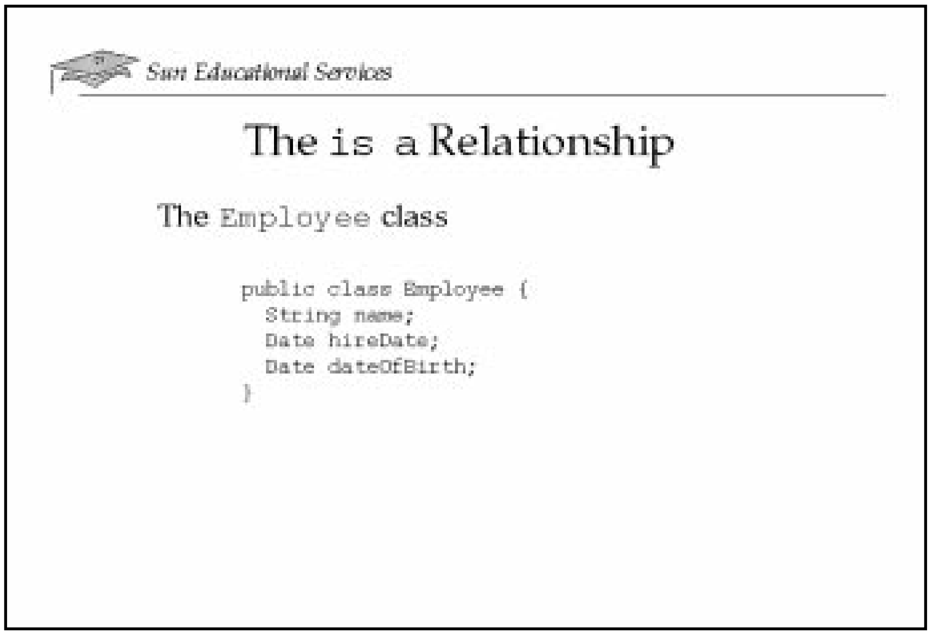
Every class has at least one constructor. If you do not write a constructor, the Java programming language provides one for you. This constructor takes no arguments and has an empty body.

The default constructor enables you to create object instances with `new Xxx()`; otherwise, you would be required to provide a constructor for every class.

---

**Note –** It is important to realize that if you add a constructor declaration with arguments to a class that previously had no explicit constructors, you will lose the default constructor. From that point, calls to `new Xxx()` will cause compiler errors.

---


Sun Educational Services

## The is a Relationship

The Employee class

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
}
```

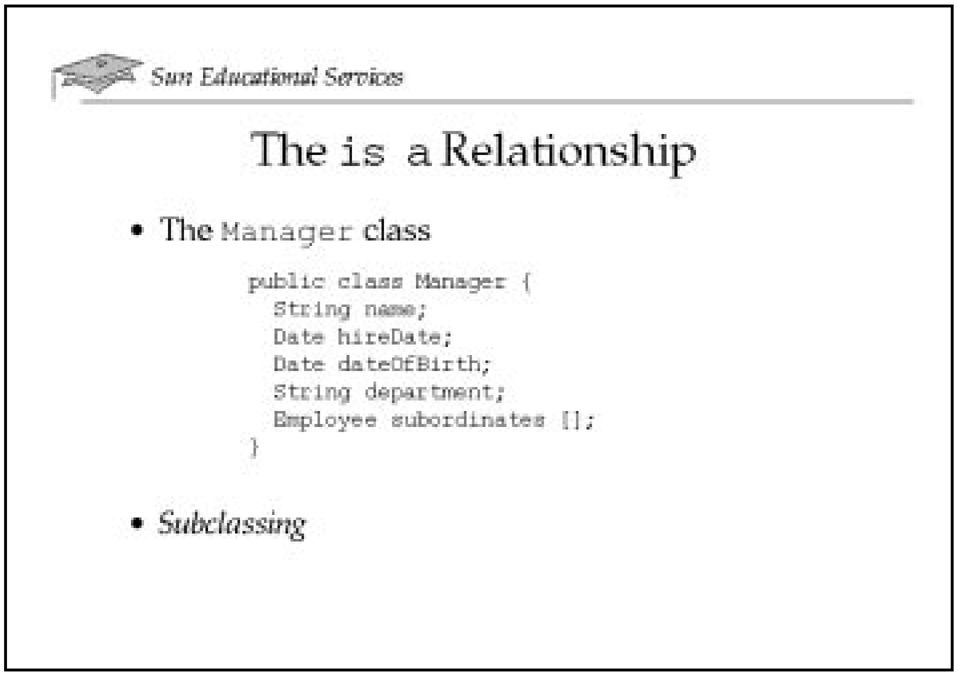
## Subclassing

### The `is a` Relationship

In programming you will often create a model of something (for example, an employee), and then need a more specialized version of that original model. For example, you might want a model for a manager. Clearly a manager actually is an *employee*, but an employee with additional features.

Consider the following sample class declarations that demonstrate this:

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
  ...
}
```



Sun Educational Services

## The is a Relationship

- The Manager class

```
public class Manager {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String department;
  Employee subordinates [];
}
```

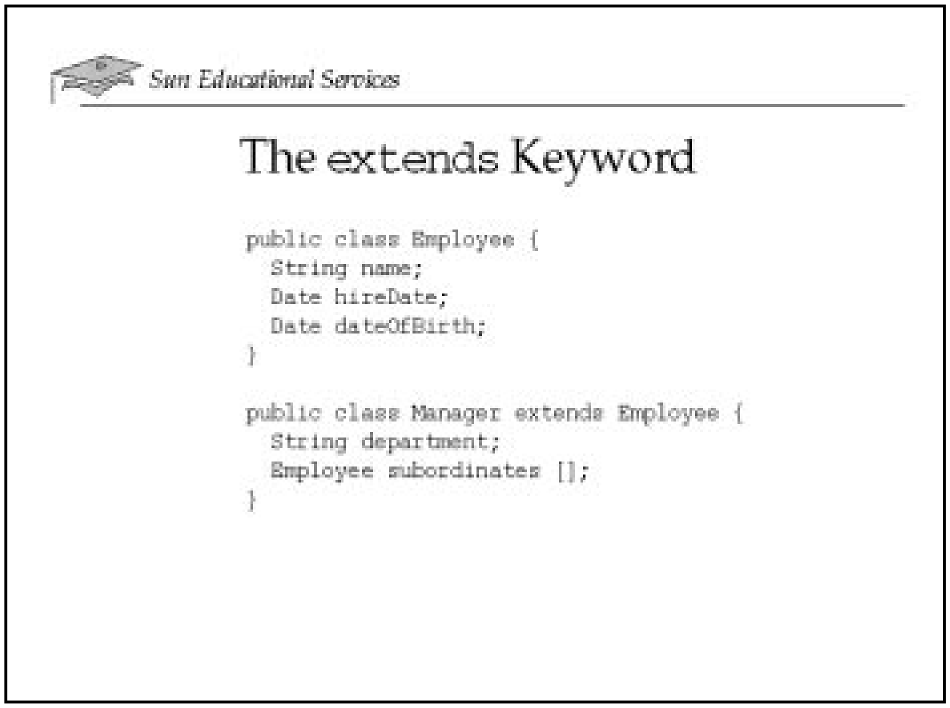
- *Subclassing*

## *Subclassing*

### *The* `is a` *Relationship (Continued)*

```
public class Manager {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String jobTitle;
  int grade;
  String department;
  Employee subordinates [];
  ...
}
```

This example illustrates the duplication of data between the `Manager` class and the `Employee` class. Additionally, there could be a number of methods that would be applicable to both `Employee` and `Manager`. Therefore, you need a way to create a new class from an existing class; this is called *subclassing*.


Sun Educational Services

## The extends Keyword

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
}

public class Manager extends Employee {
  String department;
  Employee subordinates [];
}
```

## *Subclassing*

### *The* `extends` *Keyword*

In object-oriented languages, special mechanisms are provided that allow the programmer to define a class in terms of a previously defined class. This is achieved using the keyword `extends` as follows:

```
public class Employee {
  String name;
  Date hireDate;
  Date dateOfBirth;
  String jobTitle;
  int grade;
  ...
}

public class Manager extends Employee {
  String department;
  Employee subordinates [];
  ...
}
```

## Subclassing

### *The* `extends` *Keyword (Continued)*

In this arrangement, the `Manager` class is defined to have all the variables and methods that an `Employee` has. All these variables and methods are *inherited* from the definition of the *parent class*. All the programmer needs to do is define the extra features or specify which changes to apply.

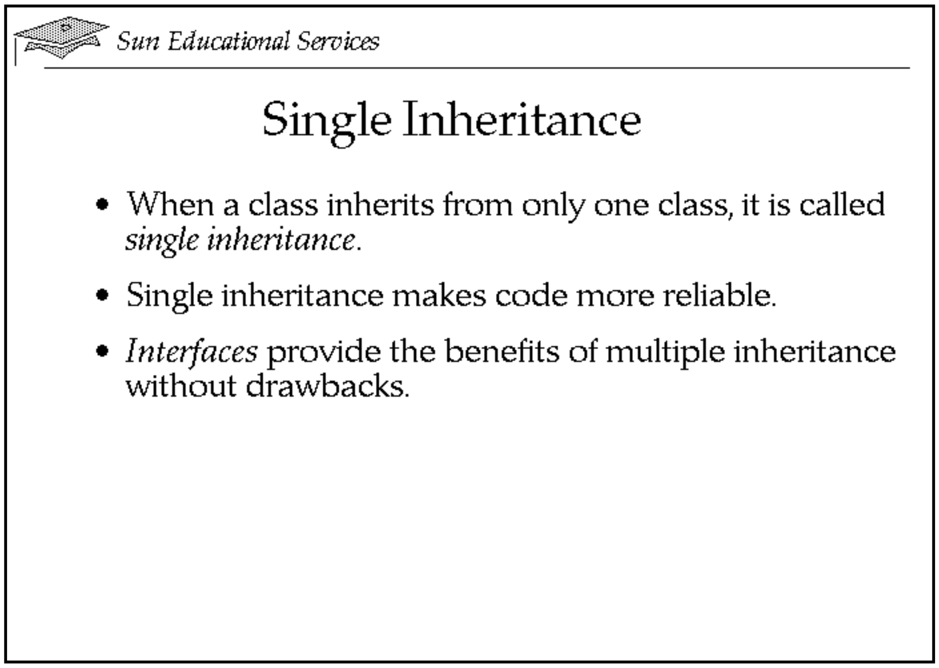
---

**Note –** This approach is a great improvement in terms of maintenance and reliability. If a correction is made in the `Employee` class, then the `Manager` class is corrected automatically without the programmer having to do anything except compile it.

---

**Note –** A description of an inherited method or variable exists only in the API documentation page of the class in which that member is defined. Be sure to check the parent class and other ancestor classes for inherited members when exploring the features of a (child) class.

---


Sun Educational Services

## Single Inheritance

- When a class inherits from only one class, it is called *single inheritance*.
- Single inheritance makes code more reliable.
- *Interfaces* provide the benefits of multiple inheritance without drawbacks.

## Subclassing

### Single Inheritance

The Java programming language allows a class to extend only one other class. This restriction is called *single inheritance*. The relative merits of single and multiple inheritance are the subject of extensive discussions among object-oriented programmers. The Java programming language imposes the single inheritance restriction to make the resulting code more reliable, although this sometimes makes more work for the programmer. Module 6 examines a language feature called *interfaces* that allows most of the benefits of multiple inheritance without suffering from any of its drawbacks.

## *Subclassing*

### *Single Inheritance*

An example of subclassing using inheritance is shown in Figure 5-1.

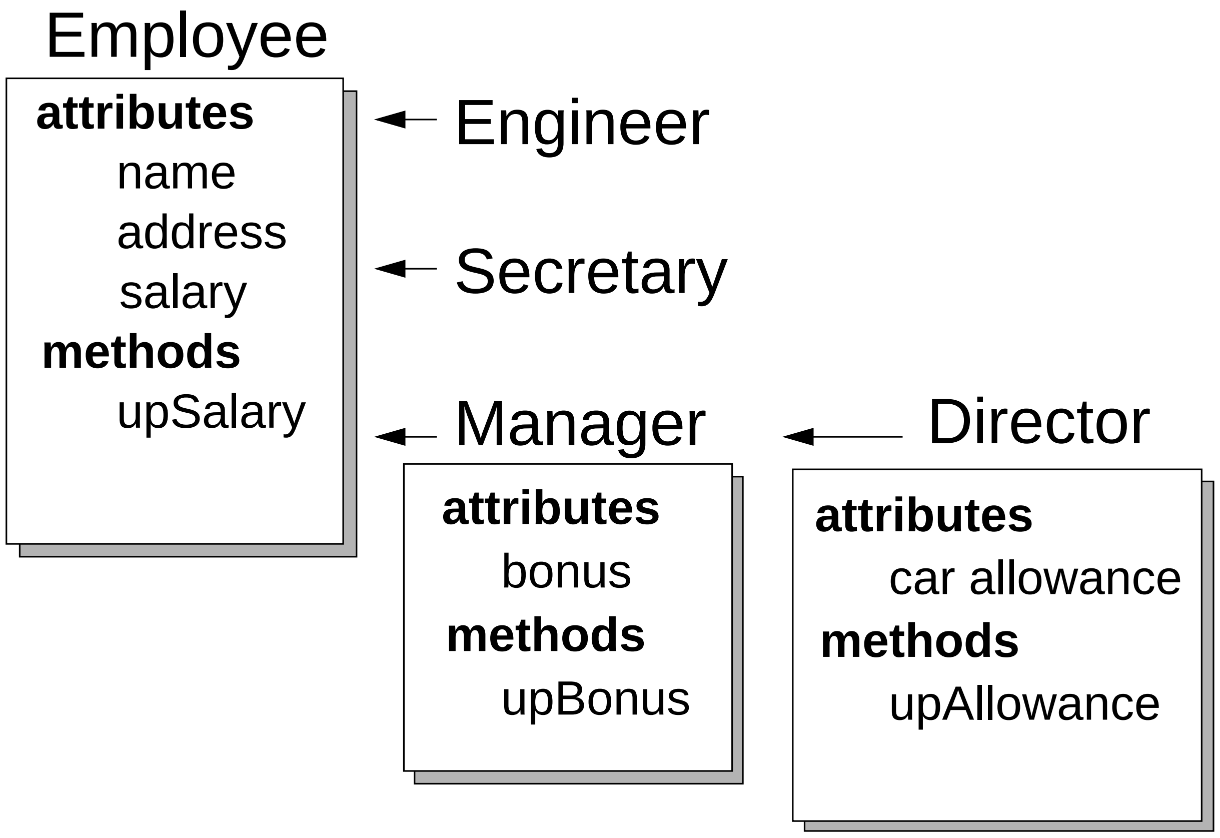
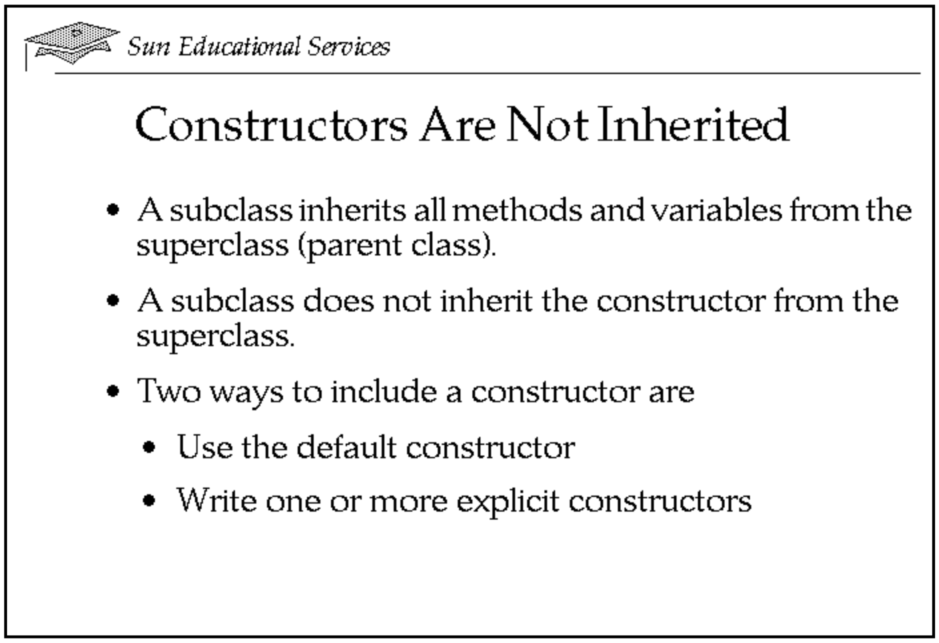


**Figure 5-1** Inheritance



Sun Educational Services

## Constructors Are Not Inherited

- A subclass inherits all methods and variables from the superclass (parent class).
- A subclass does not inherit the constructor from the superclass.
- Two ways to include a constructor are
  - Use the default constructor
  - Write one or more explicit constructors

## *Subclassing*

### *Constructors Are Not Inherited*

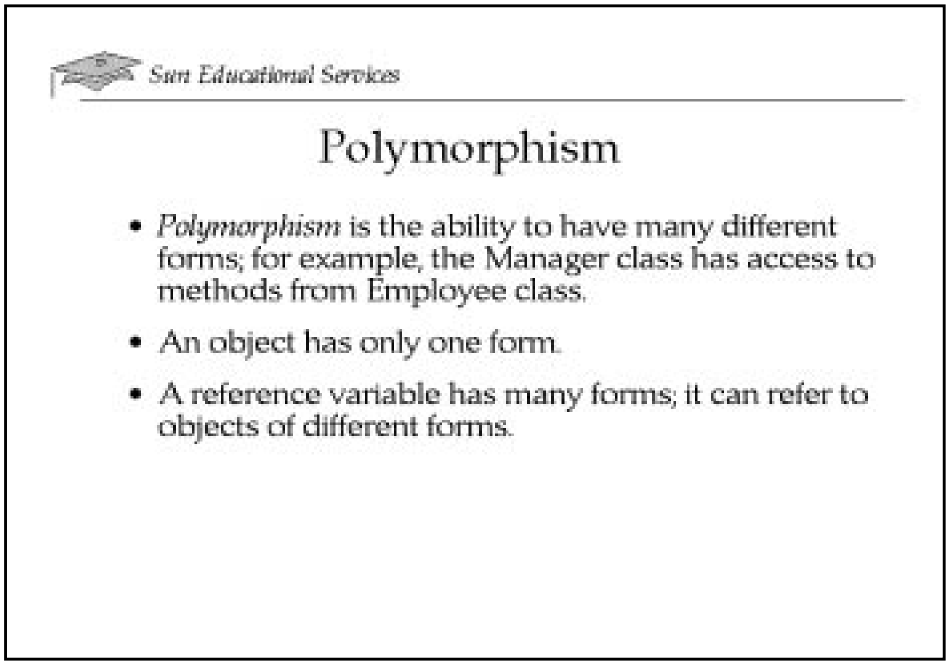
It is important to know that although a subclass inherits all of the methods and variables from a parent class, it does not inherit constructors.

There are only two ways that a class can gain a constructor; either you write the constructor, or, because you have written no constructors at all, the class has a single default constructor.

---

**Note –** A parent constructor is always called in addition to a child constructor. This will be discussed in detail in the next module.

---



Sun Educational Services

## Polymorphism

- *Polymorphism* is the ability to have many different forms; for example, the Manager class has access to methods from Employee class.
- An object has only one form.
- A reference variable has many forms; it can refer to objects of different forms.

## Subclassing

### Polymorphism

Describing a `Manager` as an `Employee` is not just a convenient way of describing the relationship between these two classes. Recall that the `Manager` has all the attributes, both member variables and methods, of the parent class `Employee`. This means that any operation that is legitimate on an `Employee` is also legitimate on a `Manager`. If the `Employee` has methods `raiseSalary()` and `fire()`, then the `Manager` class does also.

An *object* has only one form (the one that is given to it when constructed). However, a *variable* is polymorphic since it can refer to objects of different forms. In the Java programming language, there is one class that is a parent class for all others, `java.lang.Object` class. So, in effect, earlier definitions were shorthand for

`public class Employee extends Object` *and*
`public class Manager extends Employee`

## Polymorphism (Continued)

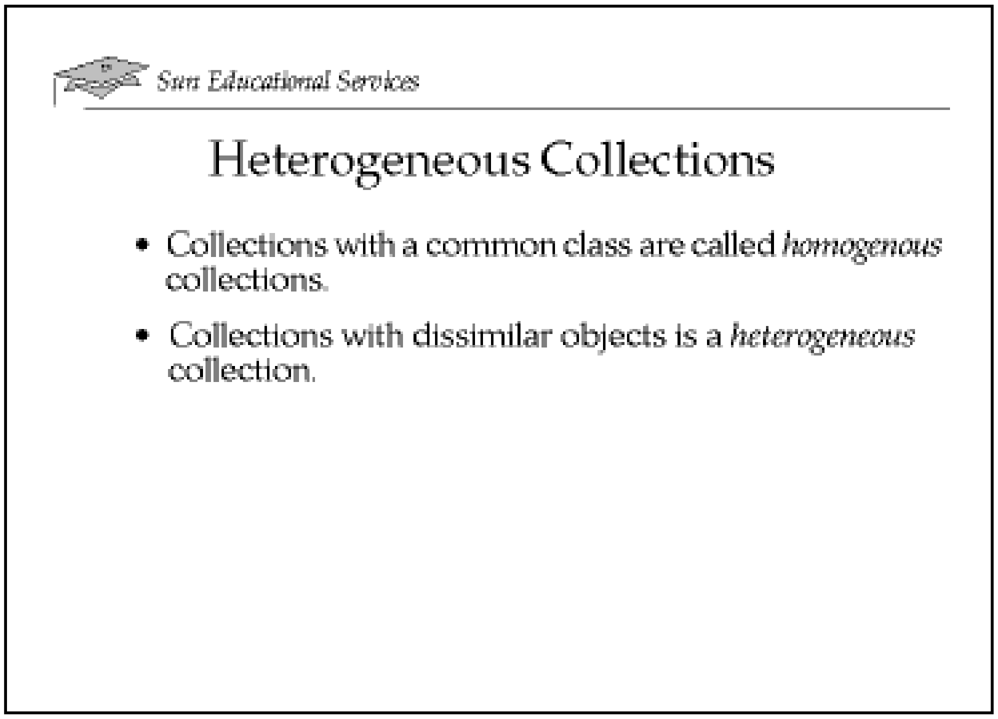
The `Object` class defines a number of useful methods, including `toString()`, which is why everything in Java software can be converted to a string representation (even if the result is of limited use).

Java, like most object-oriented languages, actually allows you to refer to an object with a variable that is one of the parent class types. So you can say

```
Employee e = new Manager();
```

Using the variable `e` as is, you can access only the parts of the object that are part of an `Employee`; the `Manager`-specific parts are hidden. This is because as far as the compiler is concerned, `e` is an `Employee`, not a `Manager`. Therefore, the following is not allowed:

```
// Illegal attempt to assign Manager member
// variable when object is a parent Employee class
e.department = "Finance";
```



Sun Educational Services

## Heterogeneous Collections

- Collections with a common class are called *homogenous* collections.
- Collections with dissimilar objects is a *heterogeneous* collection.

## *Subclassing*

### *Heterogeneous Collections*

You can create collections of objects that have a common class. Such collections are called *homogenous* collections.

The Java programming language has an `Object` class, so it is possible to make collections of all kinds of elements due to polymorphism, as all classes extend class `Object`. Such collections are called *heterogeneous* collections.

It might seem unrealistic to create a `Manager` and deliberately assign the reference to it to a variable of type `Employee`. This is possible and there are reasons why you might want to achieve this effect.

## *Heterogeneous Collections (Continued)*

A *heterogeneous* collection is a collection of dissimilar things. In object-oriented languages, you can create collections of many things. All have a common ancestor class – the `Object` class. For example:

```
Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
```

You can even write a sort method that puts the employees into age or salary order, regardless of whether some are managers.

---

**Note –** Every class is a subclass of `Object`, so you can use an array of `Object` as a container for any objects. The only things that cannot be added to an array of `Object` are primitive variables. Wrapper classes, as discussed in Module 6, take care of this.) Better than an array of `Object`, though, is the `Vector` class which is designed to store heterogeneous collections of objects.

---

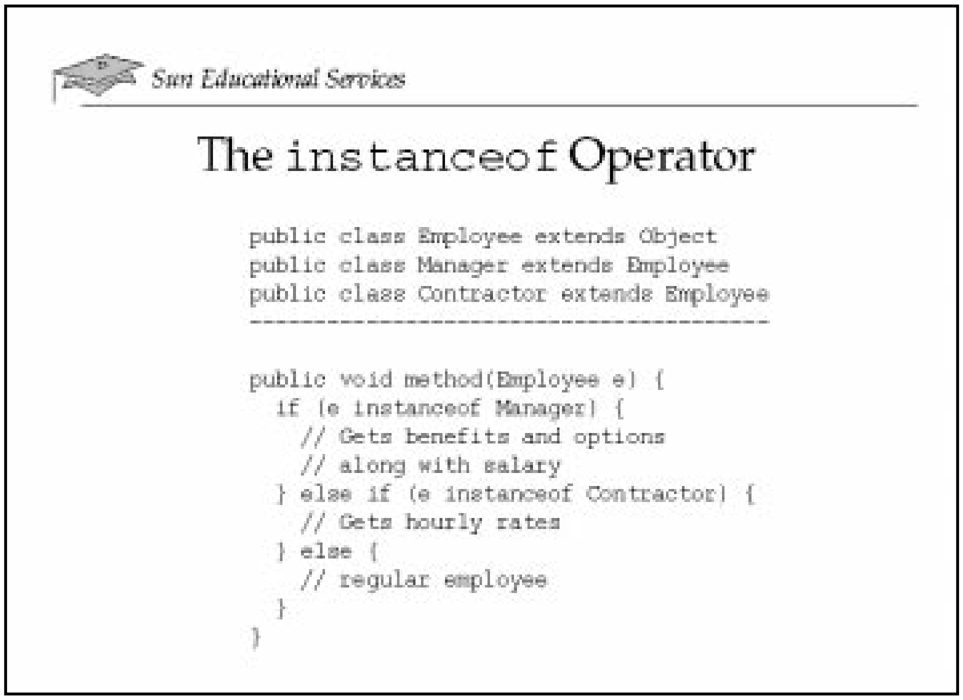
## Subclassing

### Polymorphic Arguments

Using this approach you can write methods that accept a "generic" object, in this case, the class Employee, and work properly on objects of any subclass of it. Hence you might produce a method in an application class that takes an employee and compares it with a certain threshold salary to determine the tax liability of that employee. Using the polymorphic features, you can do this as follows:

```
// In the Employee class
public TaxRate findTaxRate(Employee e) {
  // do calculations and return a tax rate for e
}

// Meanwhile, elsewhere in the application class
Manager m = new Manager();
:
TaxRate t = findTaxRate(m);
```

This is legal, because a Manager *is an* Employee.



Sun Educational Services

## The instanceof Operator

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
----------------------------------------

public void method(Employee e) {
  if (e instanceof Manager) {
    // Gets benefits and options
    // along with salary
  } else if (e instanceof Contractor) {
    // Gets hourly rates
  } else {
    // regular employee
  }
}
```

## *Subclassing*

### *The* `instanceof` *Operator*

Given that you can pass objects around using references to their parent classes, sometimes you might want to know what you actually have. This is the purpose of the `instanceof` operator. Suppose the class hierarchy is extended so that you have

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
```

---

**Note –** Remember that, while acceptable, `extends Object` is actually redundant. It is shown here only as a reminder.

---

## Subclassing

### The `instanceof` Operator (Continued)

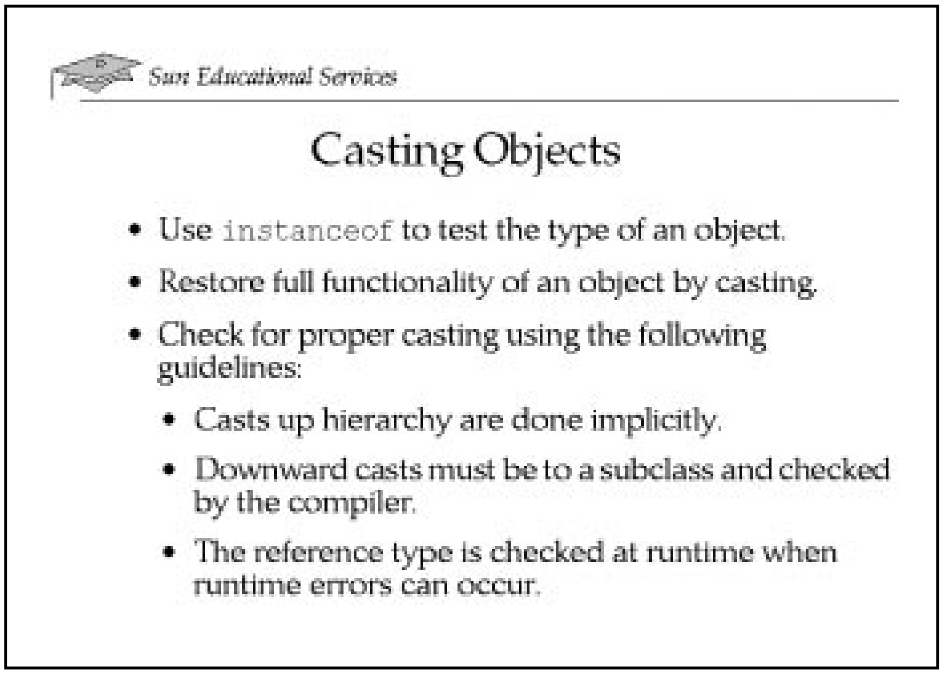
If you receive an object via a reference of type `Employee`, it might or might not turn out to be a `Manager` or a `Contractor`. You can test it by using `instanceof` as follows:

```
public void method(Employee e) {
  if (e instanceof Manager) {
    // Get benefits and options
    // along with salary
  } else if (e instanceof Contractor) {
    // Get hourly rates
  } else {
    // regular employee
  }
}
```

---

**Note –** In C++ you can do something similar using RTTI (runtime-type information), but `instanceof` in the Java programming language is more powerful.

---


Sun Educational Services

Casting Objects

- Use instanceof to test the type of an object.
- Restore full functionality of an object by casting.
- Check for proper casting using the following guidelines:
  - Casts up hierarchy are done implicitly.
  - Downward casts must be to a subclass and checked by the compiler.
  - The reference type is checked at runtime when runtime errors can occur.

## Subclassing

### Casting Objects

In circumstances where you have received a reference to a parent class, and you have determined that the object is actually a particular subclass by using the `instanceof` operator, you can restore the full functionality of the object by casting the reference.

```
public void method(Employee e) {
  if (e instanceof Manager) {
    Manager m = (Manager)e;
    System.out.println(“This is the manager of ” +
                        m.department);
  }
  // rest of operation
}
```

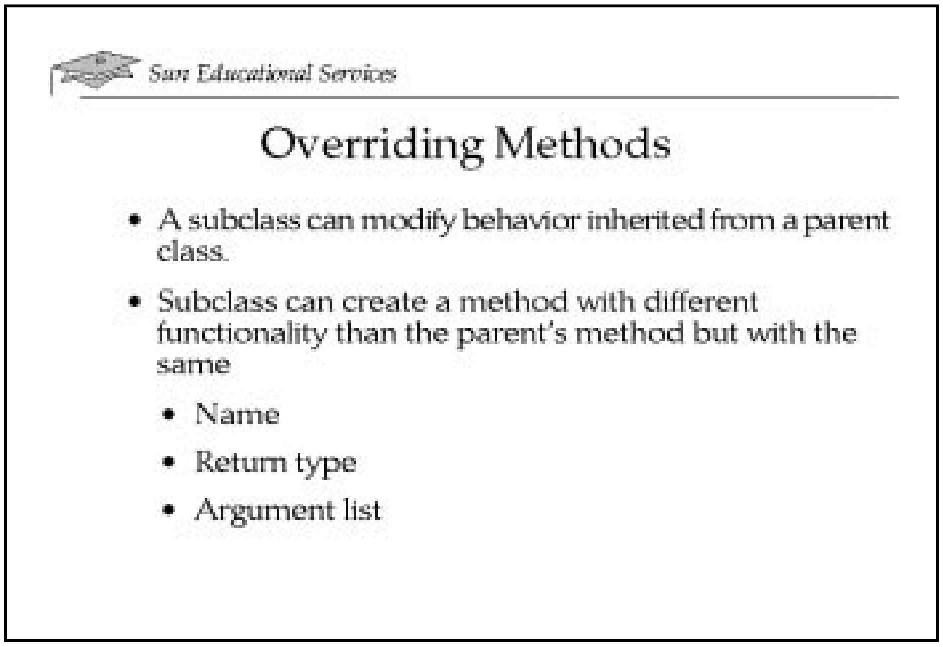
If you do not make the cast, an attempt to reference `e.department` would fail, as the compiler cannot locate a member called `department` in the `Employee` class.

## Subclassing

### Casting Objects

If you do not make the test using `instanceof`, you run the risk of the cast failing. Generally any attempt to cast an object reference is subjected to several checks:

- Casts "up" the class hierarchy are always permitted, and in fact do not require the cast operator. They can be done by simple assignment.
- For "downward" casts, the compiler must be satisfied that the cast is at least possible. For example, any attempt to cast a `Manager` reference to a `Contractor` reference is definitely not permitted, since the `Contractor` is not a `Manager`. The class to which the cast is taking place must be some subclass of the current reference type.
- If the compiler allows the cast, then the reference type is checked at runtime. For example, if it turns out that the `instanceof` check is omitted from the source, and the object being cast is not in fact an object of the type it is being cast to, then a runtime error (*exception)* occurs. Exceptions are a form of runtime error, and are the subject of a later module.


Sun Educational Services

## Overriding Methods

- A subclass can modify behavior inherited from a parent class.
- Subclass can create a method with different functionality than the parent's method but with the same
  - Name
  - Return type
  - Argument list

## *Overriding Methods*

In addition to being able to produce a new class based on an old one by adding additional features, it is possible to modify existing behavior of the parent class.

If a method is defined in a new class such that the name, return type, and argument list exactly match those of a method in the parent class, then the new method is said to *override* the old one.

---

**Note –** Remember that methods with the same name but different argument lists which are in the same class, are simply overloaded. This causes the compiler to use the supplied arguments to determine which method to call.

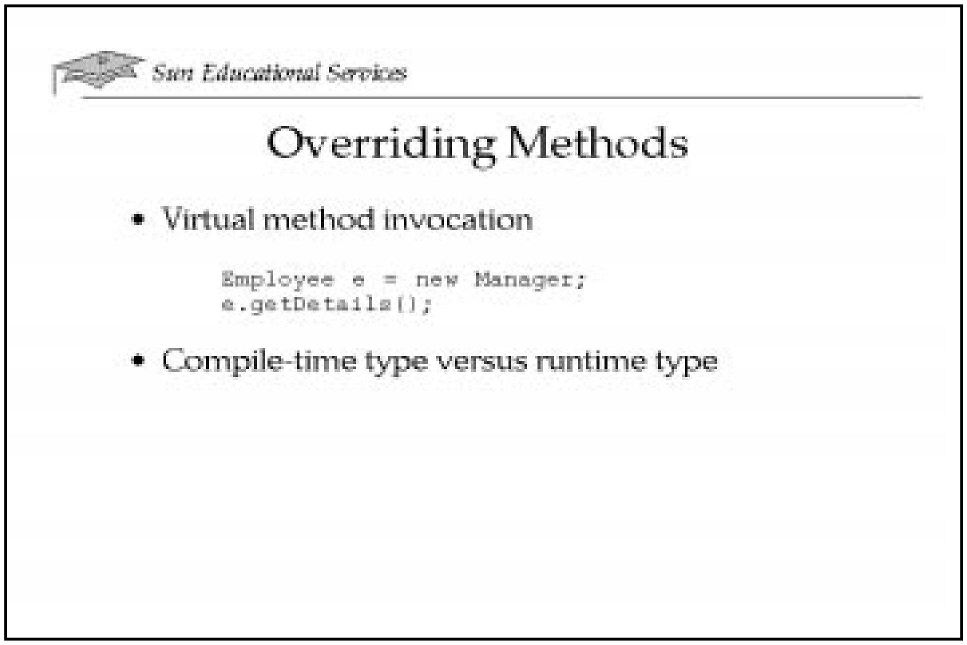
---

## Overriding Methods

Consider these sample methods in the Employee and Manager classes:

```
public class Employee {
  String name;
  int salary;

  public String getDetails() {
    return “Name: “ + name + “\n” +
           “Salary: “ + salary;
  }
}

public class Manager extends Employee {
  String department;

  public String getDetails() {
    return “Name: “ + name + “\n” +
           “Manager of “ + department;
  }
}
```

The Manager class has a getDetails() method by definition because it inherits one from the Employee class. The original method has been replaced, or overridden, by the child class’s version, however.



Sun Educational Services

## Overriding Methods

- Virtual method invocation

```
Employee e = new Manager;
e.getDetails();
```

- Compile-time type versus runtime type

## *Overriding Methods*

Assume that the example on the previous page and the following scenario are true:

```
Employee e = new Employee();
Manager m = new Manager();
```

If you ask for `e.getDetails()` and `m.getDetails()`, you invoke different behaviors. The `Employee` object will execute the version of `getDetails()` associated with the `Employee` class, and the `Manager` object will execute the version of `getDetails()` associated with the `Manager` class.

What is less obvious is what happens if you have

```
Employee e = new Manager();
e.getDetails();
```

or some similar effect, like a general method argument or an item from a heterogeneous collection.

## Overriding Methods

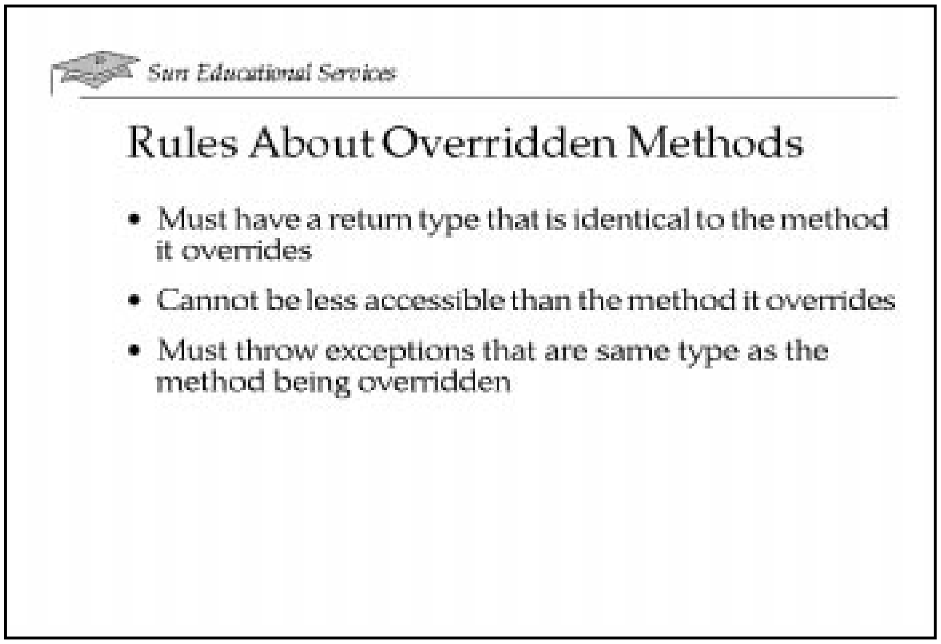
In fact, you get the behavior associated with the runtime type of the variable (that is, the type of the object referred to by the variable), not the behavior associated with the compile time type of the variable. This is an important feature of object-oriented languages. It is also another important feature of polymorphism and is often referred to as *virtual method invocation*.

In the previous example, the `e.getDetails()` method executed is from the object’s real type, a `Manager`.

---

**Note –** If you are a C++ programmer, there is an important distinction to be drawn between the Java programming language and C++. In C++ you only get this behavior if you mark the method as `virtual` in the source. In “pure” object-oriented languages; however, this is not normal. C++ does this, of course, to increase execution speed.

---



Sun Educational Services

## Rules About Overridden Methods

- Must have a return type that is identical to the method it overrides
- Cannot be less accessible than the method it overrides
- Must throw exceptions that are same type as the method being overridden

## *Invoking Overridden Methods*

### *Rules About Overridden Methods*

Remember that the method name, and order of arguments of a child method must be identical to those of the method in the parent class for that method to override the parent's version. The following rules apply to overridden methods:

- The return type of the overriding method must be identical to the method it overrides.
- An overriding method cannot be less accessible than the method it overrides.
- An overriding method cannot throw different types of exceptions than the method it overrides. (Exceptions will be discussed in the next module.)

## Invoking Overridden Methods

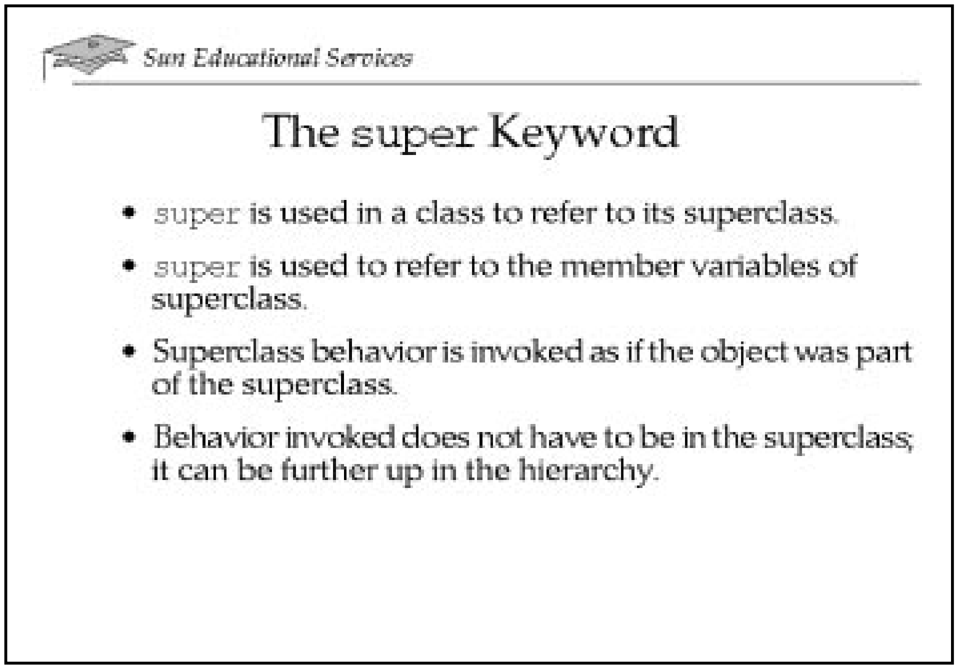
### Rules About Overridden Methods (Continued)

These rules result from the nature of polymorphism combined with the need for the Java programming language to be "typesafe." Consider this invalid scenario:

```
public class Parent {
  public void method() {
  }
}

public class Child extends Parent {
  private void method() {
  }
}

public class UseBoth {
  public void otherMethod() {
    Parent p1 = new Parent();
    Parent p2 = new Child();
    p1.method();
    p2.method();
  }
}
```

The Java programming language semantics dictate that `p2.method()` results in `Child` version of method being executed, but since the method is declared `private`, `p2` (declared as `Parent`) cannot access it. Thus, the language semantics are violated.

✓ ***This code fails at compile time, in spite of the fact that true type resolution takes effect at runtime.***


Sun Educational Services

The super Keyword

- `super` is used in a class to refer to its superclass.
- `super` is used to refer to the member variables of superclass.
- Superclass behavior is invoked as if the object was part of the superclass.
- Behavior invoked does not have to be in the superclass; it can be further up in the hierarchy.

## Subclassing

### *The* `super` *Keyword*

The `super` keyword refers to the superclass of the class in which the keyword is used. It is used to refer to the member variables or the methods of the superclass.

Quite often when you override a method, your real goal is not to replace the existing behavior but to extend that behavior in some way.

## Subclassing

### The `super` Keyword (Continued)

This can be achieved using the keyword `super` as follows:

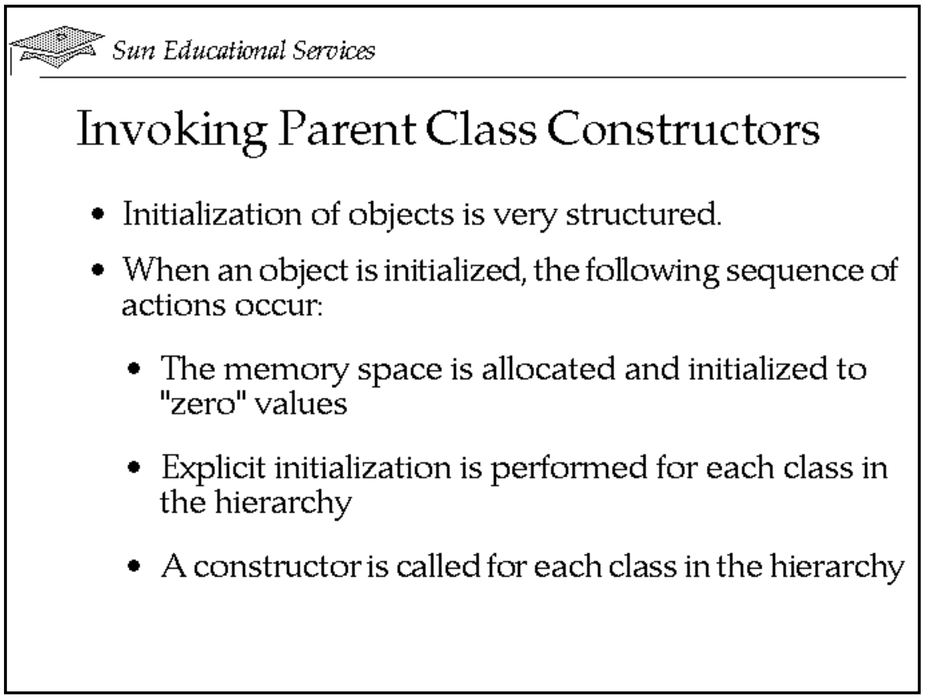
```
public class Employee {
  private String name;
  private int salary;

  public String getDetails() {
    return "Name: " + name + "\nSalary: " + salary;
  }
}

public class Manager extends Employee {
  private String department;

  public String getDetails() {
    // call parent method
    return super.getDetails() +
           "\nDepartment: " + department;
  }
}
```

Notice that a call of the form `super.method()` will invoke the entire behavior, along with any side effects, of the method that would have been invoked if the object had been of the parent class type. The method does not have to be defined in that parent class; it could be inherited from some class even further up the hierarchy.

---

**Note –** In the previous example, member variables have been declared as `private`. This is not necessary but is generally good programming practice. This topic and others related to the `private` keyword will be discussed later in this course.

---



Sun Educational Services

## Invoking Parent Class Constructors

- Initialization of objects is very structured.
- When an object is initialized, the following sequence of actions occur:
  - The memory space is allocated and initialized to "zero" values
  - Explicit initialization is performed for each class in the hierarchy
  - A constructor is called for each class in the hierarchy

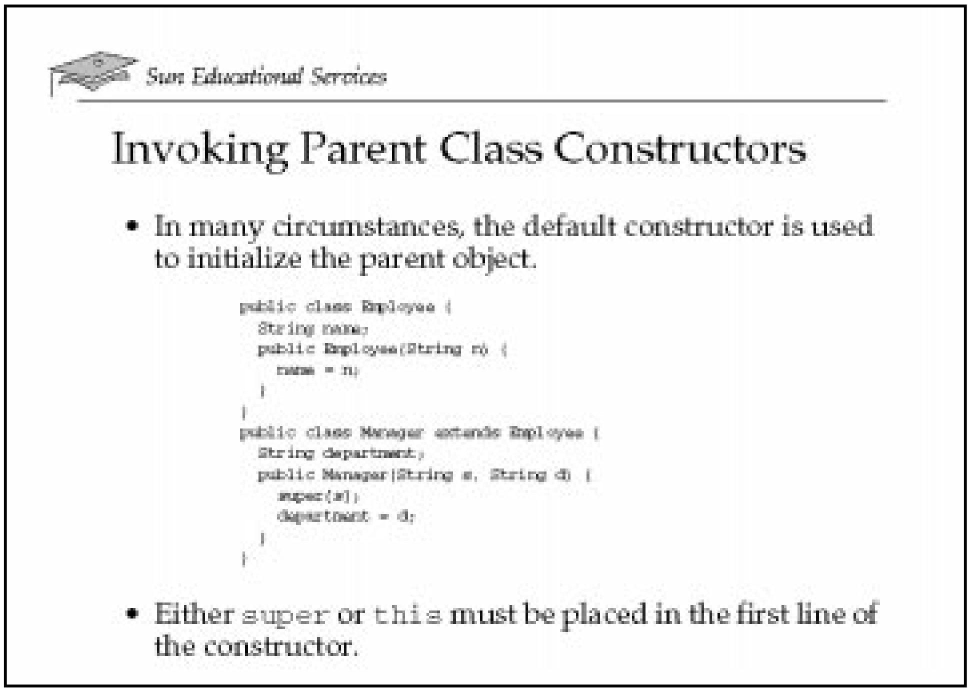
## *Invoking Parent Class Constructors*

The initialization of objects is very structured in the Java programming language. This is to ensure security. In the previous module, you saw what happens when an instance of a particular object is created. With inheritance, the picture is completed and the following sequence of actions occur:

- The memory space is allocated and initialized to zero.
- Explicit initialization is performed.
- A constructor is called.

The last two steps occur for each class in the hierarchy, starting at the top.

The Java technology security model demands that the aspects of an object that describe the parent class be initialized fully before the child class executes anything. Therefore, the Java programming language always calls a version of a parent constructor before executing the child constructor.



Sun Educational Services

## Invoking Parent Class Constructors

- In many circumstances, the default constructor is used to initialize the parent object.

```
public class Employee {
  String name;
  public Employee(String n) {
    name = n;
  }
}
public class Manager extends Employee {
  String department;
  public Manager(String s, String d) {
    super(s);
    department = d;
  }
}
```

- Either `super` or `this` must be placed in the first line of the constructor.

## *Invoking Parent Class Constructors*

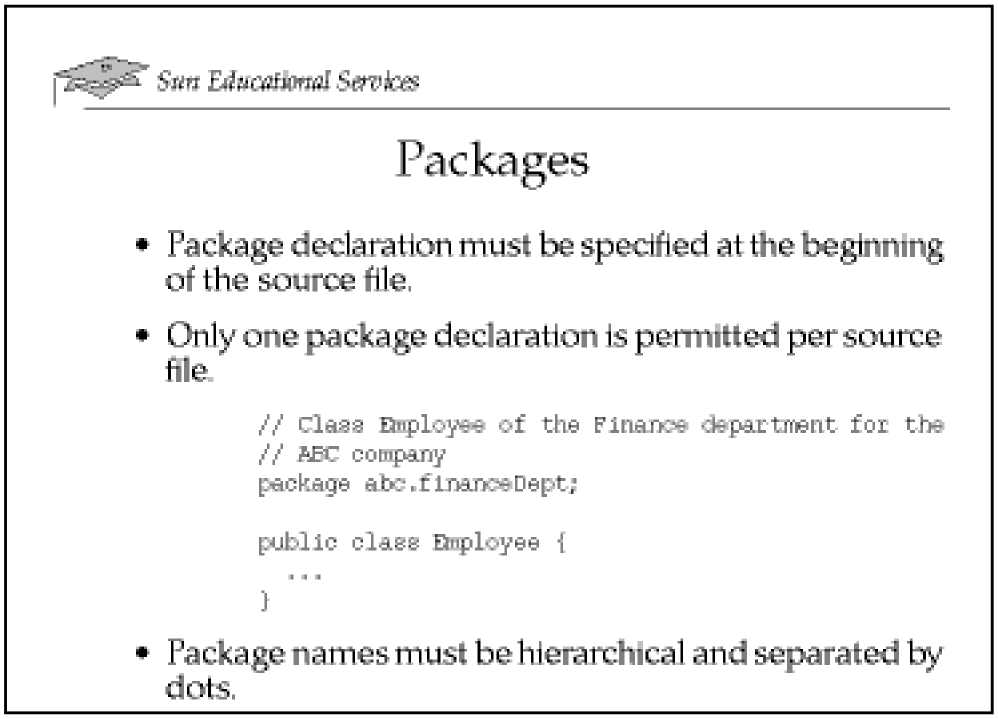
Often you will define a constructor that takes arguments and you will want to use those arguments to control the construction of the parent part of an object. You can invoke a particular parent class constructor as part of a child class initialization by “calling” the keyword `super` from the child constructor’s *first* line. To control the invocation of the specific constructor, you must provide the appropriate arguments to `super()`. When there is no call to `super` with arguments, the default parent constructor (that is, the constructor with zero arguments) is called implicitly. In this case, if there is no default parent constructor, a compiler error will result. For example:

## Invoking Parent Class Constructors

```
public class Employee {
  String name;
  public Employee(String n) {
    name = n;
  }
}

public class Manager extends Employee {
  String department;
  public Manager(String s, String d) {
    super(s);// Call parent constructor
    // with String argument
    department = d;
  }
}
```

**Note –** The call to `super()` can take any number of arguments appropriate to the various constructors available in the parent class, but it must be the first statement in the constructor.

When used, `super` or `this` must be placed in the first line of the constructor. If you write a constructor that has neither a call to `super(...)` nor `this(...)`, the compiler automatically inserts a call to the parent class constructor with no arguments. Other constructors can also call `super(...)` or `this(...)`, invoking a chain of constructors. What ultimately ends up happening is the parent class constructor (or possibly several) will execute before any child class constructor in the chain.


Sun Educational Services

## Packages

- Package declaration must be specified at the beginning of the source file.
- Only one package declaration is permitted per source file.

```
// Class Employee of the Finance department for the
// ABC company
package abc.financeDept;

public class Employee {
  ...
}
```

- Package names must be hierarchical and separated by dots.

## *Grouping Classes*

### *Packages*

The Java programming language provides the `package` mechanism as a way to group related classes. So far, all of these examples have belonged to the default or unnamed package.

You can indicate that classes in a source file belong to a particular package by using the `package` statement. For example:

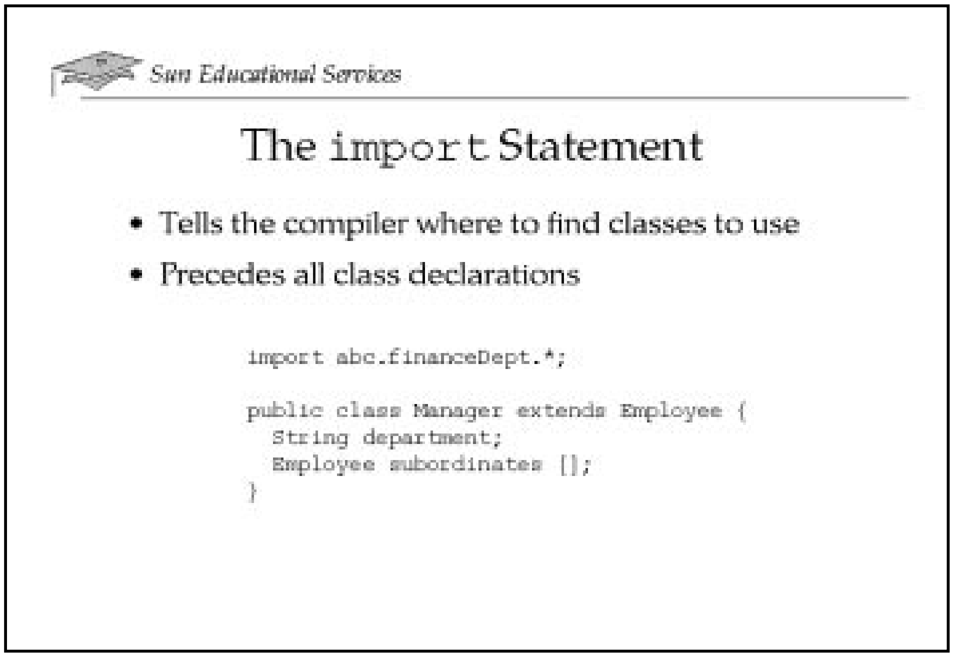
```
// Class Employee of the Finance department for the
// ABC company
package abc.financeDept;

public class Employee {
  ...
}
```

## Grouping Classes

### Packages (Continued)

The package declaration, if any, must be at the beginning of the source file. You can precede it with whitespace and comments, but nothing else. Only one package declaration is permitted and it governs the entire source file.

Package names are hierarchical, separated by dots. It is usual for the elements of the package name to be entirely lowercase. The classname, however, usually starts with a capital letter and the first letter of each additional word can be capitalized to distinguish words in the classname.


Sun Educational Services

The import Statement

- Tells the compiler where to find classes to use
- Precedes all class declarations

```
import abc.financeDept.*;

public class Manager extends Employee {
  String department;
  Employee subordinates [];
}
```

## Grouping Classes

### *The* `import` *Statement*

When you want to use packages, you use the `import` statement to tell the compiler where to find the classes. In fact, the package name (for instance, `abc.financeDept`) forms part of the name of the classes within the package. You could refer to the `Employee` class as `abc.financeDept.Employee` throughout, or you could use the `import` statement and just the class name `Employee`. For example:

```
// Class Manager belongs to the default package.
import abc.financeDept.*;
```

✓ ***The * indicates that the program can locate any class that exists in the*** `abc.financeDept` ***package.***

```
public class Manager extends Employee {
  String department;
  Employee subordinates [];
}
```

## Grouping Classes

### The `import` Statement (Continued)

---

**Note –** The `import` statements must precede all class declarations.

---

When you use a package declaration you do not need to import the same package or any element of that package. Remember that the `import` statement is used to bring classes in other packages into the current namespace. The current package, whether explicit or implicit, is always part of the current namespace.

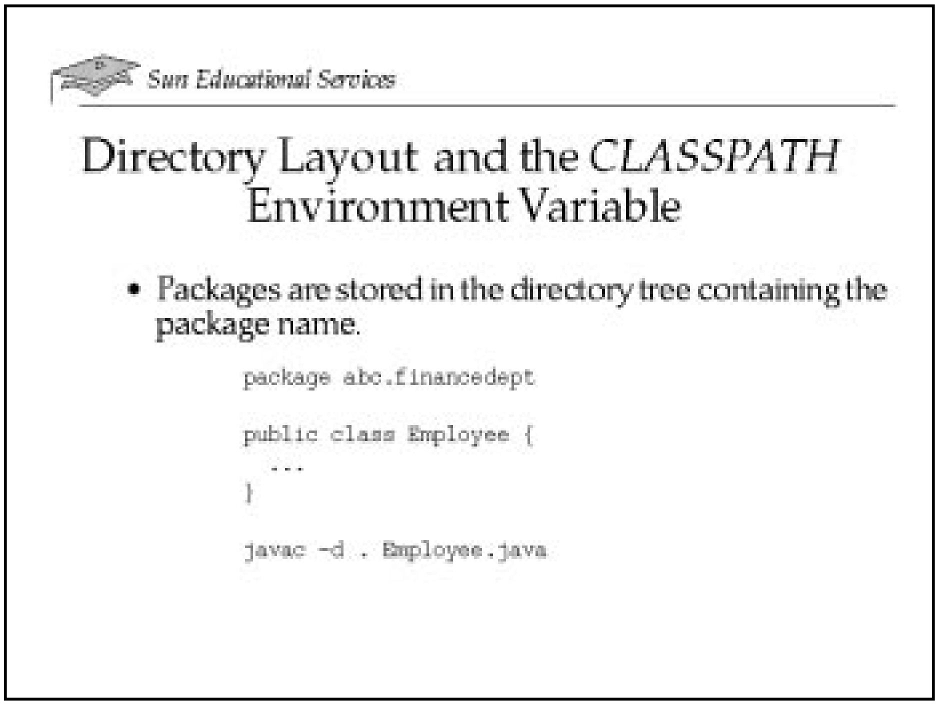
The `import` statement specifies which class you want access to. If you want access to all classes with a package, use "*". For example, to access all classes in the `java.awt` package use

```
import java.awt.*;
```

---

**Note –** The use of the `import` statement specifies a path for the compiler to find code, not actually load it as an #`include` statement would do in C or C++. Use of the `import` statement with "*" does not affect performance.

---



Sun Educational Services

## Directory Layout and the *CLASSPATH* Environment Variable

- Packages are stored in the directory tree containing the package name.

```
package abc.financedept

public class Employee {
  ...
}

javac -d . Employee.java
```

## *Grouping Classes*

### *Directory Layout and the* `CLASSPATH` *Environment Variable*

Packages are "stored" in a directory tree containing a branch which is the package name. For example, the `Employee.class` file from the previous page should exist in the following directory:

```
path/abc/financeDept
```

The roots of package trees to search for class files are in the `CLASSPATH`.

The `-d` option to the compiler specifies the root of a package hierarchy into which class files are placed (*`path`* shown previously).

## Grouping Classes

### Directory Layout and the CLASSPATH Environment Variable

The Java technology compiler creates the package directory and moves the compiled class file to it when the -d option is used. From the command line, typing

```
javac -d . Employee.java
```

will create the directory structure

```
abc/financeDept
```

in the current directory (".").

The CLASSPATH variable has not been used before because if it is not set, the default behavior of the tools automatically includes the standard location of the distribution classes and the current working directory. If you want to access packages that are located in other places, you must set the CLASSPATH variables to explicitly override default behavior.

For Solaris use

```
% javac -d /home/anton/mypackages Employee.java
```

For Windows use

```
C:> javac -d \home\anton\mypackages Employee.java
```

## Grouping Classes

### Directory Layout and the CLASSPATH Environment Variable

In order for the compiler to locate the `abc.financeDept.Employee` class while compiling the `Manager.java` file on the previous page, the `CLASSPATH` environment variable must include the following package path:

For Solaris use

```
CLASSPATH=/home/anton/mypackages:.
```

For Windows use

```
set CLASSPATH=\home\anton\mypackages;.
```

✓ ***As of JDK 1.2,*** `CLASSPATH` ***is no longer required; .***`jar` ***files can be placed in the*** `jdk1.2/jre/lib/ext` ***directory and be discovered automatically. If you have .***`class` ***files, make a classes directory under*** `jdk1.2/jre`***, and store your .***`class` ***files in that directory.***

## Exercise: Using Objects and Classes

**Exercise objective –** You will write, compile, and run three programs that use the object-oriented concepts of inheritance, constructors, and data hiding by modeling a bank account.

### Preparation

In order to successfully complete this lab, you must understand the concepts of classes and objects.

### Tasks

#### *Level 1 Lab: Create a Bank Account*

Perform the following steps:

1. Create a class, `Account.java`, that defines a bank account. Determine what the account should do, what data it needs to store, and what methods you will need.

2. Create a package named *bank* to contain the class.

#### *Level 2 Lab: Modify Account Types*

Perform the following steps:

1. Modify the Level 1 lab so you can subclass the `Account` for the specifics of a `CheckingAccount` class.

2. Allow the checking accounts to provide overdraft protection.

## *Exercise: Using Objects and Classes*

### *Tasks (Continued)*

#### *Level 3 Lab: Create an On-line Account Service*

Complete the following step:

1. Create a simple application, `Teller.java`, that uses the Level 1 or Level 2 lab to provide an on-line account-opening service.

# Exercise: Using Objects and Classes

## Exercise Summary

**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Define *encapsulation*, *polymorphism*, and *inheritance*
- ❑ Use the access modifiers `private` and `public`
- ❑ Develop a program segment to create and initialize an object
- ❑ Invoke a method on a particular object
- ❑ Describe constructor and method overloading
- ❑ Describe what the `this` reference is used for
- ❑ Discuss why Java application code is reusable
- ❑ In a Java software program, identify the following:
    - ▼ The `package` statement
    - ▼ The `import` statement
    - ▼ Classes, member functions, and variables
    - ▼ Constructors
    - ▼ Overloaded methods
    - ▼ Overridden methods
    - ▼ Parent class constructors

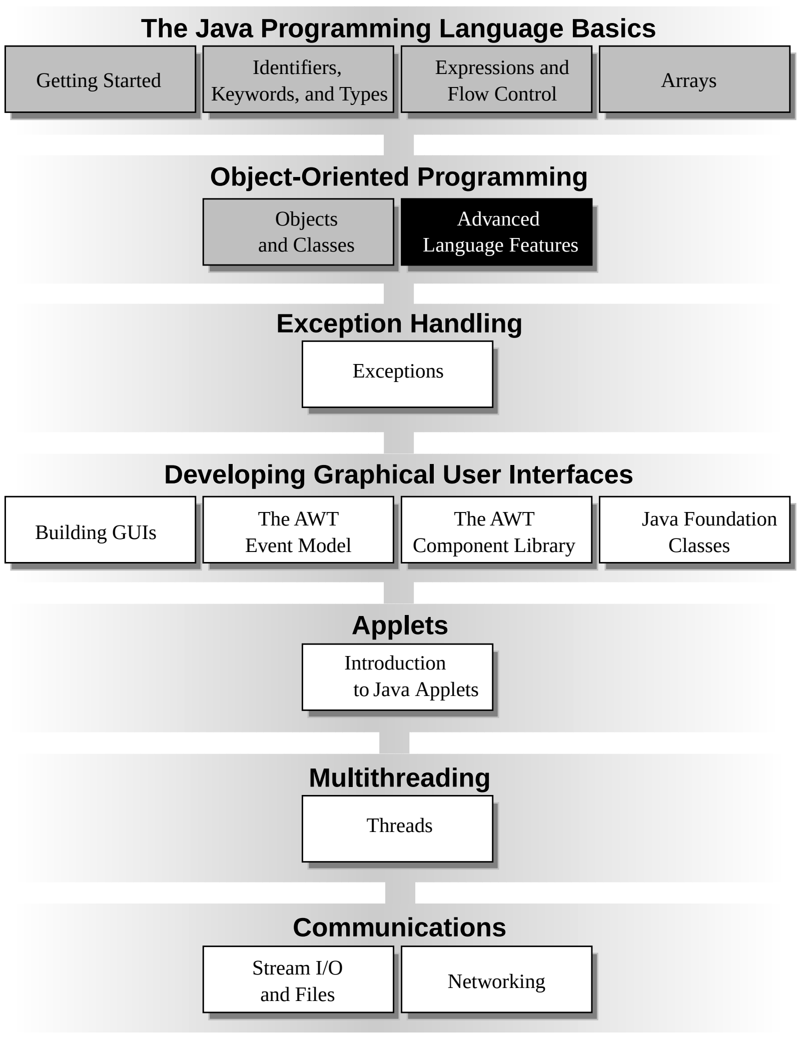
## Think Beyond

Now that you understand objects and classes, how can you use this information on a current or future project?

# Advanced Language Features 6

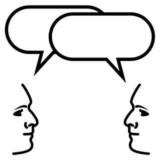
## Course Map

This module discusses more of the object-oriented features of the Java programming language.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following questions are relevant to the material presented in this module:

- How can you keep a class or method from being subclassed or overridden?
- How can you extend the use of array concepts to objects?

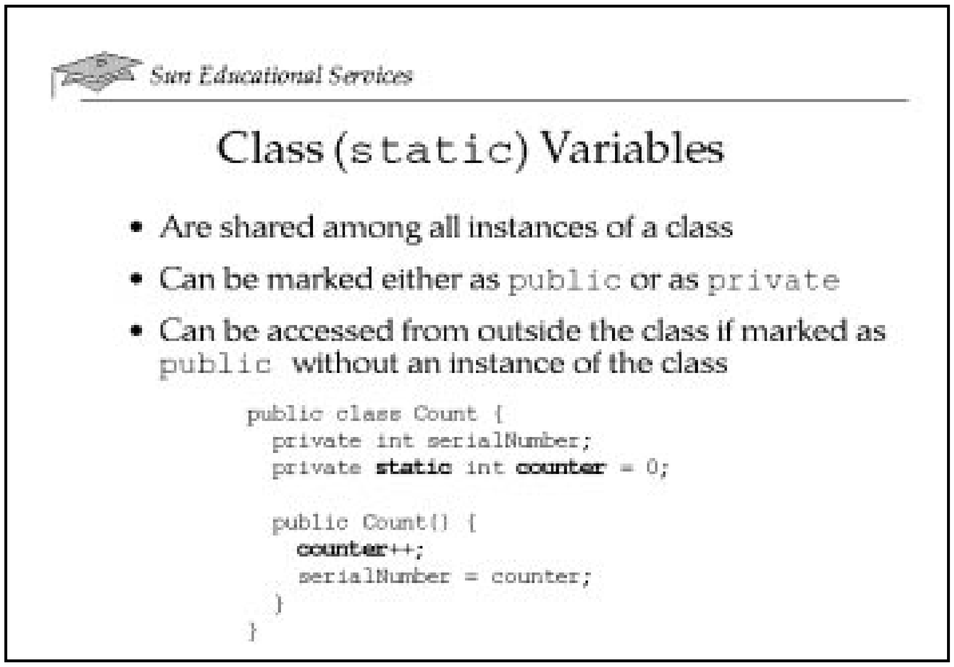
✓ ***A programmer needs to know all of the object-oriented features of the Java programming language in order to fully use the language's object-oriented power. This module discusses those remaining features of the language, picking up where the last module left off.***

✓ ***Topics in this module are more advanced, and include the functional relationship details between parent and child classes (construction, for example), plus discussion of wrapper classes; inner classes; levels of access protection; and the keywords*** `static`, `final`, ***and*** `abstract`. ***Knowledge of these language features greatly aid in the implementation of well-written Java software programs.***

## Objectives

Upon completion of this module, you should be able to

- Describe `static` variables, methods, and initializers
- Describe `final` classes, methods and variables
- List the access control levels
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to JDK 1.2
- Describe how to apply collections and reflections
- In a Java software program, identify
    - `static` methods and variables
    - `public`, `private`, `protected`, and default variables
- Use `abstract` classes and methods
- Explain how and when inner classes are used
- Explain how and when `interface` is used
- Describe the difference between `==` and `equals()`
- Define a `Vector`
- Describe wrapper classes



Sun Educational Services

### Class (`static`) Variables

- Are shared among all instances of a class
- Can be marked either as `public` or as `private`
- Can be accessed from outside the class if marked as `public` without an instance of the class

```
public class Count {
  private int serialNumber;
  private static int counter = 0;

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```

## *Class (`static`) Variables*

Sometimes it is desirable to have a variable that is shared among all instances of a class. For example, this can be used as the basis for communication between instances or to keep track of the number of instances that have been created.

You can achieve this effect by marking the variable with the keyword `static`. Such a variable is sometimes called a *class variable* to distinguish it from a member or instance variable which is not shared.

```
public class Count {
  private int serialNumber;
  private static int counter = 0;

  public Count() {
    counter++;
    serialNumber = counter;
  }
}
```

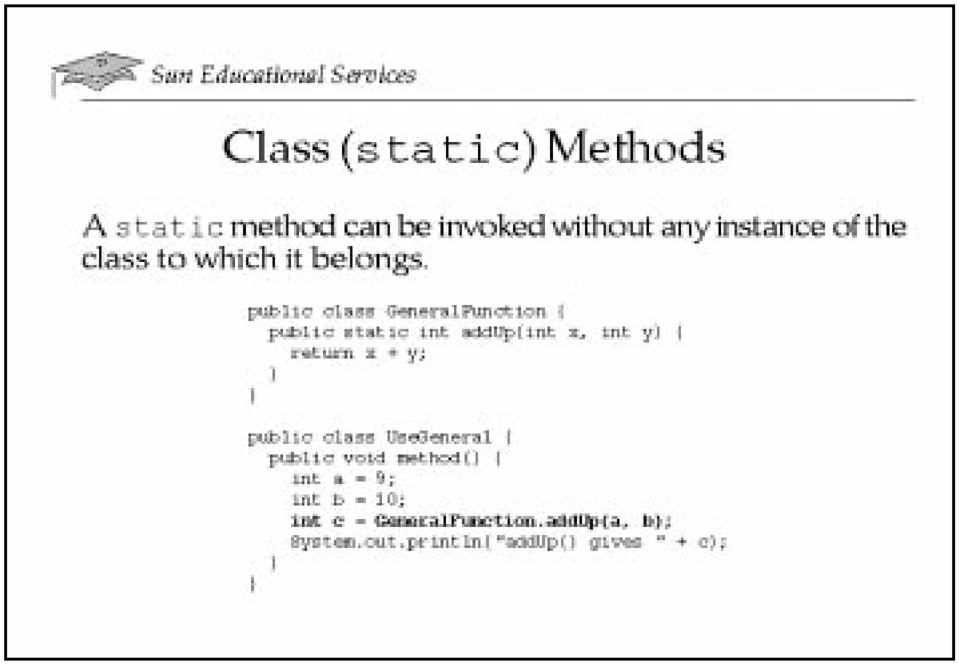
## Class (static) Variables

In this example, every object that is created is assigned a unique serial number, starting at 1 and counting upwards. The variable counter is shared among all instances, so when the constructor of one object increments counter, the next object to be created receives the incremented value.

A static variable is similar in some ways to a global variable in other languages. The Java programming language does not have globals as such, but a static variable is a single variable accessible from any instance of the class.

If a static variable is not marked as private, it can be accessed from outside the class. To do this, you do not need an instance of the class, you can refer to it through the class name.

```
public class StaticVar {
  public static int number;
}

public class OtherClass [
  public void method() {
    int x = StaticVar.number;
  }
}
```


Sun Educational Services

Class (static) Methods

A static method can be invoked without any instance of the class to which it belongs.

```
public class GeneralFunction {
  public static int addUp(int x, int y) {
    return x + y;
  }
}

public class UseGeneral {
  public void method() {
    int a = 9;
    int b = 10;
    int c = GeneralFunction.addUp(a, b);
    System.out.println("addUp() gives " + c);
  }
}
```

## *Class (*`static`*) Methods*

Sometimes you need to access program code when you do not have an instance of a particular object available. A method that is marked using the keyword `static` can be used in this way and is sometimes called a *class method*. Methods that are static can be accessed using the class name rather than a reference, as follows:

```
public class GeneralFunction {
  public static int addUp(int x, int y) {
    return x + y;
  }
}

public class UseGeneral {
  public void method() {
    int a = 9;
    int b = 10;
    int c = GeneralFunction.addUp(a, b);
    System.out.println("addUp() gives " + c);
  }
}
```

## *Class (`static`) Methods*

Because a `static` method can be invoked without any instance of the class to which it belongs, there is no `this` value. The consequence is that a `static` method cannot access any variables apart from its own arguments and `static` variables. Attempting to access non-static variables will cause a compiler error.

---

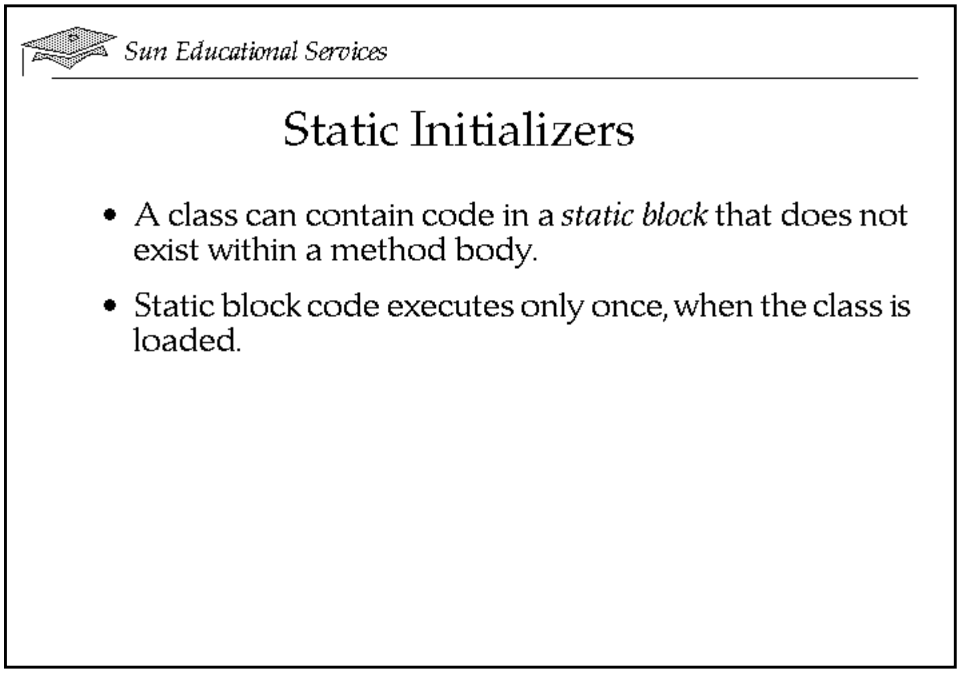
**Note –** Non-`static` variables are bound to an instance and can be accessed only through instance references.

---

```
public class Wrong {
  int x;

  public static void main(String args[]) {
    x = 9;// COMPILER ERROR!
  }
}
```

Important points to remember about `static` methods are:

- `main()` is `static` because it has to be accessible in order for an application to be run, before any instantiation can take place.
- When `main()` begins, no objects are created, so if you have member data, you must create an object to access it.
- A `static` method cannot be overridden to be non-static.



Sun Educational Services

### Static Initializers

- A class can contain code in a *static block* that does not exist within a method body.
- Static block code executes only once, when the class is loaded.

## *Static Initializers*

It is perfectly valid for a class to contain code in a “static block” that does not exist within a method body. The static block code executes only once, when the class is loaded. Different static blocks within a class are executed in the order of their appearance in the class.

```
public class StaticInitDemo {
  static int i = 5;

  static {
    System.out.println("Static code i= "+ i++ );
  }
}

public class Test {

  public static void main(String args[]) {
    System.out.println("Main code: i= "+ StaticInitDemo.i);
  }
}
```

## Static Initializers

The result will be

```
Static code i=5
Main code: i=6
```

# `Static` Methods and Data

## Object-Oriented Data and Methods

What does object-oriented really mean? A system as modeled by an object-oriented computer program consists of objects. Objects are created from a template, called a class. Attributes are defined by variables and behaviors are defined by methods. While all objects of the same class inherit the same characteristics, each object is different than the next.

For example, suppose a program which requires `Car` objects and concerns itself only with the `model` and `color` of a `Car` needs to be written.

Such a `Car` can be modeled as follows:

```
public class Car {
  String color;
  String model;

  public Car (String color, String model) {
    this.color = color;
    this.model = model;
  }

  public void whoAmI() {
    System.out.println (
        "I am a " + color + model + ".");
    }
}
```

All `Car` objects created of this class will have a `color` and a `model`, but each `Car` object is separate from the others and can have a different `color` or `model` than the others. Suppose there are two `Car` objects

```
Car JanesCar = new Car("Red","Coupe");
Car BobsCar = new Car("Blue","Hatchback");
```

## `Static` Methods and Data

### Object-Oriented Data and Methods (Continued)

The object-oriented paradigm is illustrated by the fact that both objects have separate data even though both are `Car`s. It is said that the data is *bound to their instance*. You cannot refer to this data without refering either implicitly, using `this,` or explicitly, using an instance.

Objects have methods which, even though they consist of the same code, will behave differently for various objects. This is because the data with which they operate is different for various objects. For example,

`JanesCar.whoAmI()` will print `"I am a Red Coupe."`
`JoesCar.whoAmI()` will print `"I am a Blue Hatchback."`

If one of the `Car`s gets painted, meaning the `color` attribute is changed, the other `Car` object will be unaffected. The scope of one object's data is that object itself. Changes to that object will not affect the data of any other object.

### Shared (`static`) Data and Methods

Java technology provides a means of sharing data between objects, on a class scope basis. Such data will not be bound to an instance nor stored in one, but to a class. Such methods and data are declared as *static*. No instances are needed to use `static` methods or data of a class, although they can be accessed through them.

✓ ***There may be a temptation to refer to `static` variables as "global" in nature by C and C++ programmers. The behavior is similar, but not really global because they are still bound to a class. If asked, mention the limitations of using the term global.***

## `Static` Methods and Data

### Shared (`static`) Data and Methods (Continued)

Suppose the class is changed to include a `static` variable called `nextSerialNumber` and the constructor is changed to refer to it in getting a serial number for its particular object.

```
public class Car {
  String color;
  String model;

  // Specific to this instance.
  int serialNumber;

  // Accessible by all instances.
  static int nextSerialNumber = 1;

  public Car (String color, String model) {
    this.color = color;
    this.model = model;
    serialNumber = nextSerialNumber++;
  }

  public void whoAmI() {
    System.out.println(
      "I am a " + color + " " + model +
      ", serial number = " + serialNumber);
  }

  public static void main (String args[]) {
    Car JanesCar = new Car("Red", "Coupe");
    Car JoesCar = new Car("Blue", "Hatchback");

    JanesCar.whoAmI();
    JoesCar.whoAmI();
  }
}
```

Running the program returns

```
I am a Red Coupe, serial number = 1
I am a Blue Hatchback, serial number = 2
```

## `Static` Methods and Data

### Shared (`static`) Data and Methods (Continued)

The integer `nextSerialNumber` will be shared among all `Car` instances, and the `Car` class itself. From within an object of the class, it is accessible as

```
nextSerialNumber
```

From outside an instance, it can be accessed as

```
Car.nextSerialNumber
```

The integer `nextSerialNumber` can also be accessed from any instance explicitly. Note how it differs from `serialNumber`, a nonstatic variable which is bound to its instance and not shared. Each `Car` object has a unique `serialNumber`. They all share access to the running serial number counter `nextSerialNumber` from which their `serialNumber` is initialized.

Note that while nonstatic methods can access `static` data and methods, the reverse is not true unless an instance is explicitly stated. In other words, a `static` method must call `JanesCar.whoAmI()`, not just `whoAmI()`. This is because `static` methods have no implicit instance (`this`) to refer to since they are not bound to any instance themselves. Thus they would not know which instance's nonstatic method to call or data to access. For example, does `color` refer to `JanesCar` or `JoesCar`? Using `JanesCar.color` makes it clear.

Sometimes it is helpful to declare a method as `static` because there may be no need for an instance to be created. For example, all methods in the `java.lang.Math` class are `static` because they get no data from within their class except for a few `static final` constants. Thus there is no need to instantiate them. Note that such methods get most of their data from arguments passed in. For example, consider a method `times2`.

```
public static int times2(int value) {
  return value * 2;
}
```

Methods such as this, which use only arguments or local data, `static` variables, and no instance data, can be declared `static`.

## `Static` Methods and Data

### Shared (`static`) Data and Methods (Continued)

In the following example, the `Car` class has a new `static` method called `getNextSerialNum()` which returns the `static` variable `nextSerialNumber`. The `Car` class already has the nonstatic method `whoAmI()` for accessing instance data such as `serialNumber`.

A method can also be declared `static` at times when it is impossible to instantiate. A `main()` method of a class is declared `static` because it has to execute before any instances can be created. By declaring it `static`, no instances need to be present before it can run.

Although the member and `static` variables have been marked `private`, they can now be accessed via methods. This demonstrates good object-oriented technique by encapsulating (hiding) the important data and providing access through a method.

## `Static` Methods and Data

### Shared (`static`) Data and Methods (Continued)

```
public class Car {
  private String color;
  private String model;

  // Specific to this instance.
  private int serialNumber;

  // Accessible by all instances.
  private static int nextSerialNumber = 1;

  public Car (String color, String model) {
    this.color = color;
    this.model = model;
    serialNumber = nextSerialNumber++;
  }

  public void whoAmI() {
    System.out.println(
      "I am a " + color + model +
      ", serial number = " + serialNumber);
  }

  public static void getNextSerialNum() {
    System.out.println(
        "The next available serial number is " +
        nextSerialNumber);
  }

  public static void main (String args[]) {
    Car JanesCar = new Car("Red", "Coupe");
    Car JoesCar = new Car("Blue", "Hatchback");

    // Use nonstatic method to get instance data
    JanesCar.whoAmI();
    JoesCar.whoAmI();

    // Use static method to get class data
    getNextSerialNum();
  }
}
```

## A Complete Example (Continued)

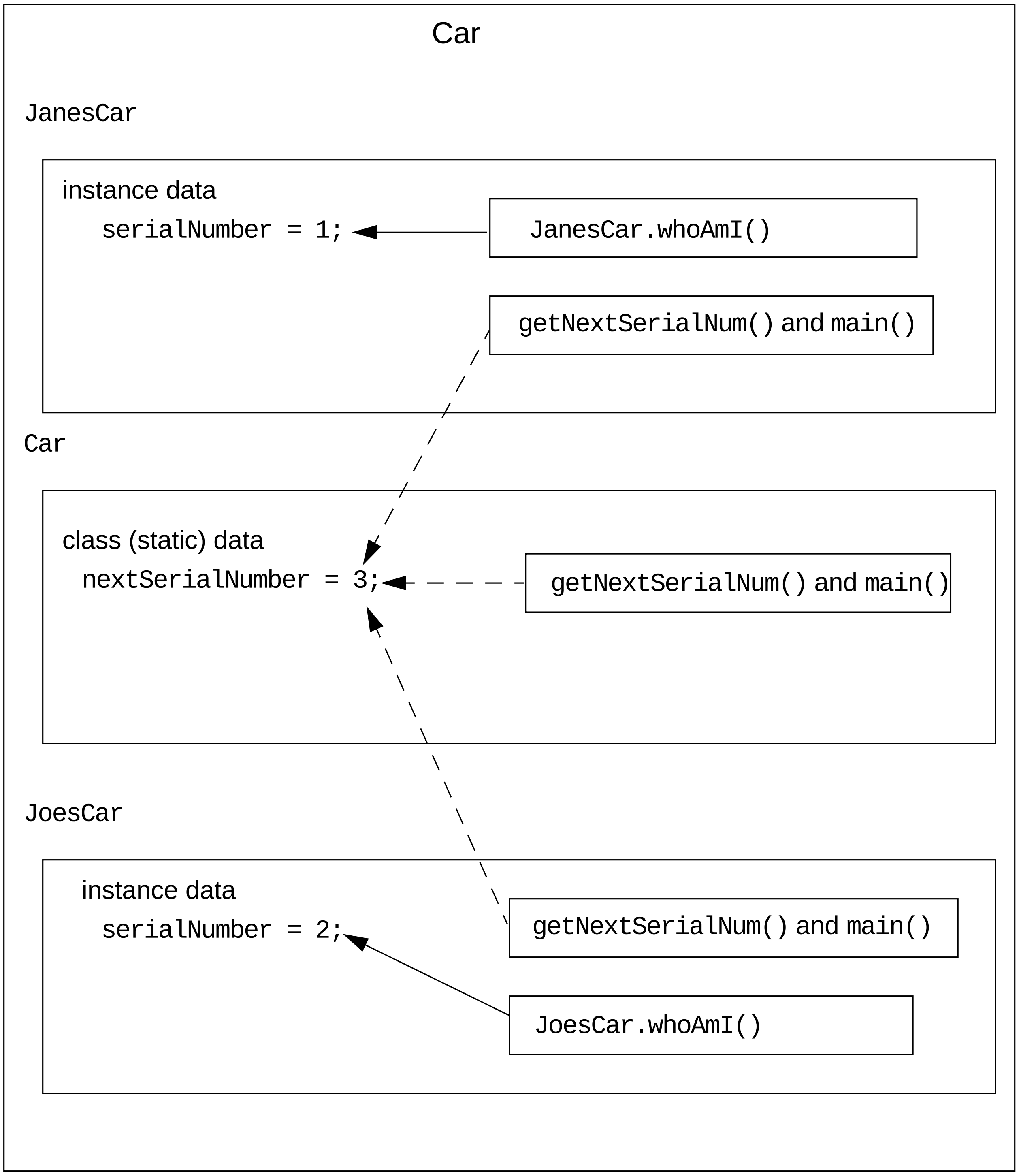
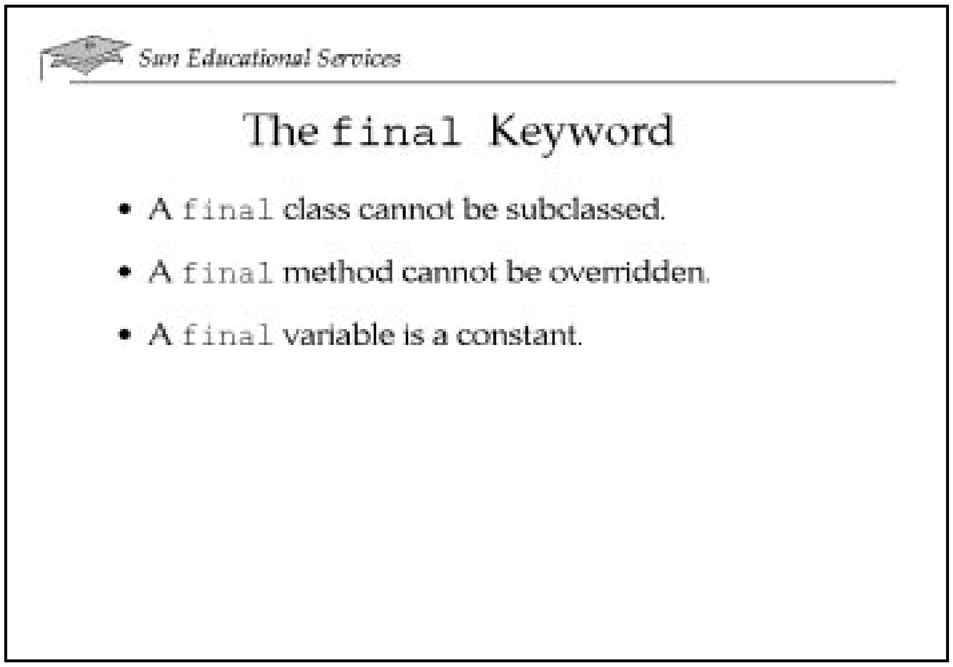


**Figure 6-1** Car Example


Sun Educational Services

## The `final` Keyword

- A `final` class cannot be subclassed.
- A `final` method cannot be overridden.
- A `final` variable is a constant.

## *The* `final` *Keyword*

### *Final Classes*

The Java programming language allows the keyword `final` to be applied to classes. If this is done, the class cannot be subclassed. The class `java.lang.String`, for example, is a `final` class. This is done for security reasons, since it ensures that if a method has a reference to a string, it is definitely a string of class `String` and not a string of a class that is a modified subclass of `String` that might have been maliciously changed.

## *The* `final` *Keyword*

### *Final Methods*

Individual methods can be marked as `final` also. Methods marked `final` cannot be overridden. This is done for security reasons. Make a method `final` if the method has an implementation that should not be changed and is critical to the consistent state of the object.

Methods declared `final` are sometimes used for optimization. The compiler can generate code that causes a direct call to the method, rather than the usual virtual method invocation that involves a runtime lookup.

Methods marked as `static` or `private` are `final` automatically since dynamic binding cannot be applied in either case.

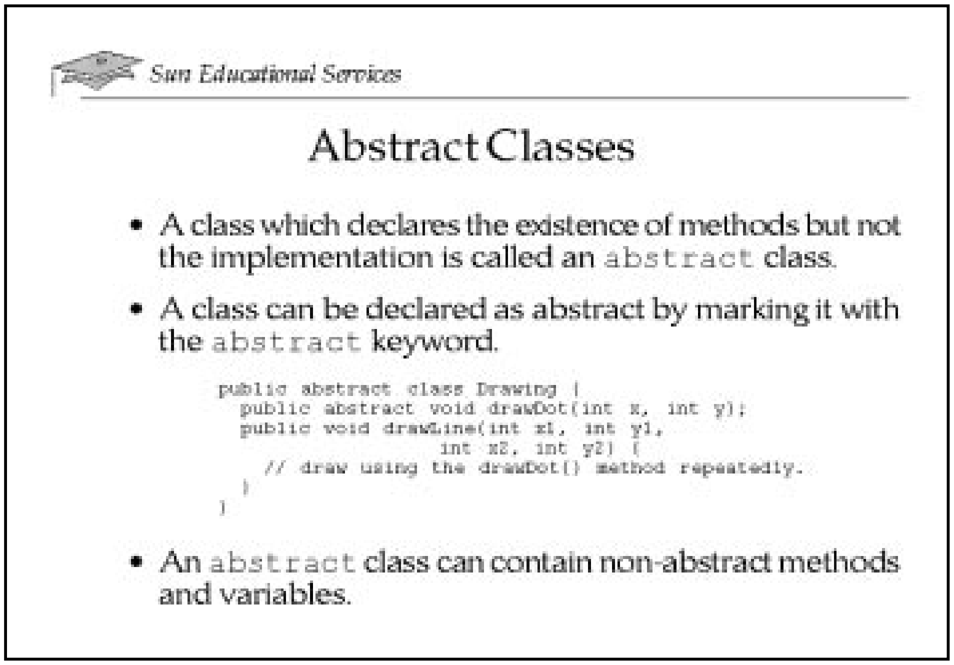
### *Final Variables*

If a variable is marked as `final`, the effect is to make it a constant. Any attempt to change the value of a `final` variable will cause a compiler error. The following example shows a properly defined `final` variable:

```
public final int MAX_ARRAY_SIZE = 25;
```

✓ ***Final variables are roughly equivalent to const in C and C++***

**Note –** If you mark a variable of reference type (that is, any class type) as `final`, that variable cannot refer to any other object. It is however possible to change the object’s contents, as only the reference itself is `final`.



Sun Educational Services

## Abstract Classes

- A class which declares the existence of methods but not the implementation is called an `abstract` class.
- A class can be declared as abstract by marking it with the `abstract` keyword.

```
public abstract class Drawing {
  public abstract void drawDot(int x, int y);
  public void drawLine(int x1, int y1,
                       int x2, int y2) {
    // draw using the drawDot() method repeatedly.
  }
}
```

- An `abstract` class can contain non-abstract methods and variables.

## *Abstract Classes*

Sometimes in library development you want to create a class that represents some fundamental behavior, and declare methods for that class, but you cannot implement the behavior in that class. Instead, you implement the methods in subclasses. Other methods, whose behavior you know, can be implemented in the class. Abstract classes can define member variables as well.

For example, consider a `Drawing` class. The class contains methods for a variety of drawing facilities, but these must be implemented in a platform-independent way. It is not possible to access the video hardware of a machine and still be platform independent. The intention is that the drawing class defines which methods should exist, but special platform-dependent subclasses will actually implement the behavior.

A class such as the `Drawing` class, which declares the existence of methods but not in the implementation, as well as the implementation of methods with known behavior, is commonly called an *abstract class*. Abstract Classes

## Abstract Classes

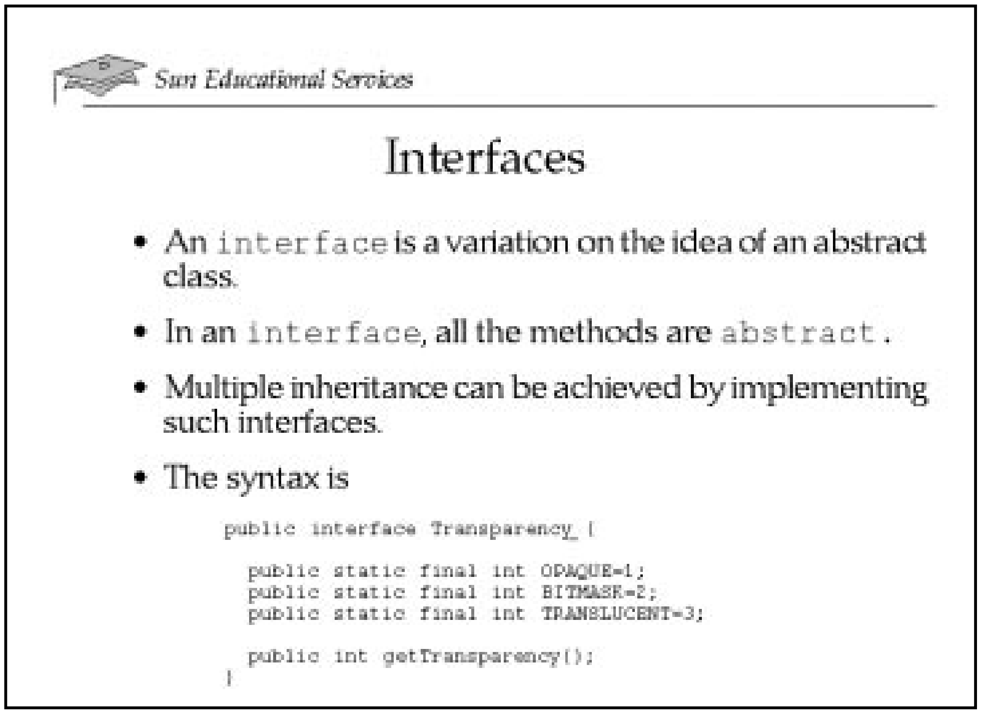
You can declare an abstract class by marking it with the `abstract` keyword. Methods which are declared but not implemented (that is, those without a body or { } ), must also be marked as abstract.

```
public abstract class Drawing {
  public abstract void drawDot(int x, int y);
  public void drawLine(int x1, int y1,
                       int x2, int y2) {
    // draw using the drawDot() method repeatedly.
  }
}
```

You cannot create an instance of an `abstract` class. You can, however, create a variable whose type is an abstract class and let it refer to an instance of a concrete subclass. You cannot have abstract constructors or abstract static methods.

Subclasses of `abstract` classes must provide implementations for all abstract methods in their parents, otherwise they too are abstract classes.

```
public class MachineDrawing extends Drawing {
  public void drawDot (int mach x, intmach y) {
    // Draw the dot
  }
}

Drawing d = new MachineDrawing();
```



Sun Educational Services

## Interfaces

- An `interface` is a variation on the idea of an abstract class.
- In an `interface`, all the methods are `abstract`.
- Multiple inheritance can be achieved by implementing such interfaces.
- The syntax is

```
public interface Transparency_ {

  public static final int OPAQUE=1;
  public static final int BITMASK=2;
  public static final int TRANSLUCENT=3;

  public int getTransparency();
}
```

## *Interfaces*

An *interface* is a variation on the idea of an `abstract` class. In an `interface` all the methods are `abstract`; none can have “bodies.” An `interface` can define only `static final` member variables.

The benefit of an `interface` is that it gives the illusion of bending the Java technology single inheritance rule. While a class definition can extend only a single class, it can implement as many interfaces as needed.

Implementing an `interface` is similar to subclassing, except that the implementing class cannot inherit behavior from the `interface` definition. When a class implements a particular interface, it defines (that is, gives bodies to) all of that interface’s methods. It is then possible to call methods of the `interface` on any object of a class that implements that `interface`.

As with `abstract` classes, it is permissible to use an `interface` name as a type of reference variable. The usual dynamic binding will take effect. References can be cast to and from interface types, and the `instanceof` operator can be used to determine if an object’s class implements an `interface`.

## Interfaces

Interfaces are defined with the keyword `interface` as follows:

```
public interface Transparency {

  public static final int OPAQUE=1;
  public static final int BITMASK=2;
  public static final int TRANSLUCENT=3;

  public int getTransparency();
}
```

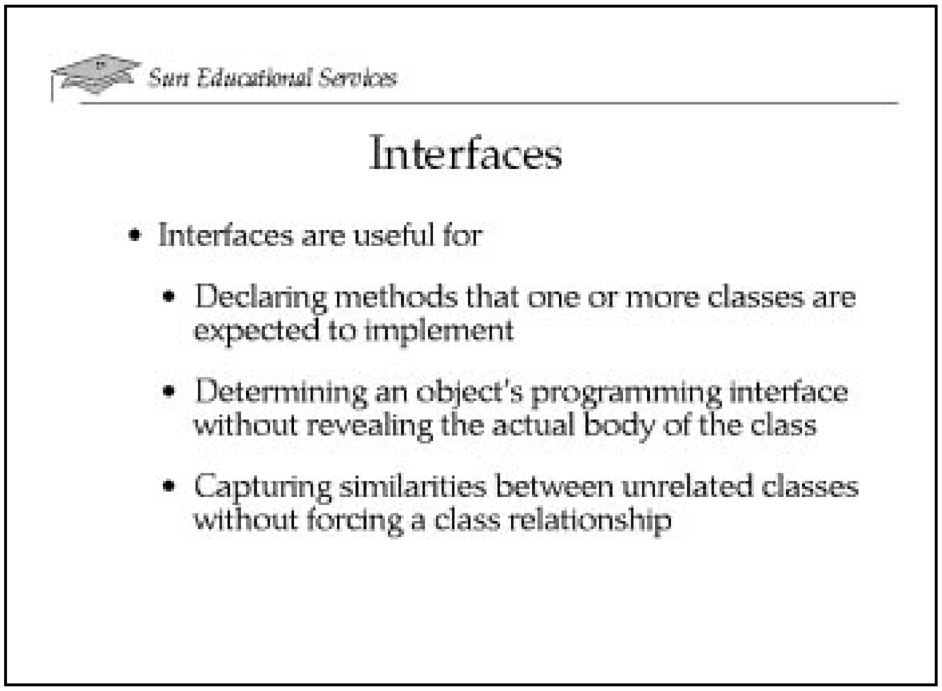
A class can implement multiple interfaces. Interfaces implemented by a class appear in a comma-separated list at the end of a class declaration as follows:

```
public class MyApplet extends Applet
            implements Runnable, MouseListener {
  "..."
}
```

The following example shows a simple interface and a class that implements it:

```
interface SayHello {
  void printMessage();
}

class SayHelloImpl implements SayHello {
  void printMessage() {
    System.out.println("Hello");
  }
}
```

The `interface SayHello` mandates that all classes which implement it must have a method called `printMessage()` with a `void` return type and no input parameters.



Sun Educational Services

## Interfaces

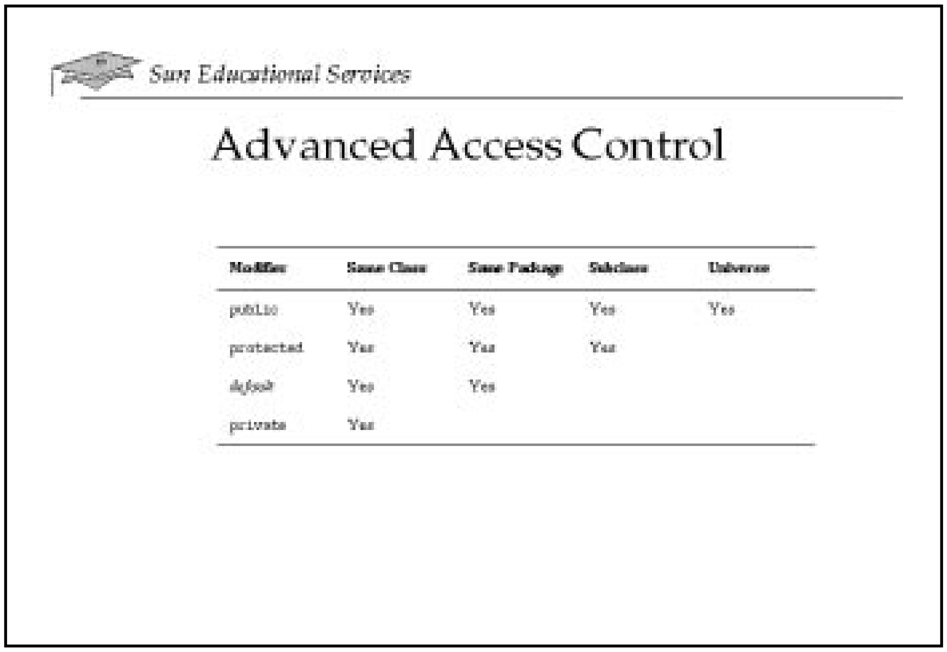
- Interfaces are useful for
  - Declaring methods that one or more classes are expected to implement
  - Determining an object's programming interface without revealing the actual body of the class
  - Capturing similarities between unrelated classes without forcing a class relationship

## *Interfaces*

Interfaces are useful for:

- Declaring methods that one or more classes are expected to implement.
- Revealing an object's programming interface without revealing the actual body of the class. (This can be useful when shipping a package of classes to other developers.)
- Capturing similarities between unrelated classes without forcing a class relationship.

✓ ***Interfaces are often looked at as an alternative to multiple inheritance, even though they provide a different functionality. The concept of interfaces was borrowed from Objective - C where they were called protocols.***



Sun Educational Services

## Advanced Access Control

| Modifier | Same Class | Same Package | Subclass | Universe |
|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | |
| *default* | Yes | Yes | | |
| private | Yes | | | |

## *Advanced Access Control*

Variables and methods can be at one of four access levels; `public`, `protected`, *default*, or `private`. Classes can be at `public` or *default* level.

✓ ***Inner classes can have `public`, `protected`, `default`, or `private` accessibility***

A variable, method, or class has default accessibility if it does not have an explicit protection modifier as part of its declaration. Such accessibility means that access is permitted from any method in classes that are members of the *same package* as the target.

A variable or method marked with the modifier `protected` is actually more accessible than one with default access control. A `protected` method or variable is accessible from methods in classes that are members of the same package and any method in any *subclass*. The protected access is to be used when it is appropriate for a class's subclass, but not unrelated classes, to have access to the member.

## Advanced Access Control

Table 6-1 summarizes the accessibility criteria.

**Table 6-1** Accessibility Criteria

| **Modifier** | **Same Class** | **Same Package** | **Subclass** | **Universe** |
|---|---|---|---|---|
| `public` | Yes | Yes | Yes | Yes |
| `protected` | Yes | Yes | Yes | |
| *default* | Yes | Yes | | |
| `private` | Yes | | | |

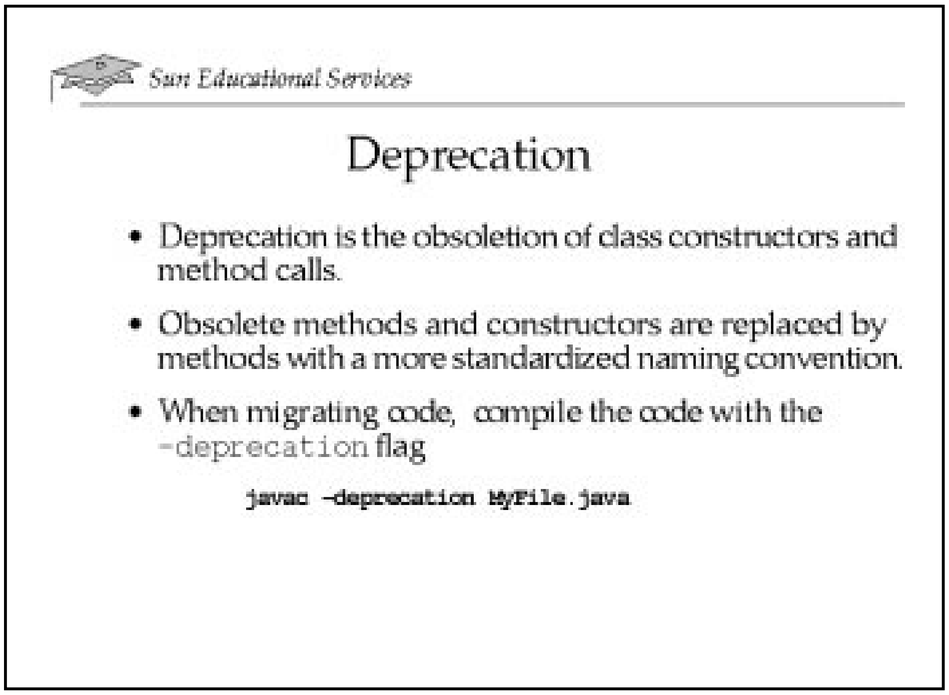
---

**Note –** Protected access is provided to subclasses that reside in a different package from the class that owns the protected feature.

---

✓ ***A design that makes extensive use of protected or default access elements is probably either very well designed or very poorly designed.***

✓ `protected` ***is particularly useful in JavaBeans™. Introspection only works on public methods;*** `protected` ***allows member variables to be accessible by subclasses and invisible to the introspector.***


Sun Educational Services

## Deprecation

- Deprecation is the obsoletion of class constructors and method calls.
- Obsolete methods and constructors are replaced by methods with a more standardized naming convention.
- When migrating code, compile the code with the `-deprecation` flag

```
javac -deprecation MyFile.java
```

## *Deprecation*

In JDK 1.1, a significant effort was made to standardize the names of methods. As a result, in JDK 1.2, a significant number of class constructors and method calls are obsolete. They have been replaced by method names that follow a more standardized naming convention and, in general, make life less complicated for the programmer.

For example, in the JDK 1.0 version of the `java.awt.Component` class:

- The methods for changing or getting the size of a component are `resize()` and `size()`.
- The methods for changing or getting the bounding box of a component are `reshape()` and `bounds()`.

## Deprecation

In the JDK 1.1 version of `java.awt.Component`

- Method names that begin with the words `set` and `get` to indicate the primary operation of the method, respectively. For example:
    - ▼ `setSize()` and `getSize()`
    - ▼ `setBounds()` and `getBounds()`

Whenever you are moving code from JDK 1.0 to JDK 1.1 or higher, or even if you are using code that previously worked with JDK 1.0, it is a good idea to compile the code with the `-deprecation` flag.

```
javac -deprecation MyFile.java
```

✓ ***Sometimes whole classes are changed between JDK 1.0 and JDK 1.1. Unfortunately, these were not deprecated, so the compiler issues an error message stating that the class cannot be found. This is annoying, but unavoidable.***

## Deprecation

The `-deprecation` flag will report any methods used within the class that are deprecated. For example, consider a utility class called `DateConverter` which converts a date in the format `mm/dd/yy` to the day of the week.

```
1  package myutilities;
2
3  import java.util.*;
4  import java.text.*;
5
6  public final class DateConverter {
7    private static String day_of_the_week [] =
8      {"Sunday", "Monday", "Tuesday", "Wednesday",
9      "Thursday", "Friday", "Saturday"};
10
11   public static String getDayOfWeek (String theDate){
12     int month, day, year;
13
14     StringTokenizer st = new StringTokenizer (theDate, "/");
15
16     month = Integer.parseInt(st.nextToken ());
17     day = Integer.parseInt(st.nextToken());
18     year = Integer.parseInt(st.nextToken());
19     Date d = new Date (year, month, day);
20
21     return (day_of_the_week[d.getDay()]);
22   }
23 }
```

When this code is compiled under JDK 1.2 with the `-deprecation` flag, you get

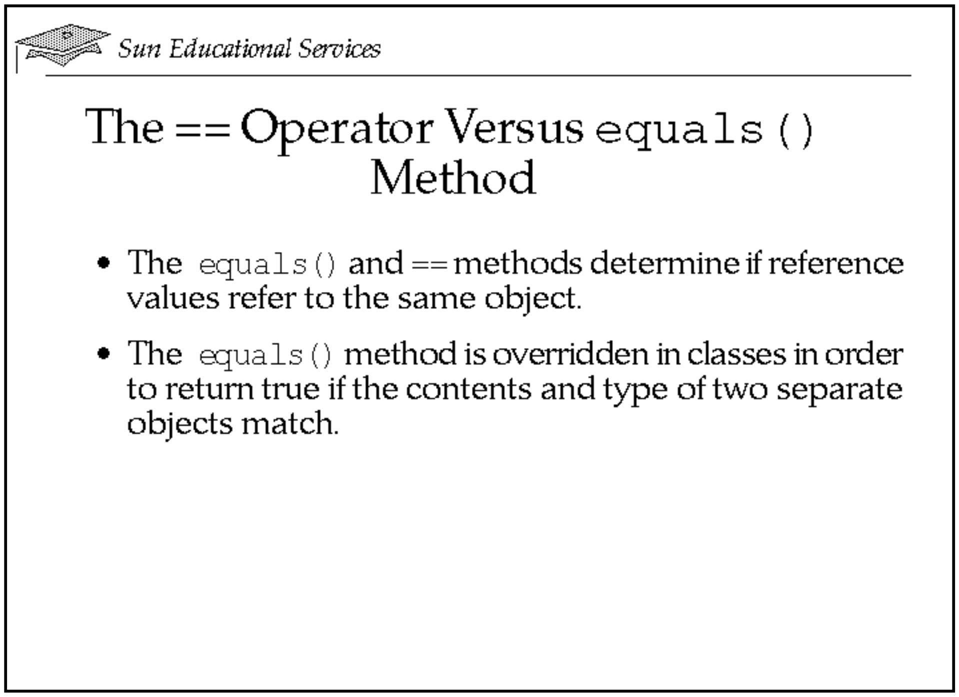
```
javac -deprecation DateConverter.java
DateConverter.java:19: Note: The constructor java.util.Date(int,int,int)
has been deprecated.
    Date d = new Date (year, month, day);
             ^
DateConverter.java:21: Note: The method int getDay() in class
java.util.Date has been deprecated.
    return (day_of_the_week[d.getDay()]);
                                   ^
Note: DateConverter.java uses or overrides a deprecated API.  Please
consult the documentation for a better alternative.   1 warning
```

## Deprecation

The rewritten DateConverter class looks like this:

```
package myutilities;

import java.util.*;
import java.text.*;

public final class DateConverter {
  private static String day_Of_The_Week[] =
      {"Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"};

  public static String getDayOfWeek (String theDate) {
    Date d = null;
    SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yy");

    try {
      d = sdf.parse (theDate);
    } catch (ParseException e) {
      System.out.println (e);
      e.printStackTrace();
    }

    // Create a GregorianCalendar object
    Calendar c =
        new GregorianCalendar(
            TimeZone.getTimeZone("EST"),Locale.US);
    c.setTime (d);

    return(
        day_Of_The_Week[(c.get(Calendar.DAY_OF_WEEK)-1)]);
  }
}
```

Here the 1.2 version uses two new classes: SimpleDateFormat, a class used to take any String date format and create a Date object, and the GregorianCalendar class, used to create a calendar with the local time zone and locale.


Sun Educational Services

The == Operator Versus `equals()` Method

- The `equals()` and == methods determine if reference values refer to the same object.
- The `equals()` method is overridden in classes in order to return true if the contents and type of two separate objects match.

## *The == Operator Versus* `equals()` *Method*

The == operator performs an equivalent comparison. That is, for any reference values `x` and `y`, `x==y` returns true if and only if `x` and `y` refer to the same object.

The `Object` class in the `java.lang` package has the method `public boolean equals(Object obj)`which compares two objects for equality. When not overridden, an object's `equals()` method returns `true` only if the two references being compared refer to the same object. However, the intention of the `equals()` method is to compare the contents of two objects whenever possible. This is why the method is frequently overridden. The classes `String`, `Date`, `File`, and all wrapper classes (`Integer`, `Double` and so on) amongst others *override* `equals()` to return true if the *contents* and *type* of two separate objects match.

For example, the `equals()` method in `String` class returns `true` if and only if the argument is not `null` and is a `String` object that represents the same sequence of characters as the `String` object with which the method is invoked.

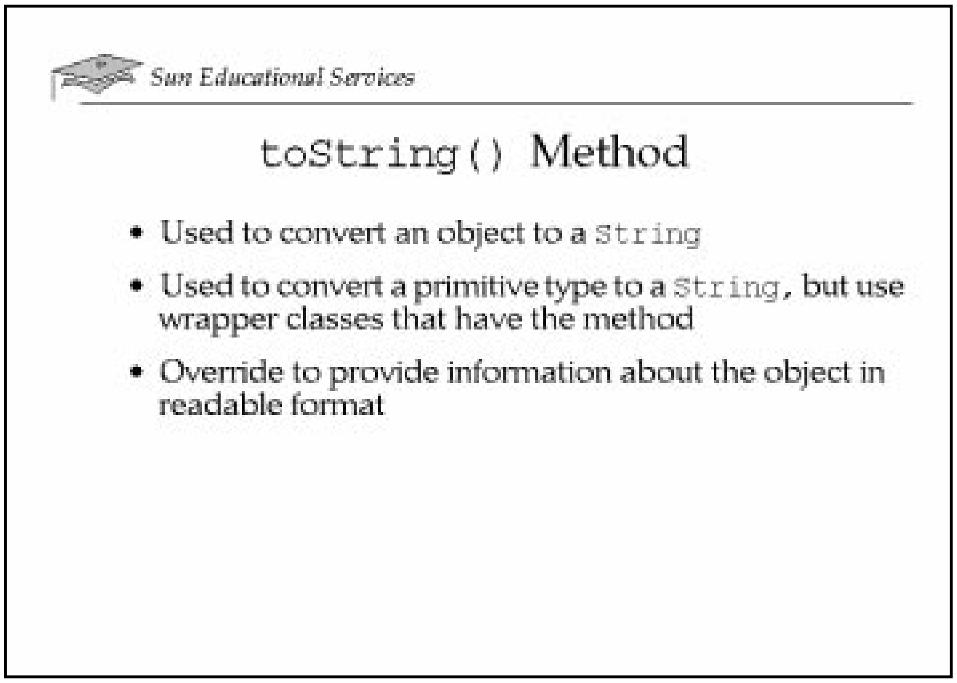
## The == Operator Versus `equals()` Method

### Example

```
class MyEquals {

  String objectString;

  // Constructor
  public MyEquals( String o ) {
    objectString = o;
  }

  public static void main (String args []) {

    // Test equals on a generic object which has no
    // overridden equals() method, so it only compares
    // reference values
    MyEquals me1 = new MyEquals("JDK1.2");
    MyEquals me2 = new MyEquals("JDK1.2");

    if( me1.equals(me2) ) {
      System.out.println("me1 IS equal to me2");
    } else {
      System.out.println("me1 NOT equal to me2");
    }

    // Test equals() on String, which overrides
    // the equals() method, which tests for content of the String
    String s1 = new String("JDK1.2");
    String s2 = new String("JDK1.2");

    if( s1.equals(s2) ) {
      System.out.println("s1 IS equal to s2");
    } else {
      System.out.println("s1 NOT equal to s2");
    }
  }
}
```

outputs

```
me1 NOT equal to me2
s1 IS equal to s2
```



Sun Educational Services

## toString() Method

- Used to convert an object to a String
- Used to convert a primitive type to a String, but use wrapper classes that have the method
- Override to provide information about the object in readable format

## *The* toString() *Method*

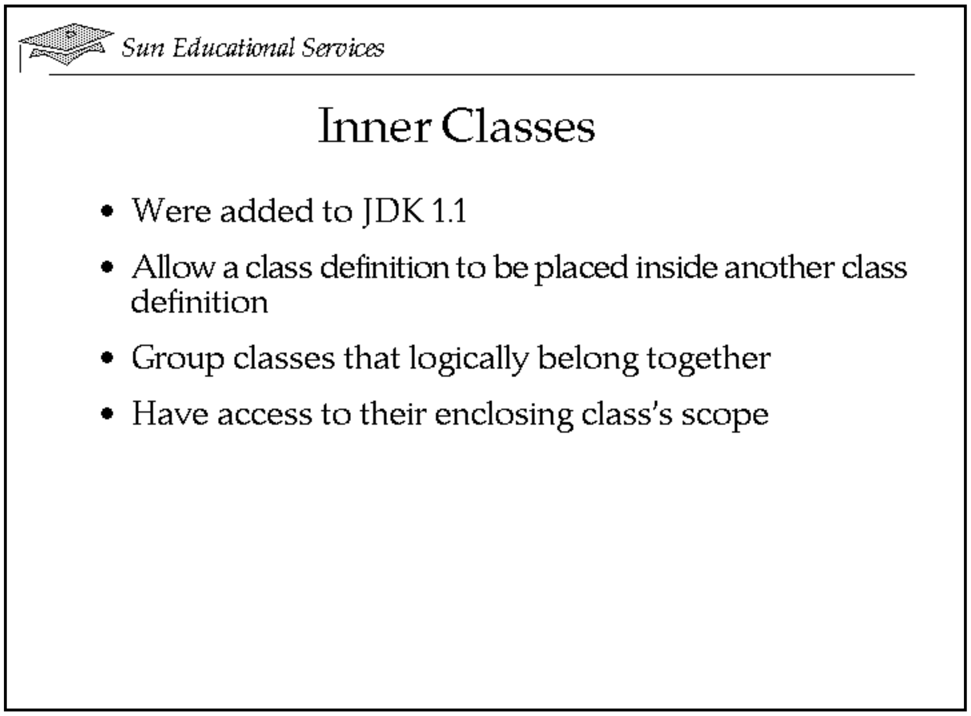
The toString() method is used to convert an object to a String representation. It is referenced by the compiler when automatic string conversion takes place. The System.out.println() call

```
Date now = new Date();
System.out.println(now);
```

will be translated into

```
System.out.println(now.toString());
```

The Object class defines a default toString() method which returns the class name and its reference address (not very useful normally). Many classes override toString() to provide more useful information. All wrapper classes (introduced later in this module), for example, override toString() to provide a string form of the value they represent. Even classes representing things without string form often implement toString() to return object state information for debugging purposes.



Sun Educational Services

## Inner Classes

- Were added to JDK 1.1
- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have access to their enclosing class's scope

## *Inner Classes*

Inner classes, sometimes called nested classes, were added to JDK 1.1 and above. Inner classes allow a class definition to be placed inside another class definition. Inner classes are a useful feature because they allow you to group classes that logically belong together and to control the visibility of one within another.

# Inner Classes

## Inner Class Basics

The following example shows a common way to use inner classes. Lines 19 through 25 describe an inner class

```
1  import java.awt.*;
2  import java.awt.event.*;
3
4  public class MyFrame extends Frame {
5    private Button myButton;
6    private TextArea myTextArea;
7    private int count;
8
9    public MyFrame() {
10     super("Inner Class Frame");
11     myButton = new Button("Click me");
12     myTextArea = new TextArea();
13     add(myButton, BorderLayout.CENTER);
14     add(myTextArea, BorderLayout.NORTH);
15     ButtonListener bList = new ButtonListener();
16     myButton.addActionListener(bList);
17   }
18
19   class ButtonListener implements ActionListener {
20     public void actionPerformed(ActionEvent e) {
21       count++;
22       myTextArea.setText(
23               "button clicked " + count + " times.");
24     }
25   }
26
27   public static void main (String args[]) {
28     MyFrame f = new MyFrame();
29     f.setSize(300,300);
30     f.setVisible(true);
31   }
32 }
```

## Inner Classes

The previous example consists of a class `MyFrame` which includes an inner class `ButtonListener`. The compiler generates a class file, `MyFrame$ButtonListener.class` in addition to `MyFrame.class`.

### How Do Inner Classes Work?

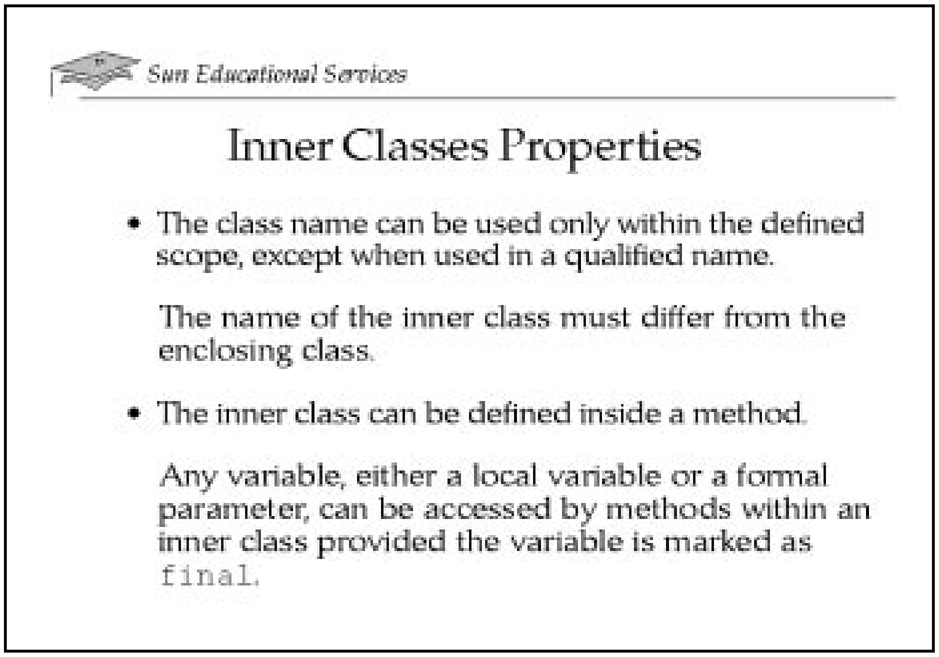
Inner classes have access to their enclosing class's scope. The accessibility of the members of the enclosing class is crucial and very useful. The access to the enclosing class's scope is possible because the inner class actually has a hidden reference to the outer class context (such as outer class "`this`").

```
public class MyFrame extends Frame {
    Button myButton;
    TextArea myTextarea;
    public MyFrame() {
       .....................
       .....................
       MyFrame$ButtonListener bList = new
                 MyFrame$ButtonListener(this);
       myButton.addActionListener(bList);
    }
    class MyFrame$ButtonListener implements
                    ActionListener{
       private MyFrame outerThis;
       Myframe$ButtonListener(MyFrame outerThisArg){
          outerThis = outerThisArg;
       }

       public void actionPerformed(ActionEvent e) {
         outerThis.MyTextArea.setText(
                                   "buttonclicked");
       .....................
       .....................
    }
    public static void main(String args[]) {
       MyFrame f = new MyFrame();
       f.setSize(300,300);
       f.setVisible(true);
    }
}
```

## How Do Inner Classes Work? (Continued)

Sometimes you might want to create an instance of an inner class from a `static` method (`main,` for example), or in some other situation where there is no `this` available. You can do this as follows:

```
public static void main(String args[]){
  MyFrame f = new MyFrame();
  MyFrame.ButtonListener bList =
     f.new ButtonListener();
  f.setSize(50,50);
  f.setVisible(true);
}
```



Sun Educational Services

## Inner Classes Properties

- The class name can be used only within the defined scope, except when used in a qualified name.

  The name of the inner class must differ from the enclosing class.

- The inner class can be defined inside a method.

  Any variable, either a local variable or a formal parameter, can be accessed by methods within an inner class provided the variable is marked as `final`.

## *Inner Classes Properties*

Inner classes have the following properties:

- The class name can be used only within the defined scope, except when referenced by its qualified name. The name of the inner class must differ from the enclosing class.
- The inner class can be defined inside a method. The rule that governs access to variables of enclosing methods is simple. Any variable, either a local variable or a formal parameter, can be accessed by methods within an inner class, provided the variable is marked as `final`.
- The inner class can use both class and instance variables of enclosing classes and local variables of enclosing blocks.
- The inner class can be defined as `abstract`.

## Inner Classes

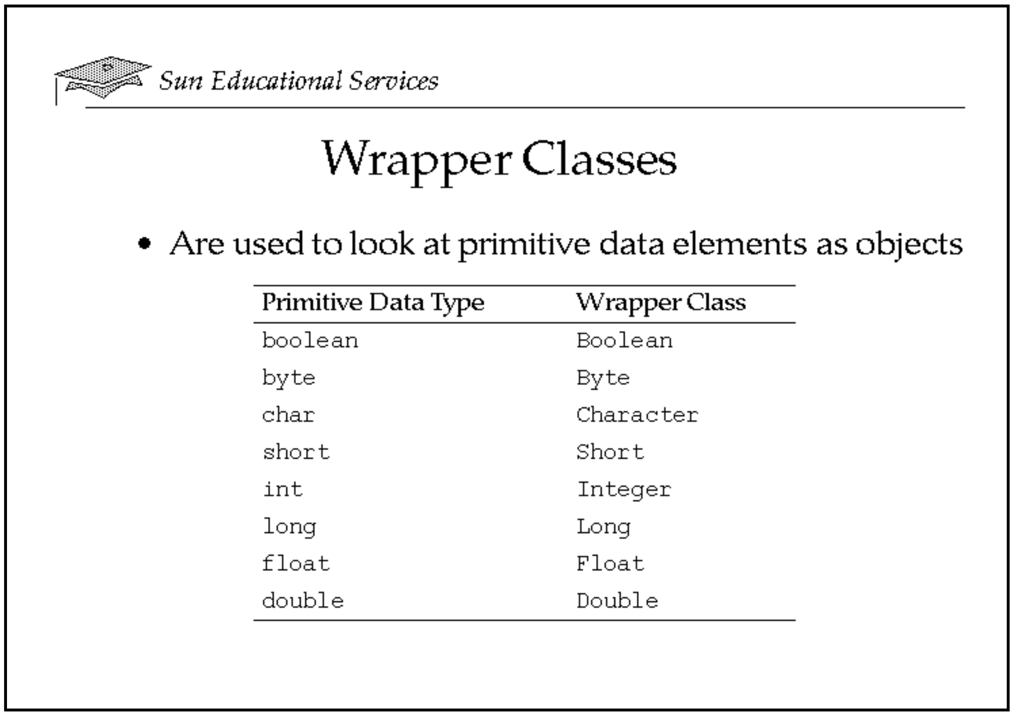
### Properties

- Only inner classes can be declared as `private` or `protected` to shield them from access by classes outside the outer class. Access protection does not prevent the inner class from using any member of another class as long as one encloses the other.
- An inner class can act as an interface implemented by another inner class.
- Inner classes that are declared `static` automatically become top-level classes. These inner classes lose their ability to use data or variables within the local scope and other inner classes.
- Inner classes cannot declare any `static` members; only top-level classes can declare `static` members. Therefore, an inner class requiring a `static` member must use one from the top-level class.

---

**Note –** Inner classes are frequently used as a convenience feature for creating event adapters. Event adapters will be discussed in subsequent modules.

---



Sun Educational Services

## Wrapper Classes

- Are used to look at primitive data elements as objects

| Primitive Data Type | Wrapper Class |
|---|---|
| `boolean` | `Boolean` |
| `byte` | `Byte` |
| `char` | `Character` |
| `short` | `Short` |
| `int` | `Integer` |
| `long` | `Long` |
| `float` | `Float` |
| `double` | `Double` |

## *Wrapper Classes*

The Java programming language does not look at primitive data types as objects. For example, numerical, boolean, and character data are treated in the primitive form itself for the sake of efficiency. The Java programming language provides *wrapper* classes to manipulate primitive data elements as objects. Such data elements are "wrapped" in an object created around them. Each Java primitive data type has a corresponding *wrapper class* in the `java.lang` package. Each wrapper class object encapsulates a single primitive value. (See Table 6-2.)

## Wrapper Classes

**Table 6-2** Wrapper Classes

| **Primitive Data Type** | **Wrapper Class** |
|---|---|
| `boolean` | Boolean |
| `byte` | Byte |
| `char` | Character |
| `short` | Short |
| `int` | Integer |
| `long` | Long |
| `float` | Float |
| `double` | Double |

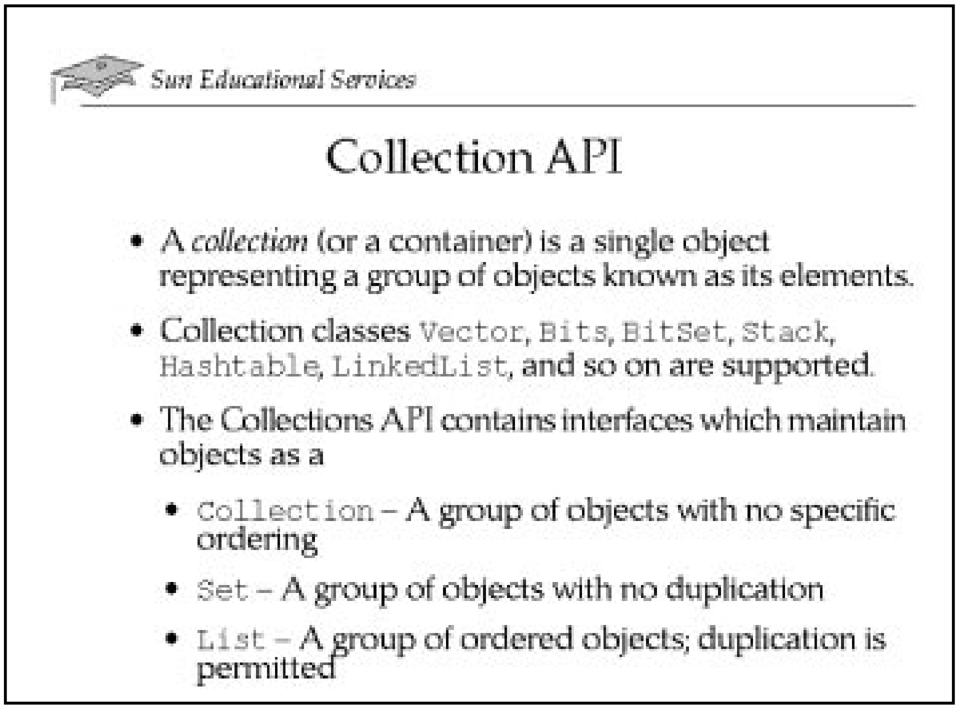
A wrapper class object can be constructed by passing the value to be wrapped into the appropriate constructor. For example:

```
int pInt = 500;
Integer wInt = new Integer(pInt);
int p2 = wInt.intValue();
```

✓ ***It can also be constructed by passing a string that represents a value to be wrapped. If the string does not represent a valid value, a*** `NumberFormatException` ***is thrown, except in the case of a*** `boolean`***. Wrapped values can be extracted as a numeric type using appropriate call. For example, a*** `long` ***value can be extracted using*** `public long longValue()`***.***

Wrapper classes are useful when converting primitive data types because of the many wrapper class methods available. For example:

```
int x = Integer.valueOf(str).intValue();
```



Sun Educational Services

## Collection API

- A *collection* (or a container) is a single object representing a group of objects known as its elements.
- Collection classes `Vector`, `Bits`, `BitSet`, `Stack`, `Hashtable`, `LinkedList`, and so on are supported.
- The Collections API contains interfaces which maintain objects as a
  - `Collection` – A group of objects with no specific ordering
  - `Set` – A group of objects with no duplication
  - `List` – A group of ordered objects; duplication is permitted

## *Collection API*

A *collection* (or a *container*) is a single object representing a group of objects, known as its *elements*. Collections typically deal with many types of objects all of which are of a particular kind (that is, they all descend from a common parent type). The Java programming language supports collection classes `Vector`, `Bits`, `Stack`, `Hashtable`, `BitSet`, `LinkedList`, and so on. `Stack`, for example, implements a last-in-first-out (LIFO) sequence and `Hashtable` provides an associative array of objects.

✓ ***If a structured query language (SQL) database API furnishes a collection and the GUI toolkit expects a collection, these APIs will interoperate seamlessly as these take collections as input and return collections as output.***

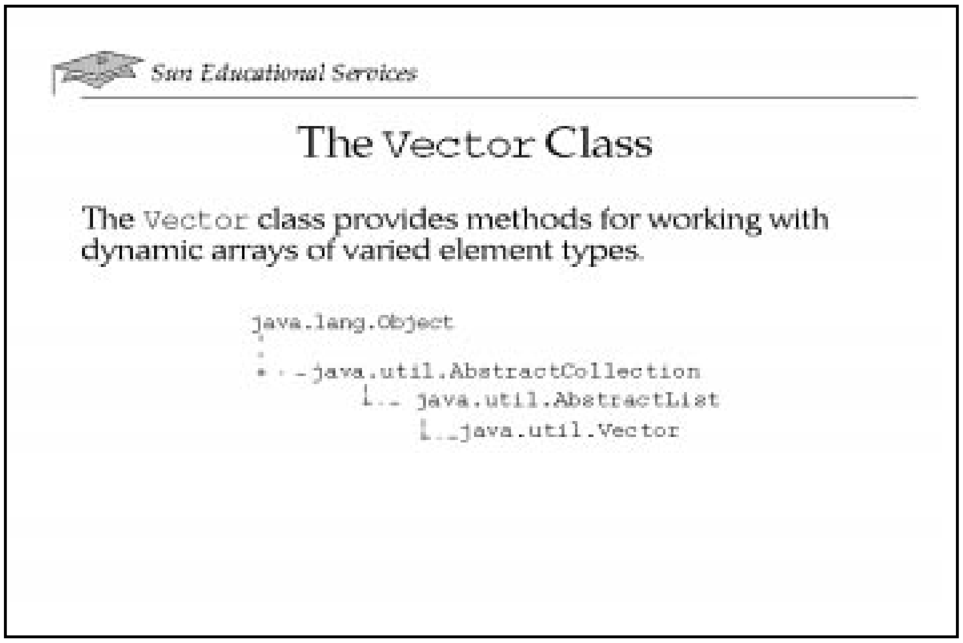
Collections maintain references to objects of type `Object`. This allows any object to be stored in the collection. It also necessiates the use of correct casting before you can use the object, after retrieving it from the collection.

## Collection API

The Collection API typically consists of interfaces which maintain objects.

- `Collection` – A group of objects with no specific ordering.
- `Set` – A group of objects with no duplication.
- `List` – A group of ordered objects; duplicates are permitted.

The API also contains classes such as `HashSet`, and `ArrayList` which implement these interfaces. Methods supporting some algorithms like sorting, binary searching, evaluating the minimum and maximum out of lists, and collections are also provided by the API.



Sun Educational Services

## The Vector Class

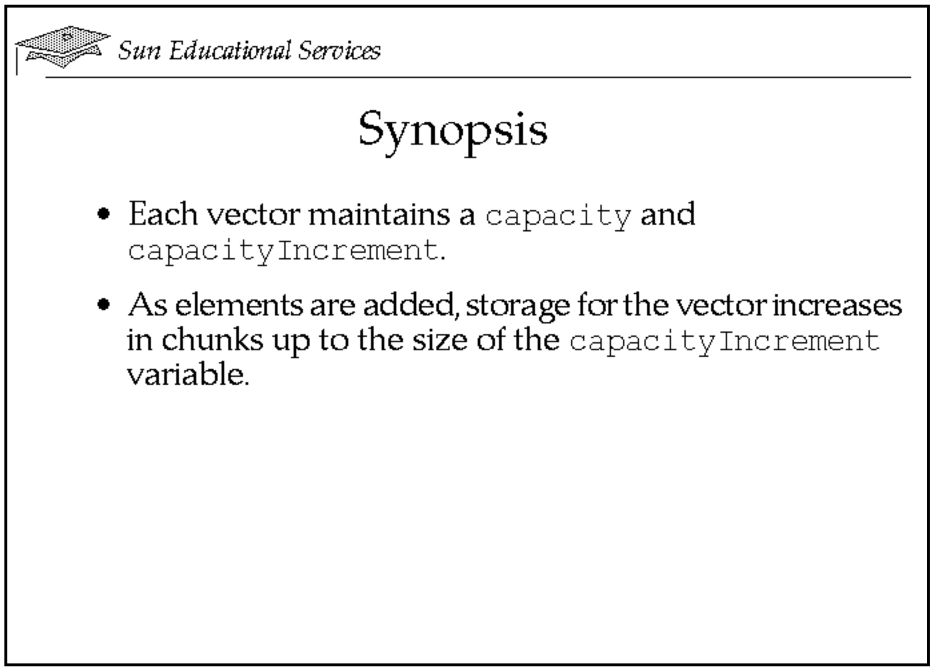
The Vector class provides methods for working with dynamic arrays of varied element types.

```
java.lang.Object
 |
 |
 +--java.util.AbstractCollection
         |
         +--java.util.AbstractList
               |
               +--java.util.Vector
```

## *The* `Vector` *Class*

The `Vector` class provides methods for working with dynamic arrays of varied element types.

```
java.lang.Object
 |
 |
 + - - java.util.AbstractCollection
            |
            L - - java.util.AbstractList
                       |
                       L - - java.util.Vector
```



Sun Educational Services

## Synopsis

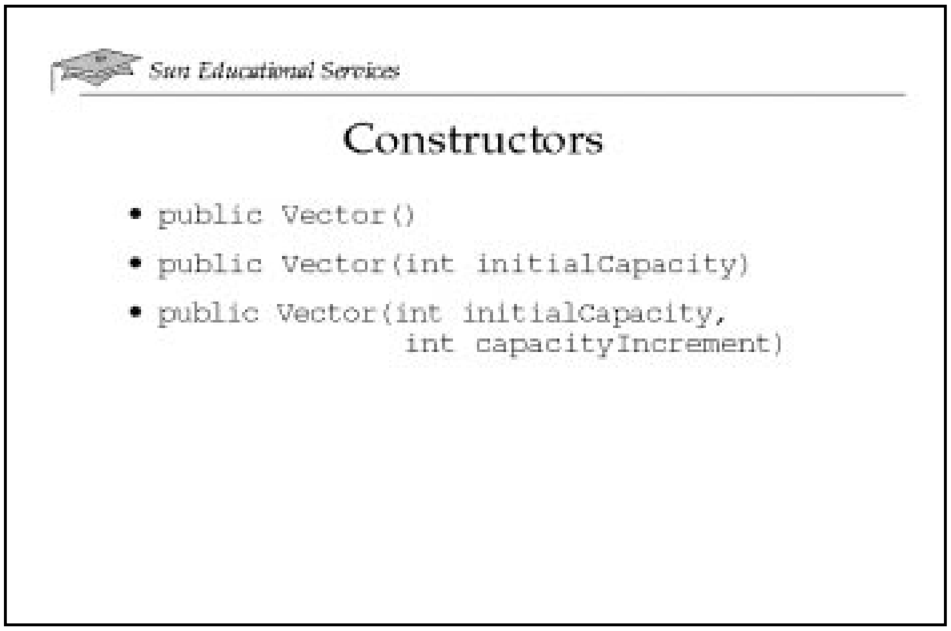
- Each vector maintains a `capacity` and `capacityIncrement`.
- As elements are added, storage for the vector increases in chunks up to the size of the `capacityIncrement` variable.

## *The* `Vector` *Class*

### *Synopsis*

```
public class Vector extends AbstractList
    implements List, Cloneable, Serlializable
```

Each vector maintains a `capacity` and `capacityIncrement`. As elements are added to a vector, storage for the vector increases, in chunks to the size of the `capacityIncrement` variable. The capacity of a vector is always at least as large as the size of the vector (It is usually larger.)


Sun Educational Services

## Constructors

- `public Vector()`
- `public Vector(int initialCapacity)`

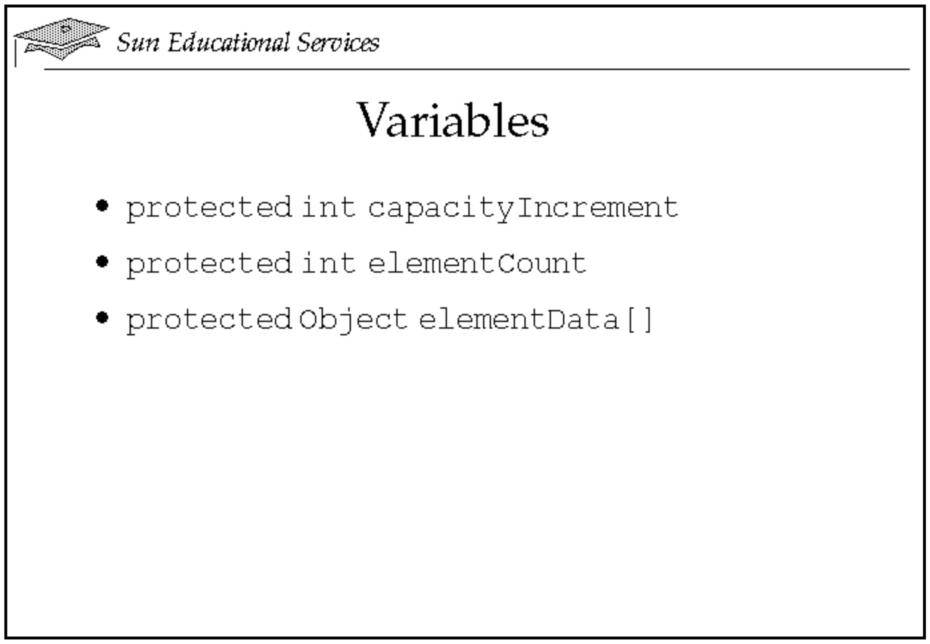
-
  ```
  public Vector(int initialCapacity,
                int capacityIncrement)
  ```

## *The* `Vector` *Class*

### *Constructors*

The constructors for the `Vector` class are:

- `public Vector()` – Constructs an empty vector
- `public Vector(int initialCapacity)` – Constructs an empty vector with the specified storage capacity
- `public Vector(`
  `int initialCapacity,int capacityIncrement)` – Constructs an empty vector with the specified storage capacity and the specified `capacityIncrement`



Sun Educational Services

## Variables

- protected int capacityIncrement
- protected int elementCount
- protected Object elementData[]

## *The* `Vector` *Class*

### *Variables*

The `Vector` class contains the following instance variables:

- `protected int capacityIncrement` – The size of the increment. (If 0, the size of the buffer is doubled every time it needs to grow.)
- `protected int elementCount` – The number of elements in the buffer.
- `protected Object elementData[]` – The buffer where elements are stored.

## *The* `Vector` *Class*

### *Methods*

Following are some of the methods in the `Vector` class. Refer to the Java API for a description of all methods in this class.

- `public final int size()` – Returns the number of elements in the vector. (This is not the same as the vector's capacity.)
- `public final boolean contains(Object elem)` – Returns true if the specified object is a value of the collection.
- `public final int indexOf(Object elem)` – Searches for the specified object starting from the first position, and returns an index to it (or -1 if the element is not found). It uses the object's `equals()` method, so if this object does not override `Object`'s `equals()` method, it will only compare object references, not object contents.
- `public final synchronized Object elementAt(int index)` – Returns the element at the specified index. It throws `ArrayIndexOutOfBoundsException` if `index` is invalid.
- `public final synchronized void setElementAt(Object obj, int index)` – Replaces the element at the specified index with the specified object. It throws `ArrayIndexOutOfBoundsException` if `index` is invalid.
- `public final synchronized void removeElementAt(int index)` – Deletes the element at the specified index. It throws `ArrayIndexOutOfBoundsException` if `index` is invalid.
- `public final synchronized void addElement(Object obj)` – Adds the specified object as the last element of the vector.
- `public final synchronized void insertElementAt(Object obj, int index)` – Inserts the specified object as an element at the specified index, shifting up all elements with equal or greater index. It throws `ArrayIndexOutOfBoundsException` if `index` is invalid.

## *The* Vector *Class*

### *Sample* Vector *Template*

The following template can be used to add different element types to a vector and to print out vector elements.

---

**Note –** This program uses methods from the classes discussed previously in this module.

---

```
import java.util.*;

public class MyVector extends Vector {
  public MyVector() {
    // storage capacity & capacityIncrement
    super(1,1);
  }

  public void addInt(int i) {
    // addElement requires Object arg
    addElement(new Integer(i));
  }

  public void addFloat(float f) {
    addElement(new Float(f));
  }

  public void addString(String s) {
    addElement(s);
  }

  public void addCharArray(char a[]) {
    addElement(a);
  }

  public void printVector() {
    Object o;

    // compare with capacity()
    int length = size();
    System.out.println("Number of vector elements is " +
        length + " and they are:");
```

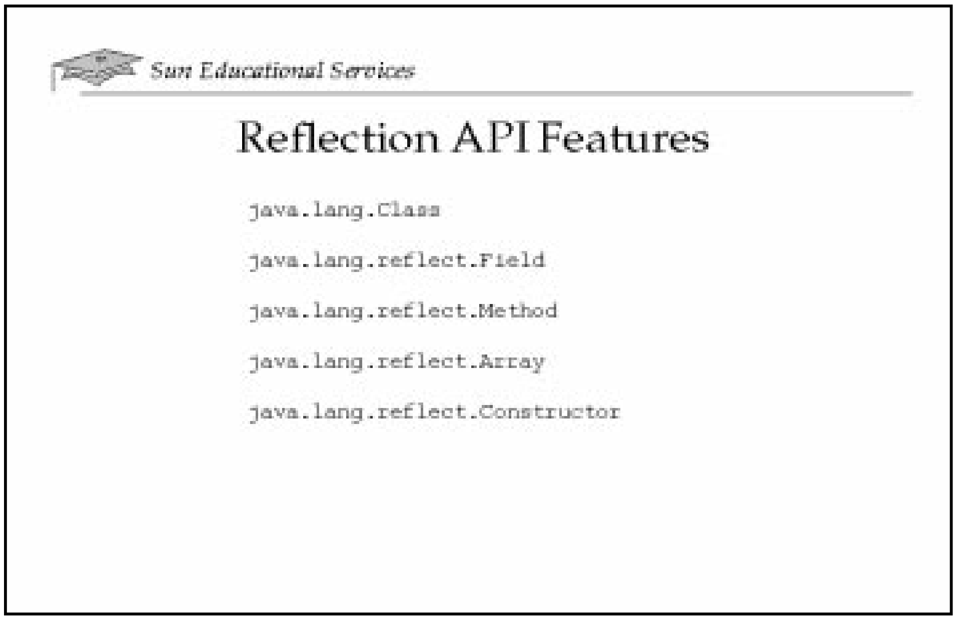
## *The* `Vector` *Class*

### *Sample* `Vector` *Template (Continued)*

```
    for (int i = 0; i < length; i++) {
      o = elementAt(i);

      if (o instanceof char[]) {

        // An array's toString() method does not print
        // what we want.
        System.out.println(String.copyValueOf((char[]) o));
      } else {
        System.out.println(o.toString());
      }
    }
  }

  public static void main (String args[]) {
    MyVector v = new MyVector();
    int digit = 5;
    float real = 3.14F;
    char letters[] = { 'a', 'b', 'c', 'd' };
    String s = new String("High there!");

    v.addInt(digit);
    v.addFloat(real);
    v.addString(s);
    v.addCharArray(letters);

    v.printVector();
  }
}
```

This program produces the following output:

```
Number of vector elements is 4 and are:
5
3.14
Hi there!
abcd
```



Sun Educational Services

## Reflection API Features

```
java.lang.Class
java.lang.reflect.Field
java.lang.reflect.Method
java.lang.reflect.Array
java.lang.reflect.Constructor
```

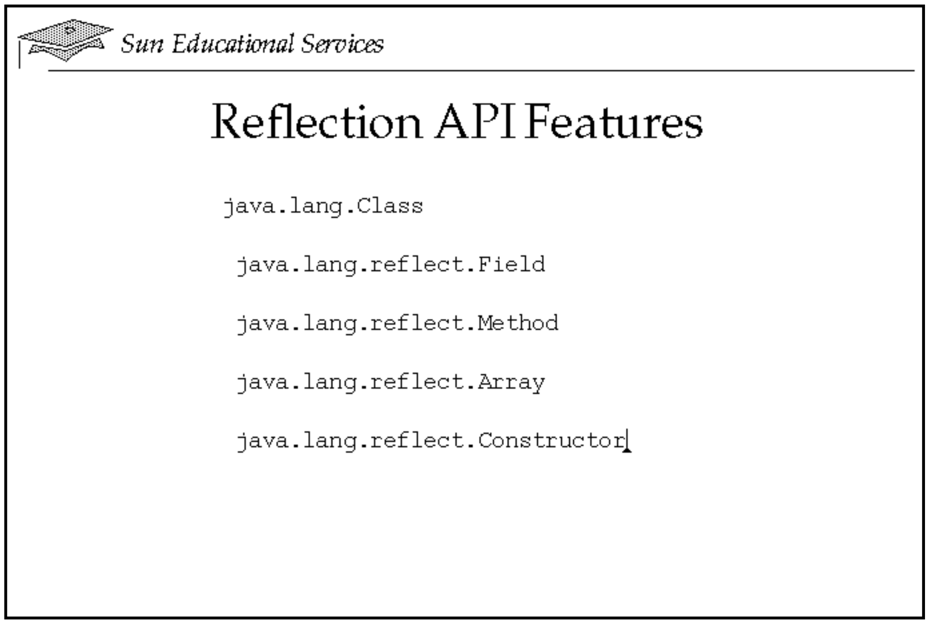
## *Reflection API*

### *Introduction*

The Java Reflection API provides a set of classes that are used to determine a class file's variables and methods. Used commonly for the purpose of dynamic discovery and code execution, the API can be used to

- Construct new class instances and new arrays
- Access and modify fields of objects and classes
- Invoke methods on objects and classes
- Access and modify elements of arrays

These operations are possible only if the security policy permits them. The Reflection API is useful in situations where you need to retrieve and manipulate information at runtime. For example, it can be used if you are writing a Java software interpreter or debugger.


Sun Educational Services

## Reflection API Features

java.lang.Class

java.lang.reflect.Field

java.lang.reflect.Method

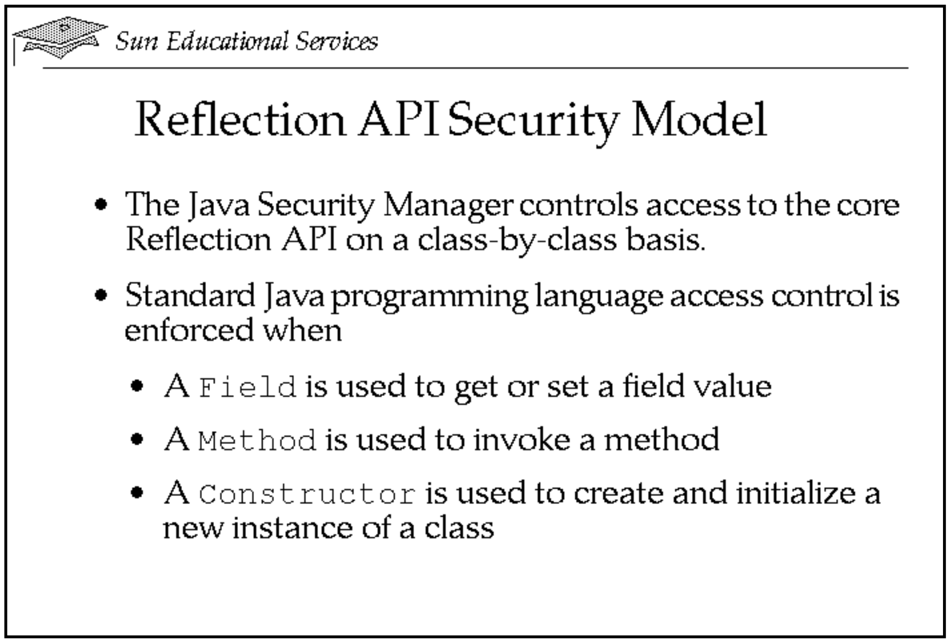
java.lang.reflect.Array

java.lang.reflect.Constructor

## *Reflection API Features*

Some features of the core Reflection API which defines classes and methods are as follows:

- The class `java.lang.Class` provides methods which get information about a class and its fields, constructors, and methods.
- The class `java.lang.reflect.Field` provides methods which set and get information about the fields in the class.
- The class `java.lang.reflect.Method` provides methods which access and call methods in the class, and get their signatures.
- The class `java.lang.reflect.Array` enables an introspection of an array object.
- The class `java.lang.reflect.Constructor` provides reflection access to constructors.



Sun Educational Services

## Reflection API Security Model

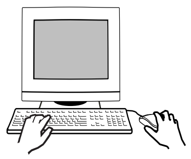
- The Java Security Manager controls access to the core Reflection API on a class-by-class basis.
- Standard Java programming language access control is enforced when
  - A `Field` is used to get or set a field value
  - A `Method` is used to invoke a method
  - A `Constructor` is used to create and initialize a new instance of a class

## *Reflection API Security Model*

The Java Security Manager controls access to the core Reflection API on a class-by-class basis. Standard Java programming language access control is enforced when

- A `Field` is used to get or set a field value.
- A `Method` is used to invoke a method.
- A `Constructor` is used to create and initialize a new instance of a class.

## Exercise: Working With Advanced Language Features



**Exercise objective –** You will rewrite, compile, and run three programs that use the bank account model and employ advanced object-oriented features such as inner classes, vector classes, and interfaces.

### Preparation

In order to successfully complete this lab, you must be familiar with the object-oriented concepts presented in this module and the previous module.

### Tasks

#### *Level 1 Lab: Modify the Bank Account Problem*

Complete the following steps:

1. Define an `interface` named `Personal`, consisting of two methods: `deposit` and `withdraw`.
2. Redefine the class, `Account.java`, from Module 5, using the Personal interface to define a set of different account types. It should be capable of handling personal accounts, which are further categorized into checking and savings account.
3. Design and develop methods which offer protection. For example, if a client has a savings and a checking account, ensure that the checking account is protected by the savings account.

## Exercise: Working With Advanced Language Features

### Tasks

#### Level 2 Lab: Use Inner Classes

Complete these steps:

1. Create a class called `BasicArray`. Declare and initialize an array called `thisArray` which contains four integers.

2. Create a class named `Factorial` that contains a method which computes the factorial of its argument.

3. Create an instance of class `Factorial` from the main method of `BasicArray`, and then call its method to compute the factorial of each of the four integers.

4. Compile and test the program.

5. Move the class `Factorial` inside class `BasicArray`. `Factorial` is now an inner class to `BasicArray`.

6. Compile and test the program.

## Exercise: Working With Advanced Language Features

### Tasks

#### Level 3 Lab: Add `find` and `delete` Methods to `MyVector` Class

Use the following steps:

1. Add a `find` method to the `MyVector` class that returns the location of the element passed as an argument.

   Have the method return -1 if the argument is not found.For example:

   ```
   java MyVectorFind 3.14
   3.14 is located at index 1
   Number of vector elements is 4 and are:
   5
   3.14
   Hi there!
   abcd

   java MyVectorFind c
   args[0]=c, not found in vector
   Number of vector elements is 4 and are:
   5
   3.14
   Hi there!
   abcd
   ```

2. Add a `delete` method to the `MyVector` class that removes all elements that match the argument.

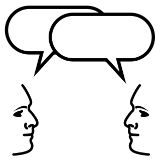
   The method should return `true` or `false`: `true` if deletion was successful and `false` if it was unsuccessful (that is the element did not exist in the vector).

   For example:

   ```
   java MyVectorDelete 3.14
   Elements 3.14 successfully deleted from vector.
   Number of vector elements is 3 and are:
   5
   Hi there!
   abcd
   ```

# *Exercise: Working With Advanced Language Features*

## *Exercise Summary*



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

❑ Describe `static` variables, methods, and initializers

❑ Describe `final` classes, methods and variables

❑ List the access control levels

❑ Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to JDK 1.2

❑ Describe how to apply collections and reflections

❑ In a Java software program, identify

  ▼ `static` methods and variables

  ▼ `public`, `private`, `protected`, and default variables

❑ Use `abstract` classes and methods

❑ Explain how and when inner classes are used

❑ Explain how and when interfaces are used

❑ Describe the difference between `==` and `equals()`

❑ Define a `Vector`

❑ Describe wrapper classes

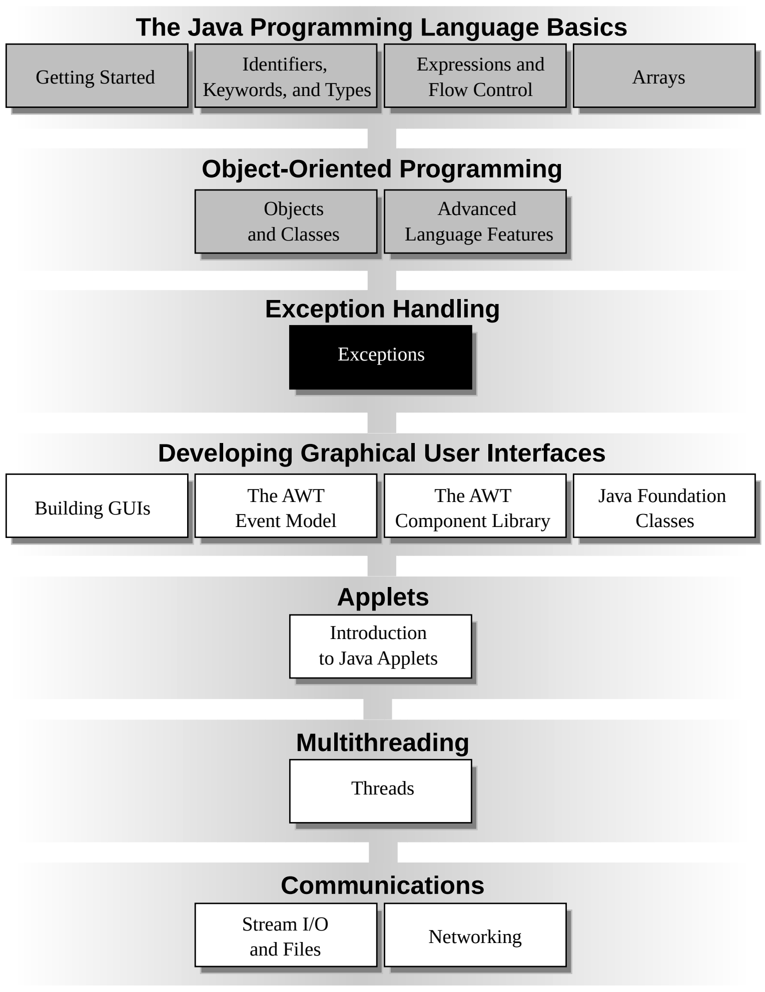
## Think Beyond

What features does the Java programming language have to deal with runtime error conditions in a straightforward manner?

# Exceptions 7

## Course Map

This module covers the error handling facilities built into the Java programming language.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to the question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following question is relevant to the material presented in this module:

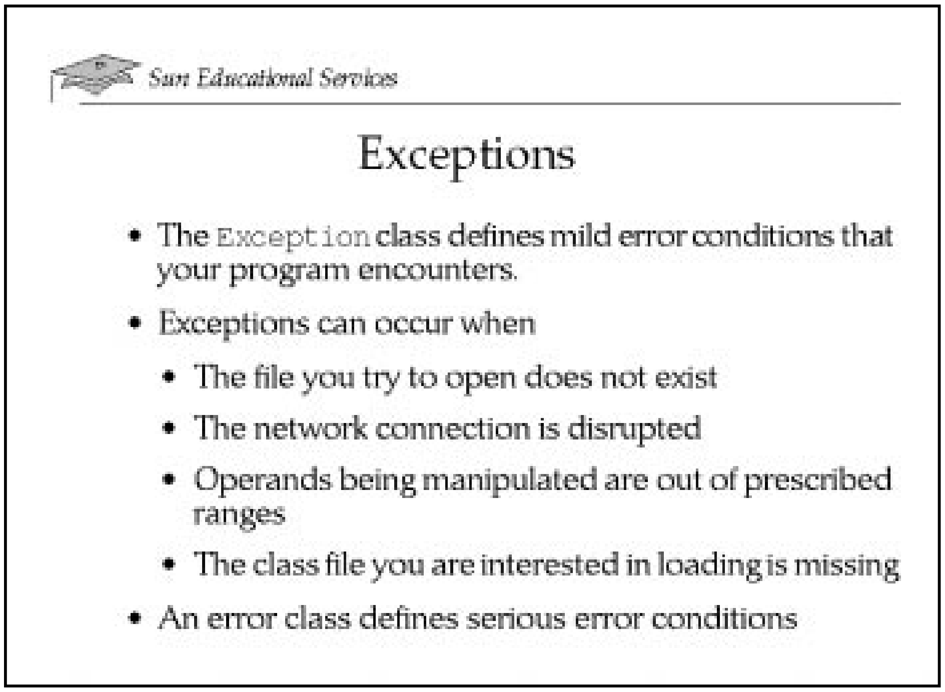
- In most programming languages, how are runtime errors resolved?

✓ ***Writing a good computer program includes proper error handling and recovery. The Exception mechanism in the Java programming language provides a simple, streamlined, organized approach. This module covers the Exception mechanism built into the Java programming language, and how to use it effectively.***

## Objectives

Upon completion of this module, you should be able to

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions


Sun Educational Services

## Exceptions

- The `Exception` class defines mild error conditions that your program encounters.
- Exceptions can occur when
  - The file you try to open does not exist
  - The network connection is disrupted
  - Operands being manipulated are out of prescribed ranges
  - The class file you are interested in loading is missing
- An error class defines serious error conditions

## *Exceptions*

### *Introduction*

What is an exception? In the Java programming language, the `Exception` class defines mild error conditions that your programs might encounter. Rather than letting the program terminate, you can write code to handle your exceptions and continue program execution.

Any abnormal condition that disturbs the normal program flow while the program is in execution is an error or exception. For example, exceptions can occur when:

- The file you try to open does not exist
- The network connection is disrupted
- Operands being manipulated are out of prescribed ranges
- The class file you are interested in loading is missing

# Exceptions

## Introduction

In the Java programming language, the `Error` class defines what are considered to be serious error conditions that you should not attempt to recover from. In most cases, it is advisable to let the program terminate when such an error is encountered.

The Java programming language implements C++ style exceptions to help you build resilient code. When an error occurs in your program, the method that finds the error can “throw” an exception back to its caller, to signal that a problem has occurred. The calling method then “catches” the thrown exception and, when possible, recover from it. This scheme gives the programmer the option of writing a “handler” to deal with the exception.

You can determine what exceptions a method throws by browsing the API.

## Example

Consider a version of the `HelloWorld.java` program which cycles through messages.

```
public class HelloWorld {
   public static void main (String args[]) {
      int i = 0;

      String greetings [] = {
         "Hello world!",
         "No, I mean it!",
         "HELLO WORLD!!"
      };

      while (i < 4) {
         System.out.println (greetings[i]);
         i++;
      }
   }
}
```

✓ ***This program quickly produces an exception. Ask students where the exception will come from.***

✓ ***The exception produced is the*** `ArrayIndexOutOfBoundsException`***. It is produced in the*** `System.out.println` ***method when*** `i` ***has a value of 3.***

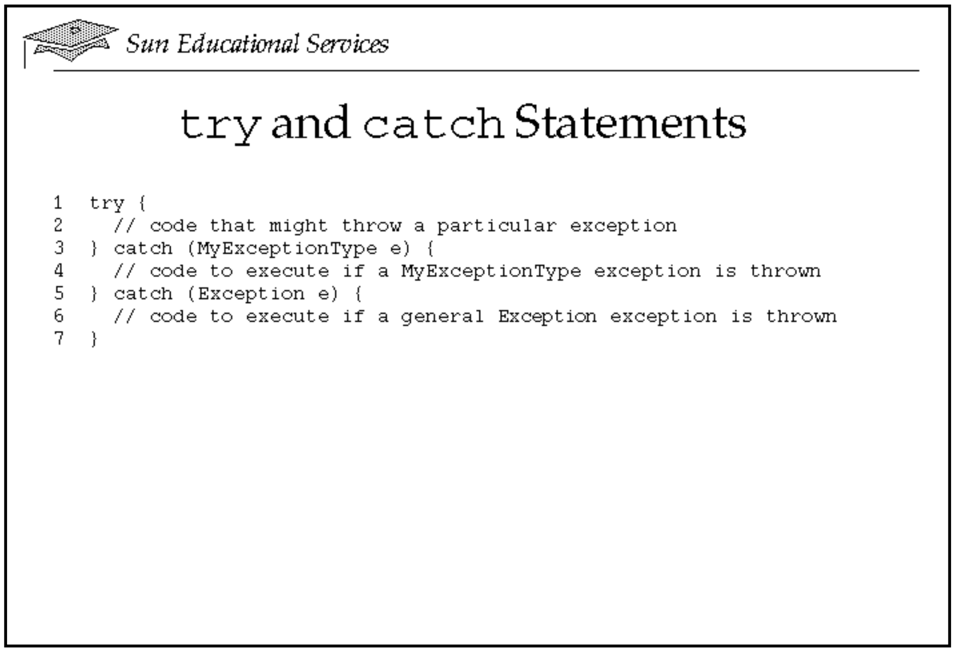
## Exception Handling

### Introduction

Normally, a program terminates with an error message when an exception is thrown, as does the program shown previously after its loop has executed four times.

```
java HelloWorld
Hello world!
No, I mean it!
HELLO WORLD!!
java.lang.ArrayIndexOutOfBoundsException: 3
   at HelloWorld.main(HelloWorld.java:12)
```

Exception handling allows a program to catch exceptions, handle them, and then continue program execution. It is structured so that error cases do not get in the way of the normal flow of a program. These special cases are handled when they occur, in separate code blocks associated with the code for normal execution. This produces more legible and manageable code.



Sun Educational Services

## try and catch Statements

```
try {
  // code that might throw a particular exception
} catch (MyExceptionType e) {
  // code to execute if a MyExceptionType exception is thrown
} catch (Exception e) {
  // code to execute if a general Exception exception is thrown
}
```

## *Exception Handling*

The Java programming language provides a mechanism for figuring out which exception was thrown and how to recover from it.

### `try` *and* `catch` *Statements*

To handle a particular exception, place code which when invoked throws exceptions inside a `try` block and create a list of adjoining `catch` blocks, one for each possible exception that can be thrown. The block statement of a `catch` clause is executed if the exception generated matches the one listed in the `catch`. There can be multiple `catch` blocks after a `try` block, each handling a different exception type.

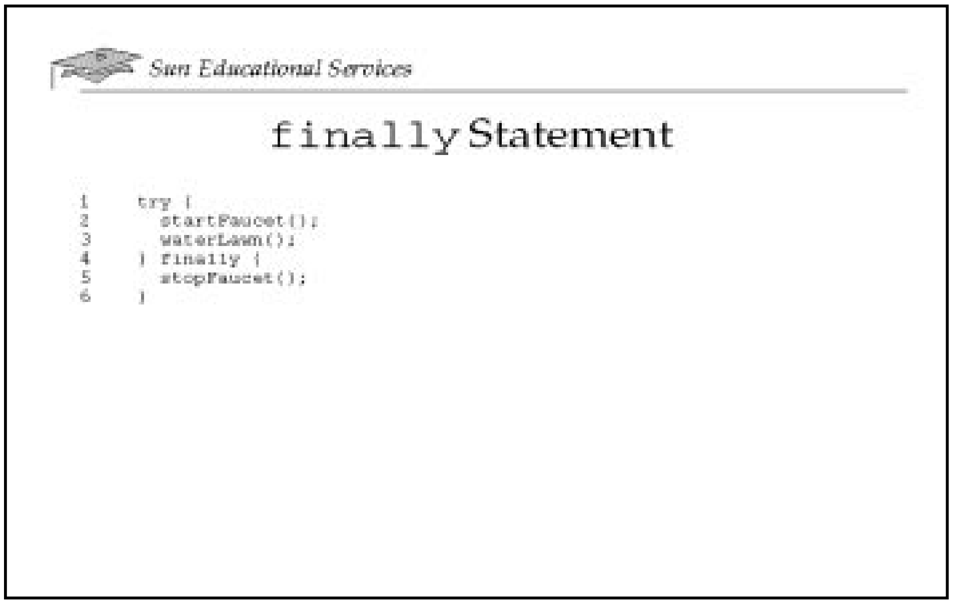
```
try {
   // code that might throw a particular exception
} catch (MyExceptionType e) {
   // code to execute if a MyExceptionType exception is thrown
} catch (Exception e) {
   // code to execute if a general Exception exception is thrown
}
```

## Exception Handling

### The Call Stack Mechanism

If a statement within a method throws an exception, that exception is thrown to the calling method. If the exception is not handled in the calling method also, it is thrown to the caller of that method. This process continues until the exception is handled. If an exception is not handled by the time it gets back to `main()` and `main()` does not handle it, the exception terminates the program abnormally.

Consider a case where the `main()` method calls another method named `first()`, and this in turn calls another method named `second()`. If an exception occurs in `second()`, it is thrown back to `first()`, where a check is made to see if there is a catch for that type of exception. If no catch exists in `first()`, the the next method in the call stack, `main()`, is checked. If the exception is not handled by the last method on the call stack, then a runtime error occurs and the program stops executing.



Sun Educational Services

## `finally` Statement

```
1    try {
2      startFaucet();
3      waterLawn();
4    } finally {
5      stopFaucet();
6    }
```

## Exception Handling

### `finally` *Statement*

The `finally` statement defines a block of code that *always* executes, regardless of whether an exception was caught. The following sample code and description is taken from the white paper, “Low Level Security in Java”, by Frank Yellin:

```
1  try {
2    startFaucet();
3    waterLawn();
4  } finally {
5    stopFaucet();
6  }
```

✓ ***Use of `catch()` is optional, depending on the situation***

## Exception Handling

### `finally` *Statement*

In the previous example, the faucet is turned off even if an exception occurs while starting the faucet or watering the lawn. The code inside the braces after the `try` is called the *protected code.*

The only time the `finally` statement would not be executed is if the `System.exit()` method, which terminates the program, is executed within the protected code. This implies that the control flow can deviate from normal sequential execution, if, for example, a `return` statement were embedded in the code inside the `try` block, the code in the `finally` block would execute before the `return`.

## Example Revisited

The following example is a rewrite of the `main()` method from page 7-21. The exception generated in the earlier version of the program is caught and the array index is reset, allowing the program to continue.

```
public class HelloWorld {
  public static void main (String args[]) {
    int i = 0;
    String greetings [] = {
      "Hello world!",
      "No, I mean it!",
      "HELLO WORLD!!"
    };

    while (i < 4) {
      try {
        System.out.println (greetings[i]);
      } catch (ArrayIndexOutOfBoundsException e){
        System.out.println("Re-stting Index Value");
        i = -1;
      } finally {
        System.out.println("This is always printed");
      }
      i++;
    }
  }
}
```

✓ ***The** `try` **statements can be nested and the exceptions can migrate upward.***

# Exception Handling

## Example Revisited (Continued)

The message displayed on the screen alternates among the following messages as the loop is executed:

```
Hello world!
This is always printed
No, I mean it!
This is always printed
HELLO WORLD!!
This is always printed
Re-setting Index Value
This is always printed
```

✓ ***Notice that the fourth message displayed is due to the generated exception being caught and handled. The array index is reset to allow continuation of the alternating messages.***

## Exception Categories

There are three broad categories of exceptions in the Java programming language. The class `java.lang.Throwable` acts as the parent class for all objects that can be thrown and caught using the exception-handling mechanisms. Methods defined in the `Throwable` class retrieve the error message associated with the exception and print the stack trace showing where the exception occurred. There are three essential subclasses of this, `Error, RuntimeException,` and `Exception`, which are shown in Figure 7-1.

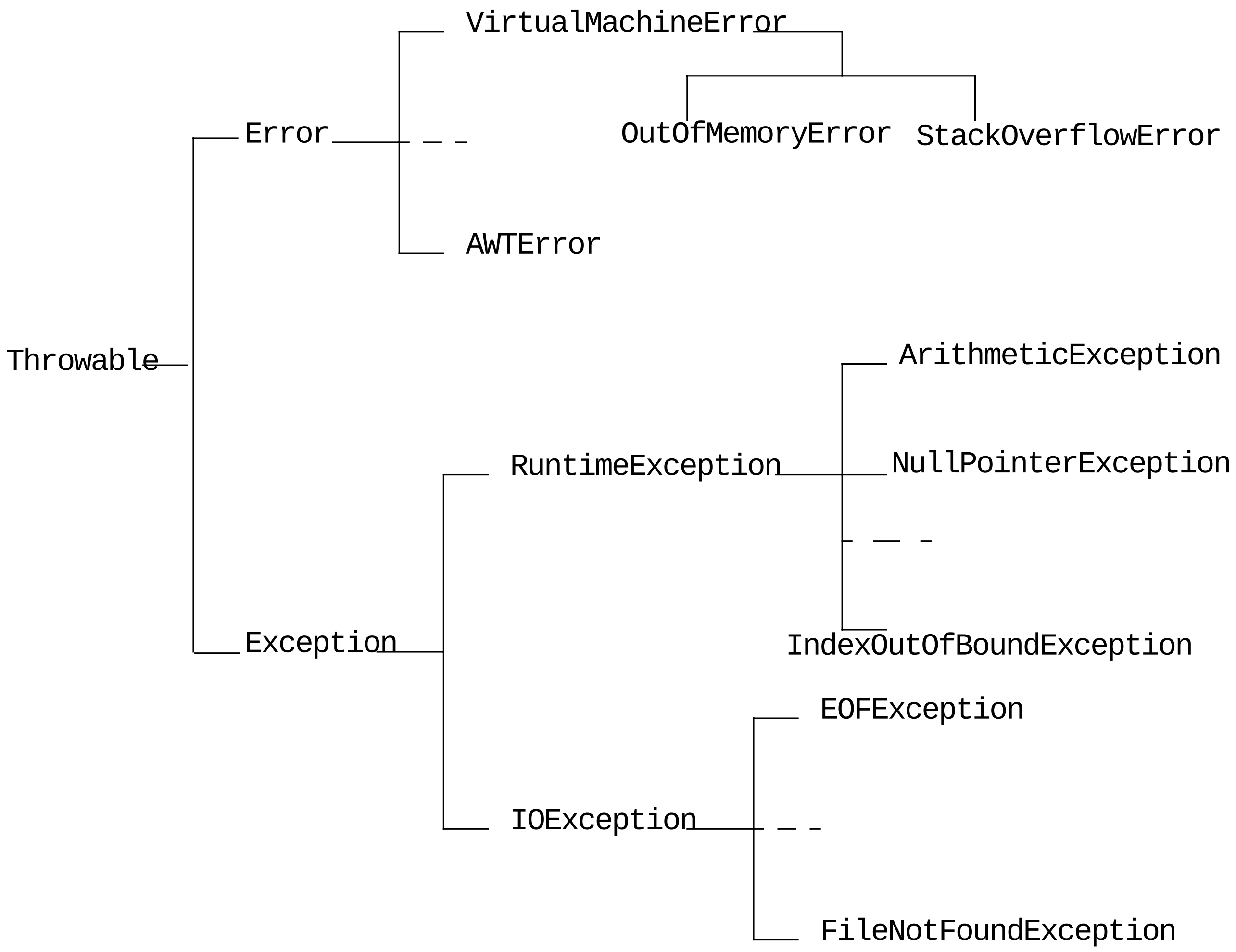
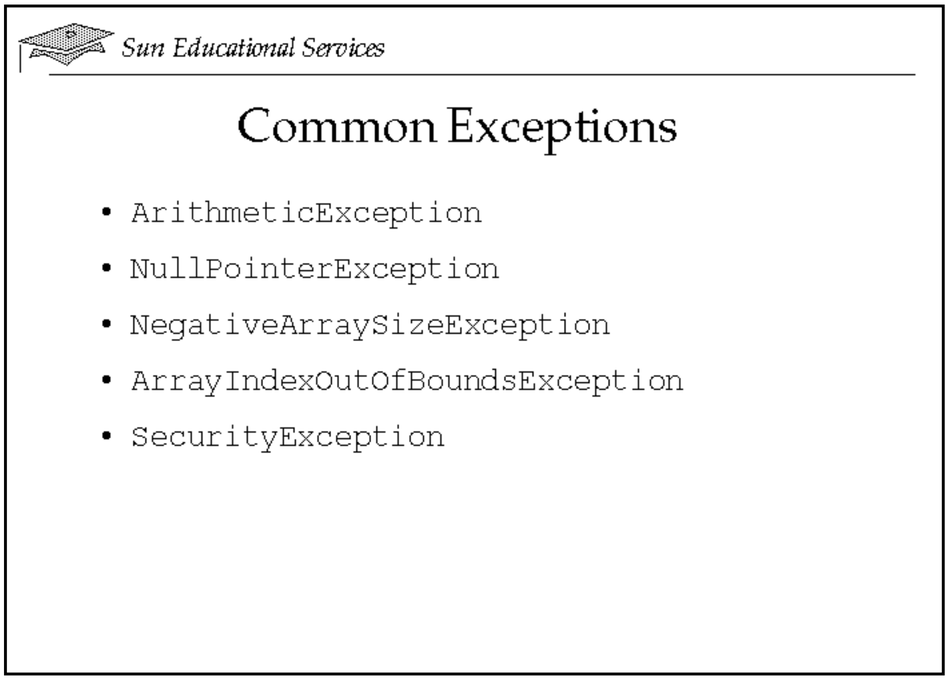


**Figure 7-1** Subclasses and Exceptions

## Exception Categories

The `Throwable` class should not be used; instead, use one of the subclass exceptions to describe any particular exception. The purpose of each exception is:

- `Error` indicates a severe problem from which recovery is difficult, if not impossible. An example is running out of memory. A program is not expected to handle such conditions.
- `RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly. An `ArrayIndexOutOfBoundsException` exception, for example, should never be thrown if the array indices do not extend past the array bounds. This would also apply, for example, to de-referencing a null object variable. Because a correctly designed and implemented program never issues this type of exception, it is usual to leave it unhandled. This results in a message at runtime, and ensures that action can be taken to correct the problem, rather than hiding it where no one will notice.
- Other exceptions indicate a difficulty at runtime that is usually caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions (user typed a wrong URL), both of which could easily occur if the user mistyped something. Since these can occur as a result of user error, programmers are encouraged to handle them.



Sun Educational Services

## Common Exceptions

- ArithmeticException
- NullPointerException
- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- SecurityException

## *Common Exceptions*

The Java programming language provides several predefined exceptions. Some of the more common exceptions are:

- `ArithmeticException` – The result of a divide by zero operation for integers.

```
int i = 12 / 0;
```

✓ ***This exception is not generated by arithmetic overflow.***

✓ `ArithmeticException`***s are not generated by dividing floating point numbers by zero; these values are defined by the IEEE and are constants (final static) which are declared in the*** **`Float`** ***class.***

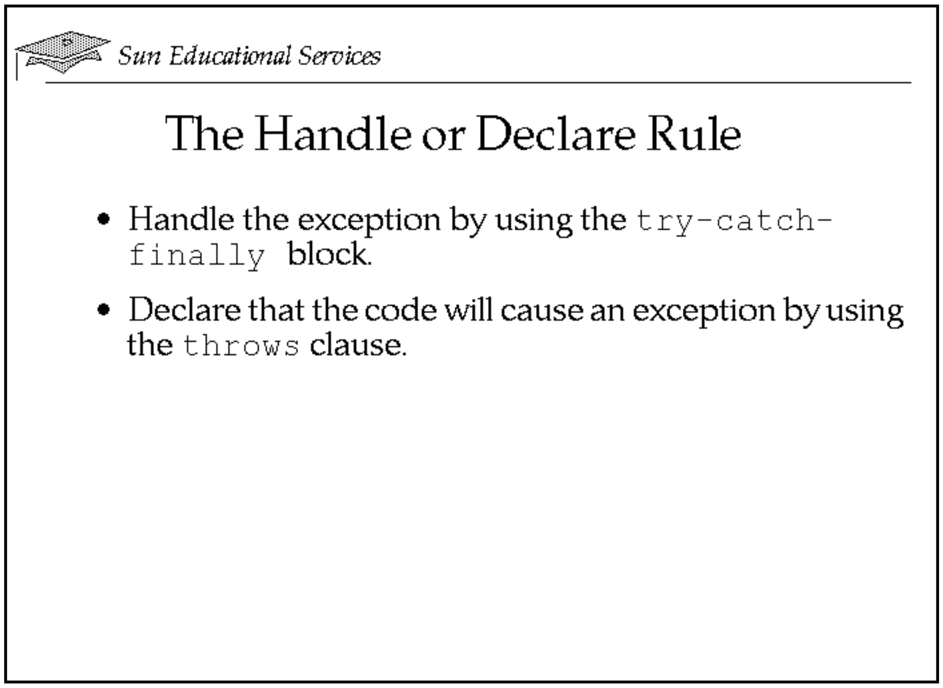
- `0.0/0 = NaN` ***(not a number)***
- ***FPN/0 = POSITIVE_INFINITY***
- ***-FPN/0 = NEGATIVE_INFINITY***

- `NullPointerException` – An attempt to access an object's attribute or method when the object is not instantiated:

```
Date d = null;
System.out.println(d.toString());
```

## Common Exceptions

- `NegativeArraySizeException` – An attempt to create an array with a negative dimension size.
- `ArrayIndexOutOfBoundsException` – An attempt to access an element of an array beyond the array's size.
- `SecurityException` – Typically thrown in a browser, the `SecurityManager` class throws an exception for applets that attempt to do any of the following (unless explicitly allowed):
    - ▼ Access a local file
    - ▼ Open a socket to the host which is not the same host that served the applet
    - ▼ Execute another program in a runtime environment


Sun Educational Services

The Handle or Declare Rule

- Handle the exception by using the `try-catch-finally` block.
- Declare that the code will cause an exception by using the `throws` clause.

## *The Handle or Declare Rule*

To encourage writing robust code, the Java programming language requires that if an `Exception` (which differs from an `Error` or `RuntimeException`) occurs while a method is on the stack (that is, it has been called), then the *caller* of that method must determine what action is to be taken if a problem arises.

There are two things the programmer can do that satisfy this requirement:

- The first is to have the calling method handle the exception by including in its code a `try {} catch(){}` block where the catch names any superclass of the thrown exception. This counts as handling the situation, even if the `catch` block is empty.

- The second is to have the calling method indicate that it will not handle the exception, and that the exception will be thrown back to *its* calling method. This is done by marking the calling method's declaration with a `throws` clause as follows:

```
public void callsTroublesome() throws IOException
```

## The Handle or Declare Rule

Following the keyword `throws` is a list of all the exceptions that the method can throw back to its caller. Although only one exception is shown here, a comma-separated list can be used if multiple possible exceptions can be thrown by this method.

✓ ***Even after an exception is caught, it can still be rethrown.***

Whether you choose to handle or declare an exception depends on whether you consider yourself or your caller a more appropriate candidate for dealing with the exception.

---

**Note –** Because the exception classes are organized into hierarchies as other classes are, and because you may use a class whenever a subclass is expected, you can catch "groups" of exceptions and handle them with the same catch code. For example, although there are several different types of `IOExceptions` (`EOFException`, `FileNotFoundException` and so on), by trapping `IOException` you can also catch instances of any subclass of `IOException`.

---

## Creating Your Own Exceptions

### Introduction

User-defined exceptions are created by extending the `Exception` class. Such exception classes can contain anything that a "regular" class contains. An example of a user-defined exception class containing a constructor, some variables, and methods is shown here:

```
public class ServerTimedOutException extends Exception {
  private int port;

  public ServerTimedOutException(String reason, int port) {
    super(reason);
    this.port = port;
  }

  // Use Exception class`s getMessage() to get the
  // reason the exception was madE

  public int getPort() {
    return port;
  }
}
```

To throw an exception that you have created, use the syntax

```
throw new ServerTimedOutException
     ("Could not connect", 80);
```

## Creating Your Own Exceptions

### Example

Consider a client-server program. In the client code, you try to connect to the server and expect the server to respond within 5 seconds. If the server does not respond, your code could throw an exception (such as a user-defined `ServerTimedOutException`) as follows:

```
public void connectMe(String serverName)
          throws ServerTimedOutException {
    int success;
    int portToConnect = 80;

    success = open(serverName, portToConnect);

    if (success == -1) {
      throw new ServerTimedOutException(
          "Could not connect", 80);
    }
  }
```

✓ ***Point out that although the*** `open()` ***call here is fictitious, the*** `throw` ***command uses the correct syntax.***

✓ ***Point out that you should create the exception at the point it is thrown. This is because the stack trace and line number information are added during construction and will be misleading otherwise.***

To catch your exception, use the `try` statement

```
  public void findServer() {
    try {
      connectMe(defaultServer);
    } catch (ServerTimedOutException e) {
      System.out.println(
          "Server timed out, trying alternative");
      try {
        connectMe(alternativeServer);
      } catch (ServerTimedOutException e1) {
        System.out.println(
            "Error: " + e1.getReason() +
            " connecting to port " + e1.getPort());
      }
    }
  }
```

## Creating Your Own Exceptions

### Example (Continued)

---

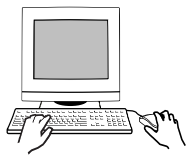
**Note –** The `try` and `catch` blocks can be nested as shown in the previous example.

---

It is also possible to partially process an exception and then throw it as well. For example:

```
try {
  connectMe(defaultServer);
} catch (ServerTimedOutException e) {
     System.out.println(“Error caught “);
     throw e;
}
```

## Exercise: Handling and Creating Exceptions



**Exercise objective –** You will gain experience with the Exception mechanism by writing Java software programs which create and handle Exceptions.

### Preparation

In order to successfully complete this lab, you must understand the concepts of handling runtime errors called exceptions.

### Tasks

#### *Level 1 Lab: Handle an Exception*

Complete the following steps:

1. Use the sample exception program from page 7-6 to create an exception when the array index exceeds the size of the array. (Or modify your own program so that it creates an exception.)
2. Use the `try` and `catch` statements to recover from the exception.

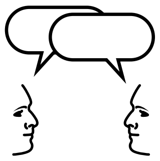
#### *Level 2 Lab: Create Your Own Exception*

Perform this step,

1. Use the `bank` package you created in the Module 5 lab and add the following exceptions:
   - `AccountOverdrawnException` – When an attempt is made to take more money out of the account than the account holds.
   - `InvalidDepositException` – When invalid amounts of money (less than zero) are deposited.

## Exercise: Handling and Creating Exceptions

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Define exceptions
- ❑ Use `try`, `catch`, and `finally` statements
- ❑ Describe exception categories
- ❑ Identify common exceptions
- ❑ Develop programs to handle your own exceptions

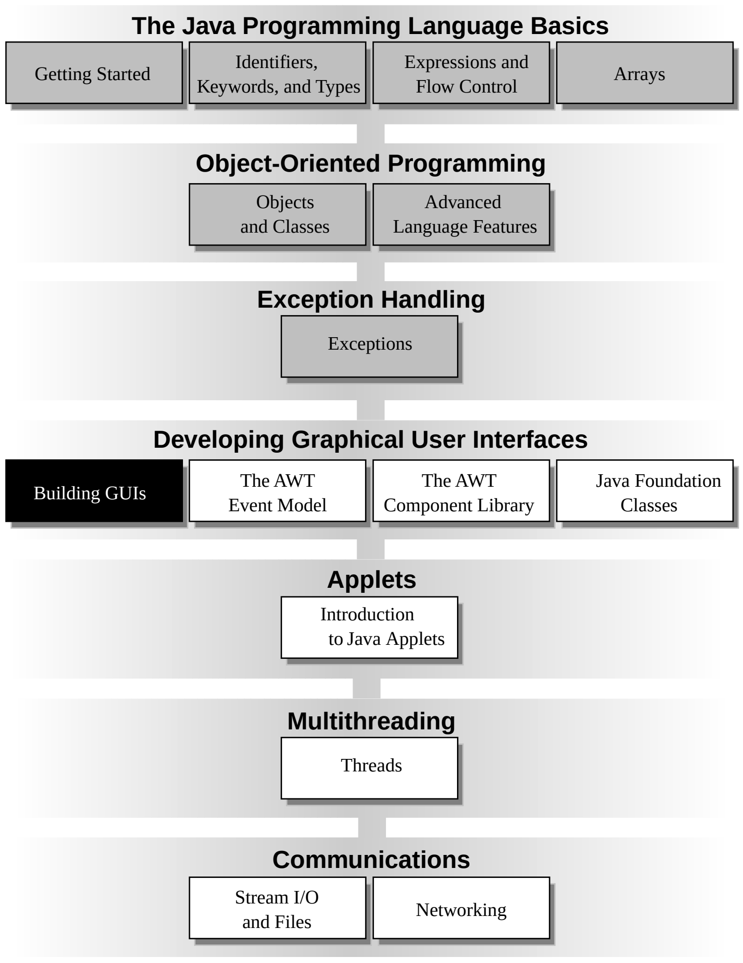
## Think Beyond

What features does the Java application environment have that support user interface development?

# Building GUIs 8

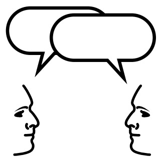
## Course Map

This module covers setup and layout of graphical user interfaces. It introduces the Abstract Windowing Toolkit, a package of classes from which GUIs are built.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

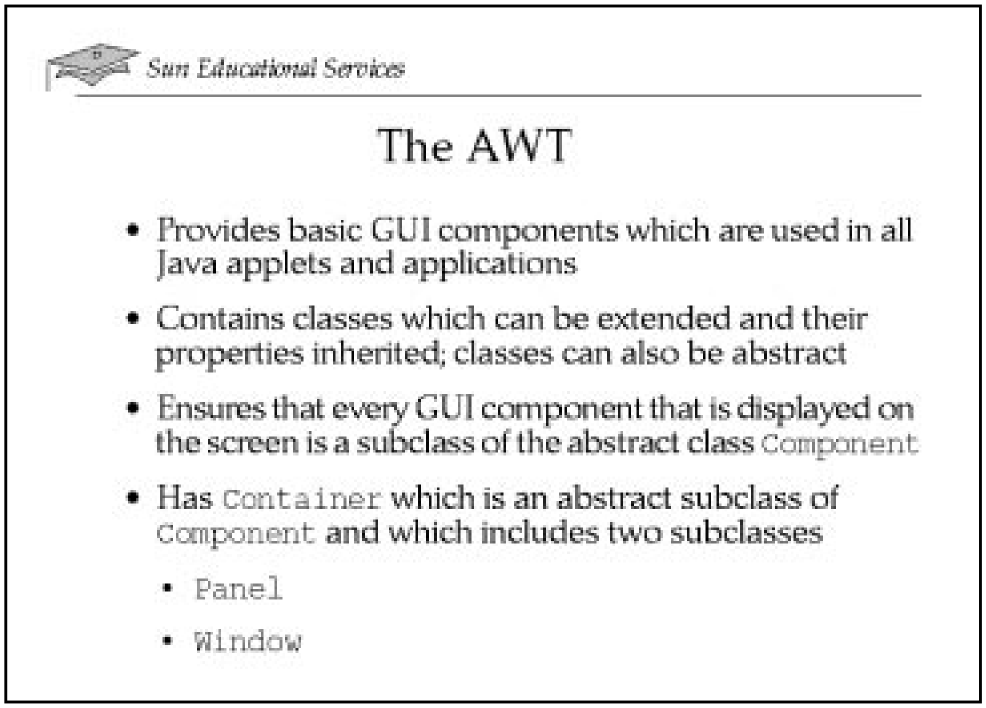
- The Java programming language is a platform-independent programming language. GUI environments are usually platform dependent. How does Java technology approach this subject in order to make the GUI platform independent?

✓ ***Most interactive user input and output will be done through a graphical user interface (GUI). The Java platform provides a number of classes in the AWT (Abstract Window Toolkit) package to make programming and setup of GUIs very easy. The AWT contains classes of*** `Components` ***from which GUIs are pieced together, and contains other classes which deal with other aspects such as color, font, and layout of components. This module explains how GUIs are put together and formatted, and describes some of the*** `Components` ***from which GUIs are built.***

## Objectives

Upon completion of this module, you should be able to

- Describe the AWT package and its components
- Define the terms *containers*, *components* and *layout managers*, and how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately
- Describe how complex layouts with nested containers work
- In a Java software program, identify the following:
  - Containers
  - The associated layout managers
  - The layout hierarchy of all components



Sun Educational Services

## The AWT

- Provides basic GUI components which are used in all Java applets and applications
- Contains classes which can be extended and their properties inherited; classes can also be abstract
- Ensures that every GUI component that is displayed on the screen is a subclass of the abstract class `Component`
- Has `Container` which is an abstract subclass of `Component` and which includes two subclasses
  - `Panel`
  - `Window`

## *The AWT*

The AWT provides basic GUI components which are used in Java applets and applications. The AWT provides a machine-independent interface for applications. This ensures that what appears on one computer will be comparable to what appears on another.

Before looking at the AWT, briefly review the object hierarchy. Remember that super classes can be extended and their properties inherited. Also, classes can be abstract, meaning that they are templates that are subclassed to provide concrete implementations of the class.

Every GUI component that appears on the screen is a subclass of the abstract class `Component`. That is, every graphics object which extends from the `Component` class shares a number of methods and instance variables that allow them to be operated.

## The AWT

`Container` is an abstract subclass of `Component`, which allows other components to be nested inside it. These components can also be containers allowing other components to be nested inside it, thus creating a complete hierarchical structure. Containers are helpful in arranging GUI components on the screen. A `Panel` is the simplest concrete subclass of `Container`. Another subclass of `Container` is a `Window`.

# The `java.awt` Package

The `java.awt` package contains classes that generate GUI components. A basic overview of this package is shown in Figure 8-1. The classes shown in bold type highlight the main focus of this module.

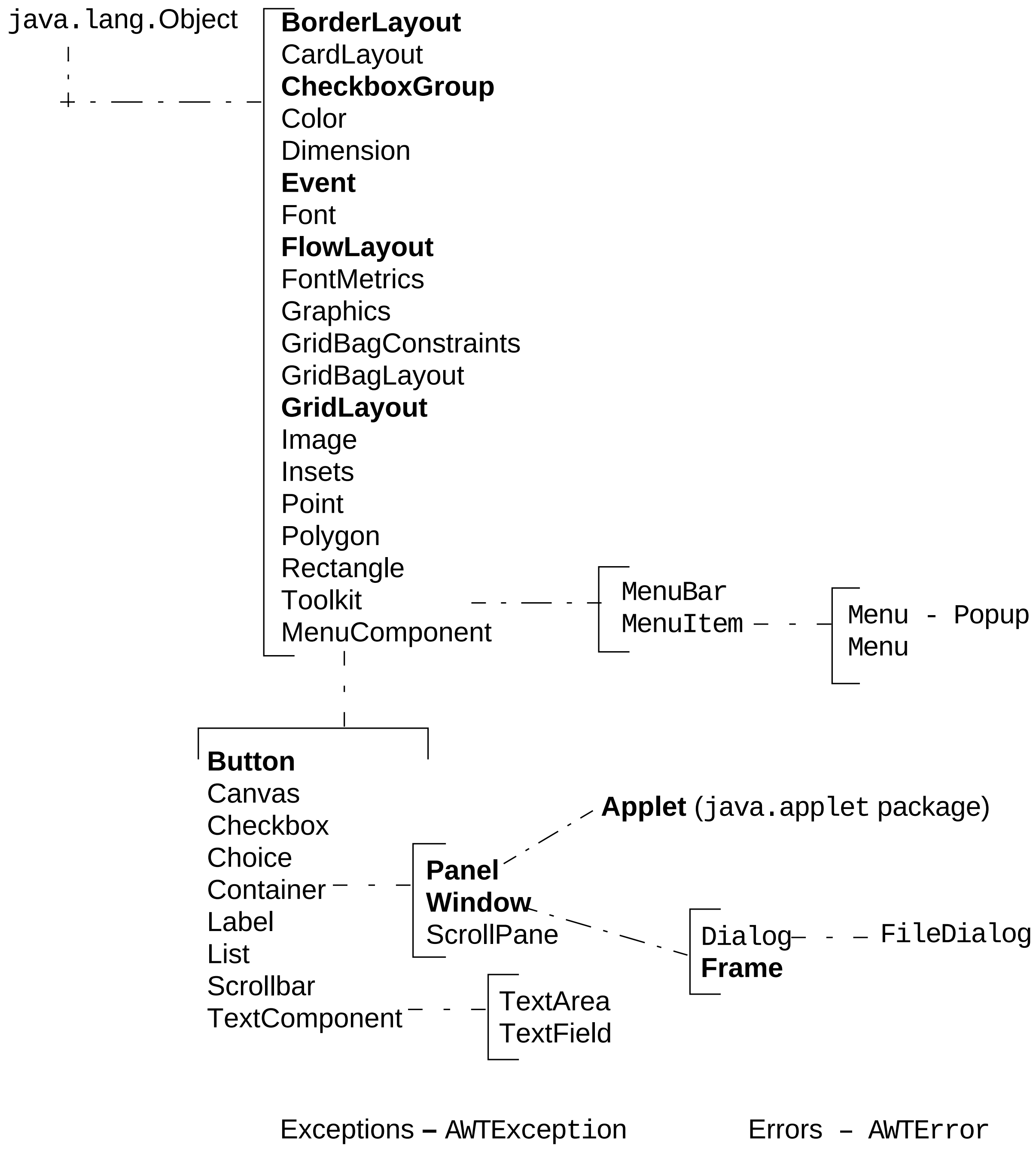
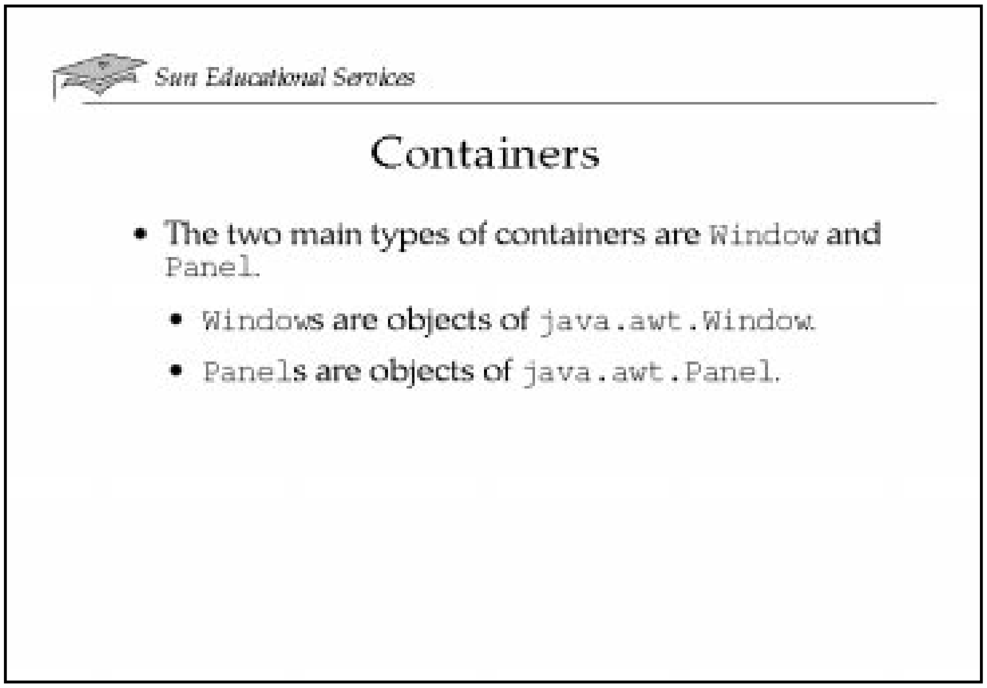


**Figure 8-1** `java.awt` Package



Sun Educational Services

Containers

- The two main types of containers are `Window` and `Panel`.
  - `Windows` are objects of `java.awt.Window`.
  - `Panels` are objects of `java.awt.Panel`.

## Building Graphical User Interfaces

### Containers

There are two main types of containers: `Window` and `Panel`.

A `Window` is an object of `java.awt.Window`. A `Window` is a free-standing native window on the display that is independent of other containers.

There are two important types of `Window`: `Frame` and `Dialog`. `Frame` is a window with a title and resize corners. `Dialog` does not have a menu bar. Although it can be moved, it cannot be resized.

## Building Graphical User Interfaces

### Containers

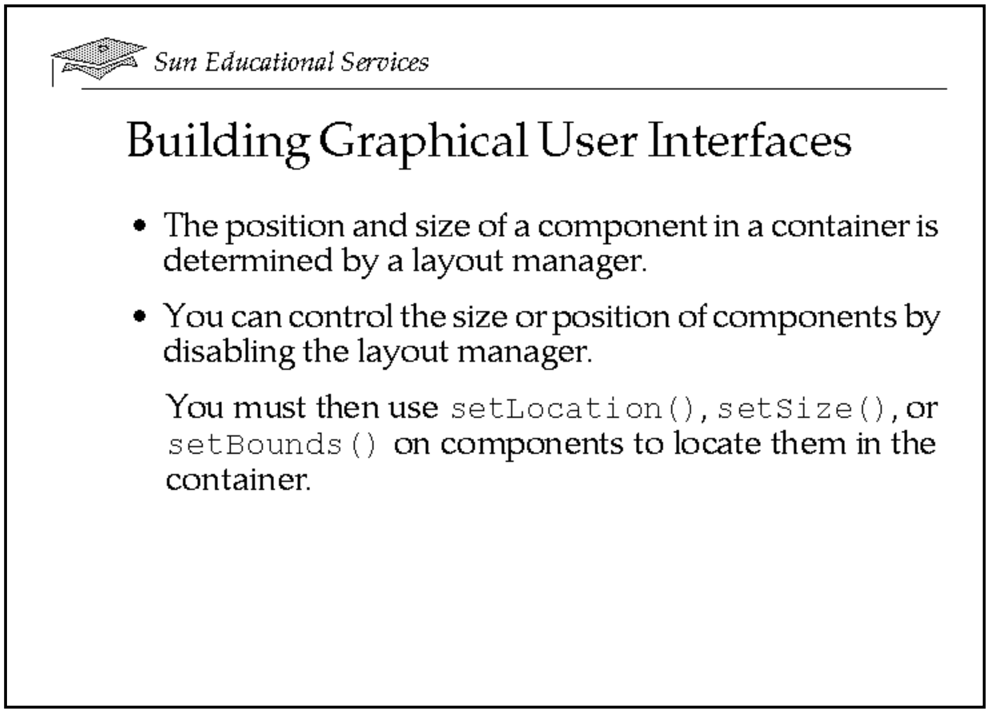
Panel is an object of java.awt.Panel. Panel is contained within another Container, or inside a Web browser's window. Panel identifies a rectangular area into which other components can be placed. Panel must be placed into a Window (or a subclass of Window) in order to be displayed.

---

**Note –** The fact that a container can hold not only components, but also other containers, is critical and fundamental to building complex layouts.

---

ScrollPane is also a subclass of Window. It is discussed in Module 10, "The AWT Component Library."


Sun Educational Services

## Building Graphical User Interfaces

- The position and size of a component in a container is determined by a layout manager.
- You can control the size or position of components by disabling the layout manager.

  You must then use `setLocation()`, `setSize()`, or `setBounds()` on components to locate them in the container.

## *Building Graphical User Interfaces*

### *Positioning Components*

The position and size of a component in a container is determined by a layout manager. A container keeps a reference to a particular instance of a layout manager. When the container needs to position a component it invokes the layout manager to do so. The same delegation occurs when deciding on the size of a component. The layout manager takes full control over all of the components within the container. It is responsible for computing and defining the preferred size of the object in the context of the actual screen size.

✓ ***The preferred size expresses how big a component will be displayed. For example, the preferred size of a button would be the size of the label text plus the border space and the shadowed decorations that mark the boundary of the button. Note that the preferred size is platform dependent.***

# Building Graphical User Interfaces

## Component Sizing

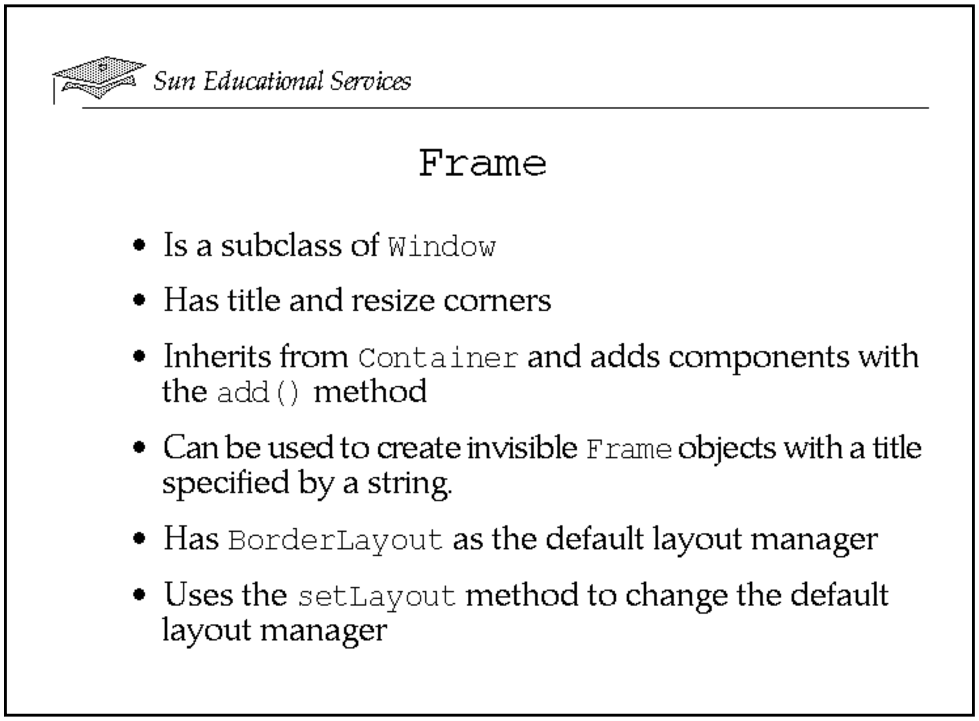
Because the layout manager generally is responsible for the size and position of components on its container, you should not normally attempt to set the size or position of components yourself. If you try to do so (using any of the methods `setLocation()`, `setSize()` or `setBounds()`), the layout manager can override your decision.

If you must control size or position of components in a way that cannot be done using the standard layout managers, it is possible to disable the layout manager by issuing the following method call to your container:

```
    cont.setLayout(null);
```

After this step you must use `setLocation()`, `setSize()` or `setBounds()` on all components to locate them in the container.

Be aware that this approach results in platform-dependent layouts due to the differences between window systems and font sizes. A better approach is to create a new subclass of `LayoutManager`.


Sun Educational Services

## Frame

- Is a subclass of `Window`
- Has title and resize corners
- Inherits from `Container` and adds components with the `add()` method
- Can be used to create invisible `Frame` objects with a title specified by a string.
- Has `BorderLayout` as the default layout manager
- Uses the `setLayout` method to change the default layout manager

## *Frames*

`Frame` is a subclass of `Window`. It is a `Window` with a title and resize corners. `Frame` inherits from `java.awt.Container` so you can add components to a `Frame` using the `add()` method. The default layout manager for a `Frame` is `BorderLayout`. This can be changed using the `setLayout()` method.

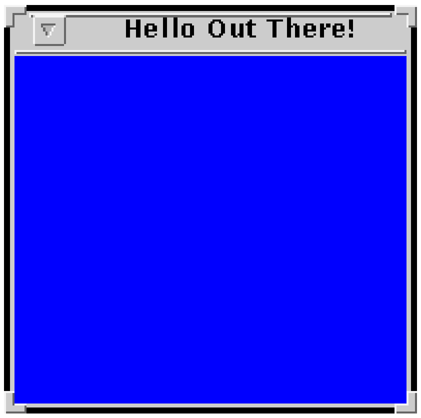
The constructor `Frame(String)` in the `Frame` class creates a new, invisible `Frame` object with the title specified by `String`. Add all the components to the `Frame` while it is still invisible.

## Frames

The following program creates a `Frame` which has a specific title, size, and background color (Figure 8-2).

```
import java.awt.*;

public class MyFrame extends Frame {

  public MyFrame (String str) {
    super(str);
  }

  public static void main (String args[]) {
    MyFrame fr = new MyFrame("Hello Out There!");
    fr.setSize(500,500);
    fr.setBackground(Color.blue);
    fr.setVisible(true);
  }
}
```

✓ ***You can use, rather than extend, a*** `Frame` ***or put*** `main()` ***in a separate class. This code was chosen for its simplicity and it is not a good object-oriented design.***



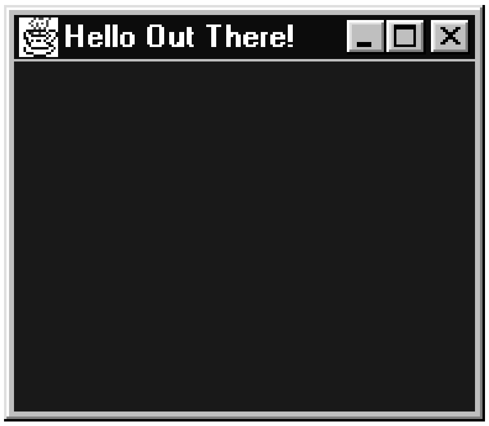
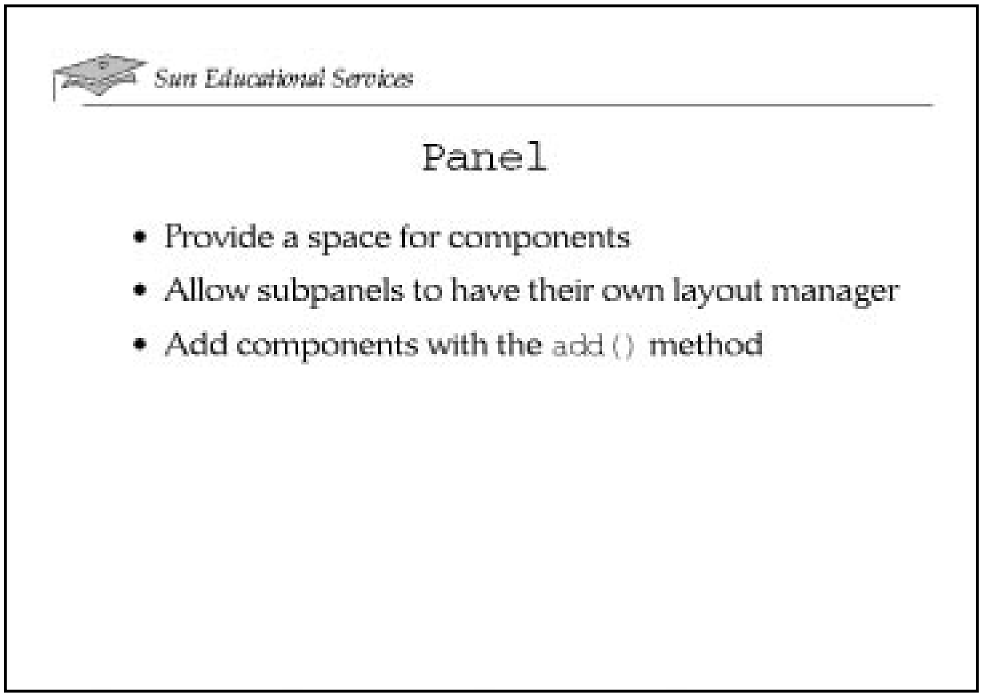


**Figure 8-2** Example `Frame`

---

**Note –** A `Frame` must be made visible (via a call to `setVisible(true)`) and its size defined (via a call to `setSize()` or `pack()`) before it will be displayed on-screen.

---



Sun Educational Services

## Panel

- Provide a space for components
- Allow subpanels to have their own layout manager
- Add components with the add() method

## *Panels*

`Panel`, like `Frame`, provides the space for you to attach any GUI component, including other panels. Each `Panel`, which inherits from `java.awt.Container`, can have its own layout manager.

Once a `Panel` is created, it must be added to a `Window` or `Frame` in order to be visible. This is done using the `add()` method of the `Container` class.

The following program creates a small yellow `Panel`, and adds it to a `Frame` (Figure 8-3).

## Panels

```
import java.awt.*;

public class FrameWithPanel extends Frame {

  // Constructor
  public FrameWithPanel (String str) {
    super(str);
  }

  public static void main (String args[]) {
    FrameWithPanel fr =
          new FrameWithPanel("Frame with Panel");
    Panel pan = new Panel();

    fr.setSize(200,200);
    fr.setBackground(Color.blue);
    fr.setLayout(null);  // Override default layout mgr

    pan.setSize(100,100);
    pan.setBackground(Color.yellow);

    fr.add(pan);
    fr.setVisible(true);
  }
}
```

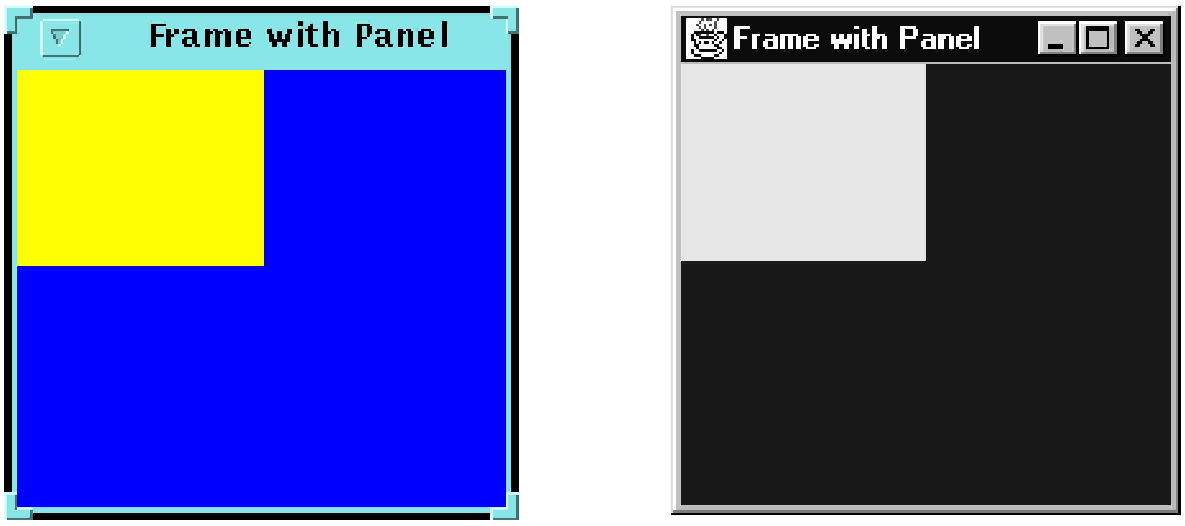
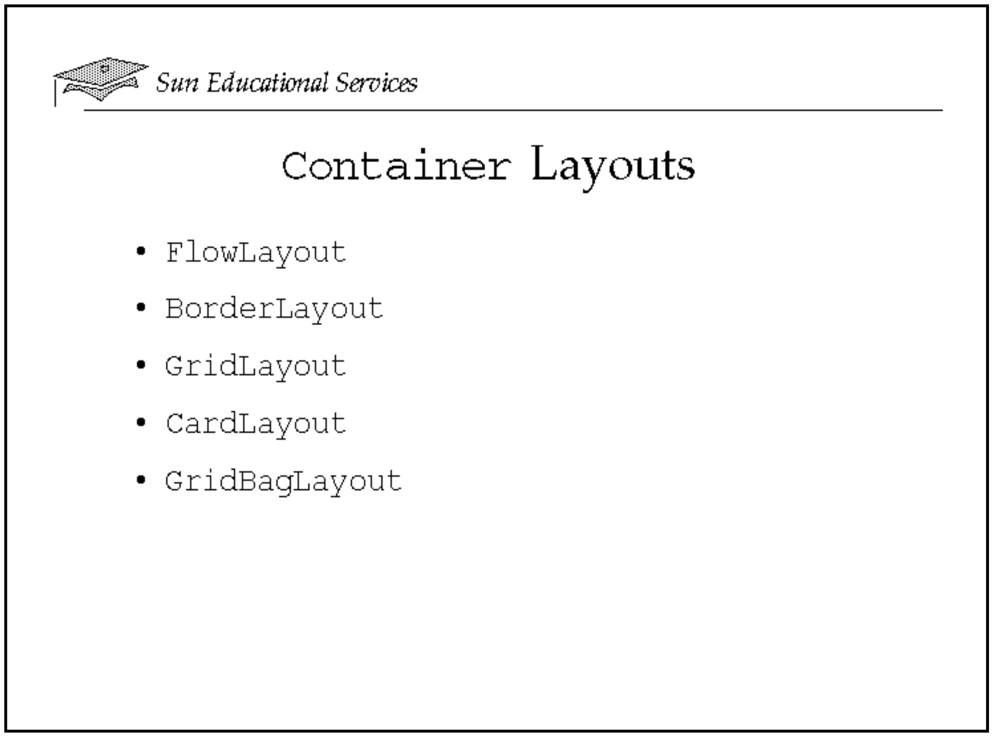


**Figure 8-3** Example Panel


Sun Educational Services

## Container Layouts

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

## Container Layouts

The layout of components in a container is usually governed by a layout manager. Each container (such as `Panel` or `Frame`) has a default layout manager associated with it, which can be changed by calling `setLayout()`.

The layout manager is responsible for deciding the layout policy and size of each of its container's child components.

✓ ***The layout manager gives first preference to the layout policy. If honoring a component's preferred size means violating the layout policy, the layout manager overrules the component's preferred size.***

# Container Layouts

## Layout Managers

The following layout managers are included with the Java programming language:

- `FlowLayout` – The default layout manager of `Panel` and `Applet`
- `BorderLayout` – The default layout manager of `Window`, `Dialog`, and `Frame`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

---

**Note –** The `GridBagLayout` manager is not discussed in-depth in this module.

---

✓ ***`GridBagLayout` manager is examined in Appendix A.***

## Container Layouts

### Default Layout Managers

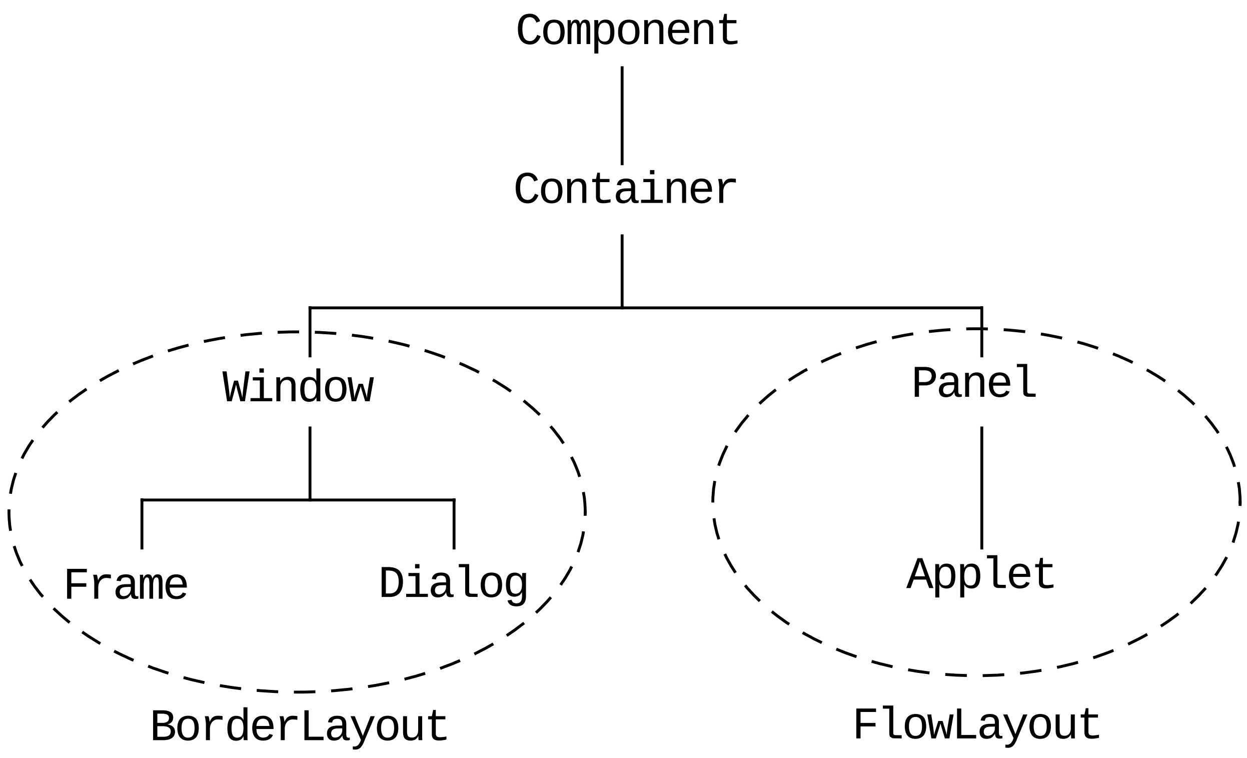


**Figure 8-4** Layout Managers

## *A Simple* `FlowLayout` *Example*

This sample code demonstrates several important points, which will be discussed in the following sections.

```
import java.awt.*;

public class ExGui {
  private Frame f;
  private Button b1;
  private Button b2;

  public void go() {
    f = new Frame("GUI example");
    f.setLayout(new FlowLayout());
    b1 = new Button("Press Me");
    b2 = new Button("Don't press Me");
    f.add(b1);
    f.add(b2);
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui guiWindow = new ExGui();
    guiWindow.go();
  }
}
```

This code creates

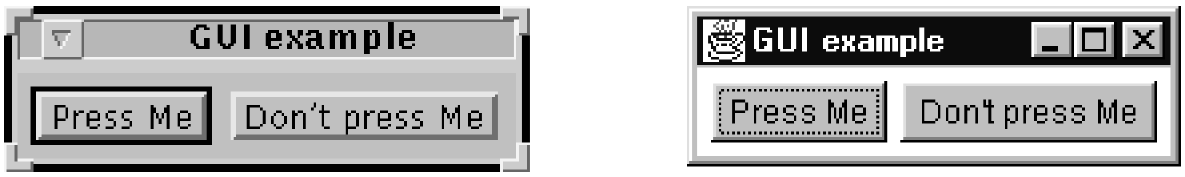


**Figure 8-5** Example of `FlowLayout`

## *A Simple* `FlowLayout` *Example*

### *The* `main()` *Method*

The `main()` method in line 8 of this example does two things. First, it creates an instance of the `ExGui` object. Recall that until an instance exists, there are no real data items called `f`, `b1,` and `b2` for use. Second, when the data space has been created, `main()` calls the instance method `go()` in the context of that instance. In `go()`, the real action occurs.

### `new Frame("GUI Example")`

This method creates an instance of the class `java.awt.Frame`. `Frame` in the Java programming language is a top-level window, with a title bar—defined by the constructor argument; "GUI Example" in this case—resize handles, and other decorations, according to local conventions. `Frame` has a 0x0 size and currently is not visible.

### `f.setLayout(new FlowLayout())`

This method creates an instance of the flow layout manager, and installs it in the `Frame`. The default layout manager for every frame, `BorderLayout`, is not used for this example. The `FlowLayout` manager is the simplest in the AWT and positions components somewhat like words on a page, line by line. Note that the `FlowLayout` centers each line by default.

### `new Button("Press Me")`

This method creates an instance of the class `java.awt.Button`. A button is the standard push button taken from the local window toolkit. The button label is defined by the string argument to the constructor.

## *A Simple* `FlowLayout` *Example*

### `f.add(b1)`

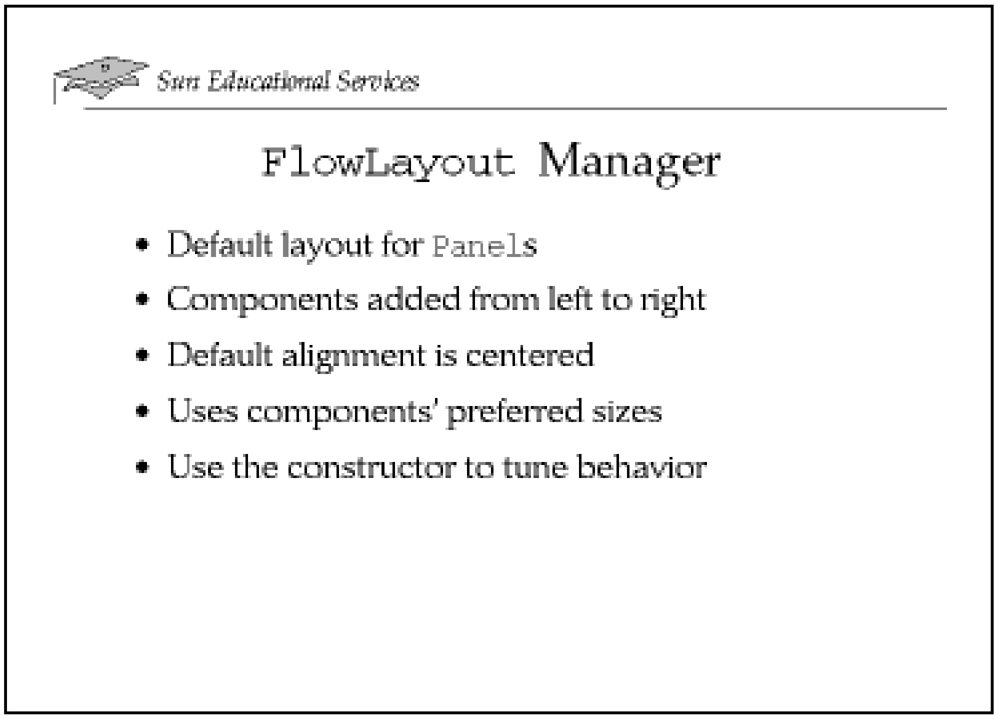
This method tells `Frame f` (which is a container) that it is to contain the component b1. The size and position of b1 are under the control of the frame's layout manager from this point onward.

### `f.pack()`

This method tells the frame to set a size that "neatly encloses" the components that it contains. In order to determine what size to use for the `Frame`, `f.pack()`queries the layout manager, which is responsible for the size and position of all the components within the `Frame`.

### `f.setVisible(true)`

This method causes the `Frame`, and all its contents, to become visible to the user.


Sun Educational Services

FlowLayout Manager

- Default layout for Panels
- Components added from left to right
- Default alignment is centered
- Uses components' preferred sizes
- Use the constructor to tune behavior

## *Layout Managers*

### FlowLayout *Manager*

The FlowLayout used in the example for this topic positions components on a line-by-line basis. Each time a line is filled, a new line is started.

Unlike other layout managers, the FlowLayout manager does not constrain the size of the components it manages, but instead allows them to have their preferred size.

FlowLayout constructor arguments allow you to flush the components to the left or to the right if you prefer that to the default centering behavior.

Gaps can be specified if you want to create a larger minimum space between the components.

## `FlowLayout` *Manager (Continued)*

When the area being managed by a FlowLayout is resized by the user, the layout can change. (See Figure 8-6):

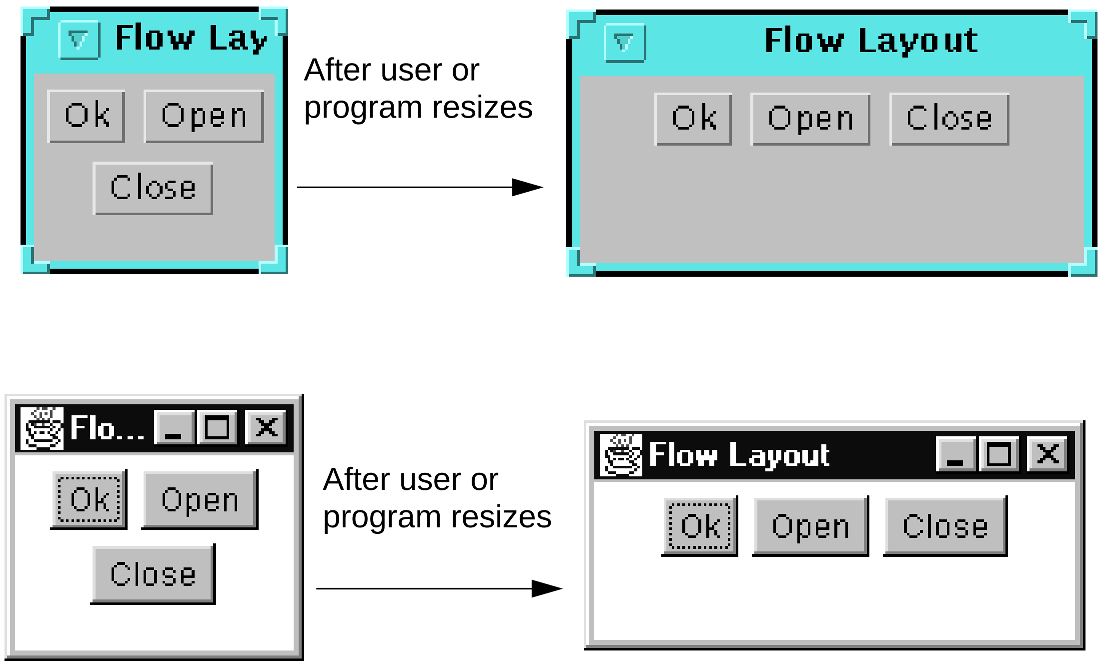


**Figure 8-6** `FlowLayout` Resizing

## Layout Managers

### `FlowLayout` *Manager (Continued)*

The following are examples of how to create `FlowLayout` objects and install them using the `setLayout()` method of class `Container`

```
setLayout(new FlowLayout(
                int align,int hgap,int vgap));
```

The value of `align` must be `FlowLayout.LEFT`, `FlowLayout.RIGHT`, or `FlowLayout.CENTER`. For example:

```
setLayout(new FlowLayout(FlowLayout.RIGHT, 20, 40));
```

The following constructs and installs a new `FlowLayout` with the specified alignment and a default five-unit horizontal and vertical gap.

```
setLayout(new FlowLayout(int align);
setLayout(new FlowLayout(FlowLayout.LEFT));
```

The following constructs and installs a new `FlowLayout` with centered alignment and a default five-unit horizontal and vertical gap:

```
setLayout(new FlowLayout());
```

### `FlowLayout` *Manager (Continued)*

The following sample code adds several buttons to a flow layout on a `Frame`:

```
import java.awt.*;

public class MyFlow {
  private Frame f;
  private Button button1, button2, button3;

  public void go() {
    f = new Frame("Flow Layout");
    f.setLayout(new FlowLayout());
    button1 = new Button("Ok");
    button2 = new Button("Open");
    button3 = new Button("Close");
    f.add(button1);
    f.add(button2);
    f.add(button3);
    f.setSize(100,100);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    MyFlow mflow = new MyFlow();
    mflow.go();
  }
}
```

## *Layout Managers*

### `BorderLayout` *Manager*

The `BorderLayout` manager provides a more complex scheme for placing your components within a container. It is the default layout manager for `Frame` and `Dialog`. The `BorderLayout` manager contains five distinct areas: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`, indicated by `BorderLayout.NORTH`, and so on.

`NORTH` occupies the top of a frame, `EAST` occupies the right side, and so on. The `CENTER` area represents everything left over once the `NORTH`, `SOUTH`, `EAST`, and `WEST` areas are filled. When the `Window` is stretched vertically, the `EAST`, `WEST` and `CENTER` regions are stretched, whereas when the window is stretched horizontally, the `NORTH`, `SOUTH` and `CENTER` regions are stretched.

---

**Note –** The relative positions of the buttons do not change as the window resized, but the sizes of the buttons do change.

---

## `BorderLayout` *Manager (Continued)*

The following line:

```
setLayout(new BorderLayout());
```

constructs and installs a new `BorderLayout` with no gaps between the components.

The line

```
setLayout(new BorderLayout(int hgap, int vgap);
```

constructs and installs a `BorderLayout` with the specified gaps between components as specified by `hgap` and `vgap`.

Components must be added to named regions in the `BorderLayout` manager; otherwise, they will not be visible. Spell the region names correctly; especially if you choose not to use constants (for instance, `add(button,"Center")` instead of `add(button, BorderLayout.CENTER).` Spelling and capitalization are crucial.

You can use a `BorderLayout` manager to produce layouts with elements that stretch in one direction, the other direction, or both, when resized.

---

**Note –** `NORTH`, `SOUTH` and `CENTER` regions change if the window is resized horizontally, and `EAST`, `WEST`, and `CENTER` regions change if the window is resized vertically.

---

If you leave a region of a `BorderLayout` unused, it behaves as if its preferred size were zero by zero. The `CENTER` region will still appear as background even if it contains no components.

## Layout Managers

### `BorderLayout` *Manager (Continued)*

You can add only a single component to each of the five regions of the `BorderLayout` manager. If you try to add more than one, only the last one added will be visible. A later example shows how you can use intermediate containers to allow more than one component to be laid out in the space of a single `BorderLayout` manager region.

---

**Note –** The layout manager honors the preferred height of the `NORTH` and `SOUTH` components, but forces them to be as wide as the container. In the case of the `EAST` and `WEST` components, the preferred width is honored and the height is constrained.

---

### `BorderLayout` *Manager (Continued)*

The following code is a modification of the previous example and demonstrates the behavior of the `BorderLayout` manager. You can set the layout to use `BorderLayout` by using the `setLayout()` method inherited from the class `Container`.

```
import java.awt.*;

public class ExGui2 {
  private Frame f;
  private Button bn, bs, bw, be, bc;

  public void go() {
    f = new Frame("Border Layout");
    bn = new Button("B1");
    bs = new Button("B2");
    bw = new Button("B3");
    be = new Button("B4");
    bc = new Button("B5");

    f.add(bn, BorderLayout.NORTH);
    f.add(bs, BorderLayout.SOUTH);
    f.add(bw, BorderLayout.WEST);
    f.add(be, BorderLayout.EAST);
    f.add(bc, BorderLayout.CENTER);

    f.setSize(200,200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui2 guiWindow2 = new ExGui2();
    guiWindow2.go();
  }
}
```

## `BorderLayout` *Manager (Continued)*

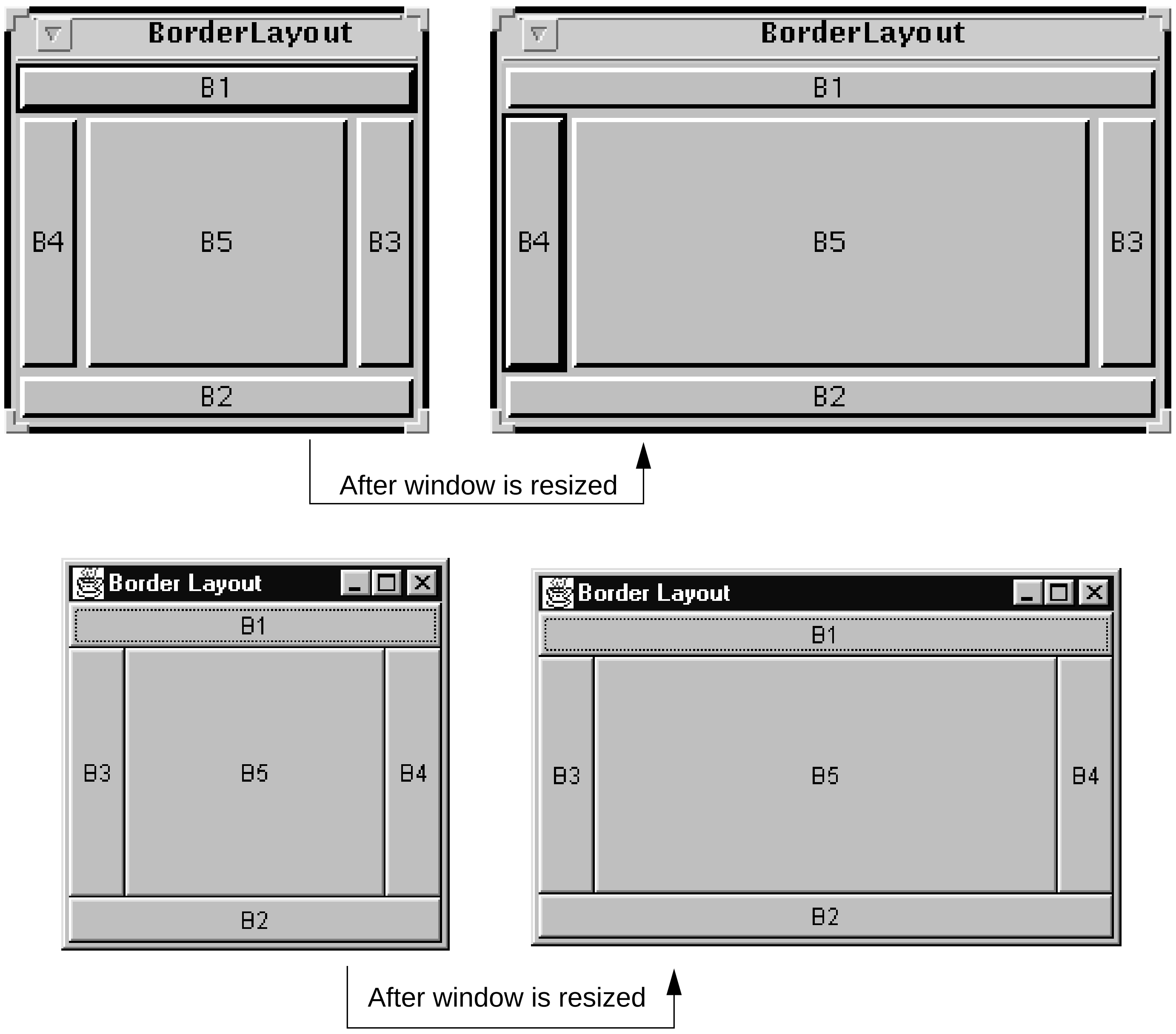
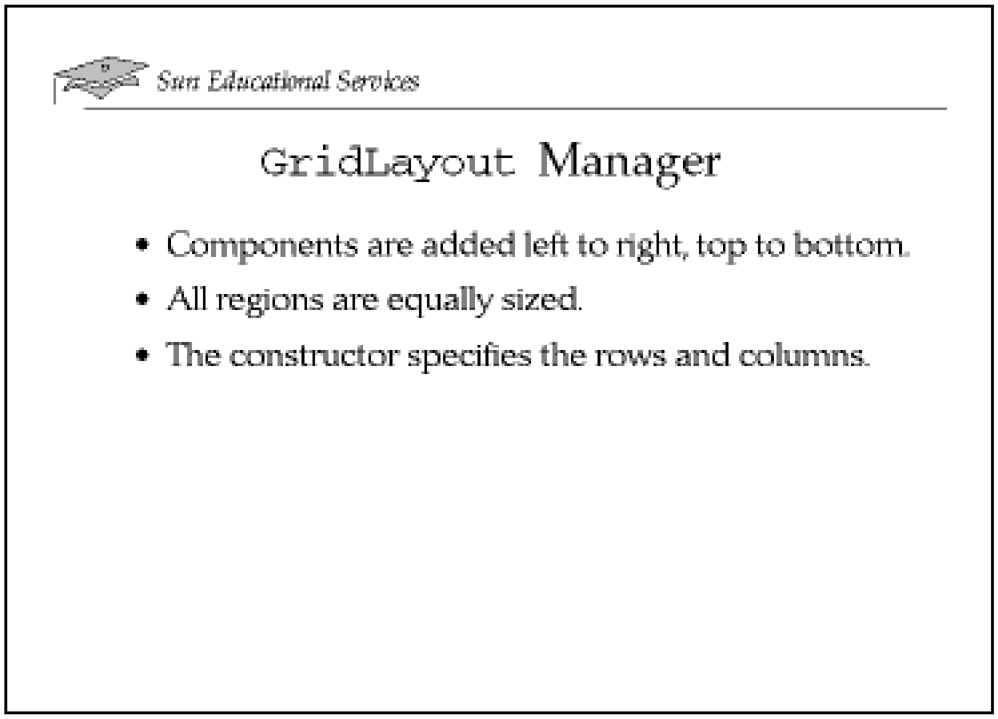


**Figure 8-7** Example of `BorderLayout`

✓ `BorderLayout` ***regions can also be spelled out (such as,*** `f.add(bn, "North")`***). However, runtime exceptions can occur for misspellings (including capitalization mistakes). Using predefined constants such as*** `borderLayout.NORTH` ***is preferred, because errors with them will be caught at compile time instead of runtime.***



Sun Educational Services

GridLayout Manager

- Components are added left to right, top to bottom.
- All regions are equally sized.
- The constructor specifies the rows and columns.

## *Layout Managers*

### `GridLayout` *Manager*

The `GridLayout` manager provides flexibility for placing components. You create the manager with a number of rows and columns. Components then fill up the cells defined by the manager. For example, a `GridLayout` with three rows and two columns created by the statement `new GridLayout(3, 2)` would create six cells.

As with the `BorderLayout` manager, the relative position of components does not change as the area is resized. Only the sizes of the components change.

The `GridLayout` manager always ignores the component's preferred size. The width of all cells is identical and is determined by dividing the available width by the number of cells. Similarly, the height of all cells is determined by dividing the available height by the number of rows.

## `GridLayout` *Manager (Continued)*

The order in which components are added to the grid determines the cell that they occupy. Lines of cells are filled left to right, like text, and the “page” is filled with lines from top to bottom.

The line

```
setLayout(new GridLayout());
```

creates and installs a `GridLayout` with a default of one column per component in a single row.

The line

```
setLayout(new GridLayout(int rows, int cols);
```

creates and installs a grid layout with the specified number of rows and columns. All components in the layout are given equal size.

The following lines:

```
setLayout(new GridLayout(
    int rows, int cols, int hgap, int vgap);
```

create and install a `GridLayout` with the specified number of rows and columns. All components in the layout are given equal size. `hgap` and `vgap` specify the respective gaps between components. Horizontal gaps are placed between each of the columns. Vertical gaps are placed between each of the rows.

---

**Note –** One, but not both, of the rows and columns can be zero. The zero value parameter is treated as a flexible guideline. For example, zero rows and two columns mean that there will always be two columns, but as few or many rows as are needed to hold all of the added components.

---

## GridLayout *Manager (Continued)*

The following code produces the objects displayed in Figure 8-8.

```
import java.awt.*;

public class GridEx {
  private Frame f;
  private Button b1, b2, b3, b4, b5, b6;

  public void go() {
    f = new Frame("Grid Example");
    f.setLayout (new GridLayout(3,2));

    b1 = new Button("1");
    b2 = new Button("2");
    b3 = new Button("3");
    b4 = new Button("4");
    b5 = new Button("5");
    b6 = new Button("6");

    f.add(b1);
    f.add(b2);
    f.add(b3);
    f.add(b4);
    f.add(b5);
    f.add(b6);

    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    GridEx grid = new GridEx();
    grid.go();
  }
}
```

## `GridLayout` *Manager (Continued)*

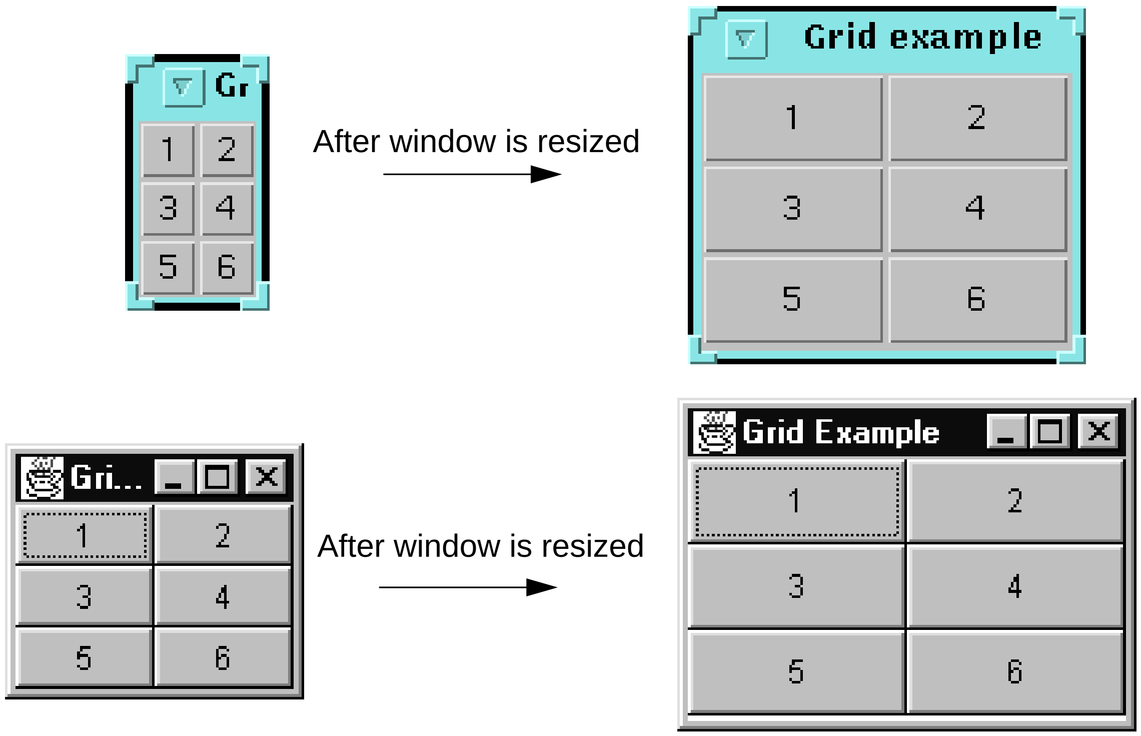


**Figure 8-8** Example of `GridLayout`

## CardLayout *Manager*

The CardLayout manager enables you to treat the interface as a series of cards, one of which is viewable at any one time. The add() method is used to add cards to a CardLayout. The add() method takes a String as an argument and identifies the Panel in the program. The CardLayout manager's show() method switches to a new card on request. The example for this topic demonstrates a single Frame that shows five different Panels with each mouse click. A mouse click in the left Panel switches the view to the right Panel, and so on.

```
import java.awt.*;
import java.awt.event.*;

public class CardTest implements MouseListener {
  private Panel p1, p2, p3, p4, p5;
  private Label lb1, lb2, lb3, lb4, lb5;

  // Declare a CardLayout object to call its methods.
  private CardLayout myCard;
  private Frame f;

  public void go() {
    f = new Frame ("Card Test");
    myCard = new CardLayout();
    f.setLayout(myCard);

    // Create the panels that I want
    // to use as cards.
    p1 = new Panel();
    p2 = new Panel();
    p3 = new Panel();
    p4 = new Panel();
    p5 = new Panel();

    // Create a label to attach to each panel, and
    // change the color of each panel, so they are
    // easily distinguishable

    lb1 = new Label("This is the first Panel");
    p1.setBackground(Color.yellow);
    p1.add(lb1);
```

## `CardLayout` *Manager (Continued)*

```

    lb2 = new Label("This is the second Panel");
    p2.setBackground(Color.green);
    p2.add(lb2);

    lb3 = new Label("This is the third Panel");
    p3.setBackground(Color.magenta);
    p3.add(lb3);

    lb4 = new Label("This is the fourth Panel");
    p4.setBackground(Color.white);
    p4.add(lb4);

    lb5 = new Label("This is the fifth Panel");
    p5.setBackground(Color.cyan);
    p5.add(lb5);

    // Set up the event handling here.
    p1.addMouseListener(this);
    p2.addMouseListener(this);
    p3.addMouseListener(this);
    p4.addMouseListener(this);
    p5.addMouseListener(this);

    // Add each panel to my CardLayout
    f.add(p1, "First");
    f.add(p2, "Second");
    f.add(p3, "Third");
    f.add(p4, "Fourth");
    f.add(p5, "Fifth");

    // Display the first panel.
    myCard.show(f, "First");

    f.setSize(200,200);
    f.setVisible(true);
  }

```

## `CardLayout` *Manager (Continued)*

```
  public void mousePressed(MouseEvent e) {
    myCard.next(f);
  }

  public void mouseReleased(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }

  public static void main (String args[]) {
    CardTest ct = new CardTest();
    ct.go();
  }
}
```

This code creates

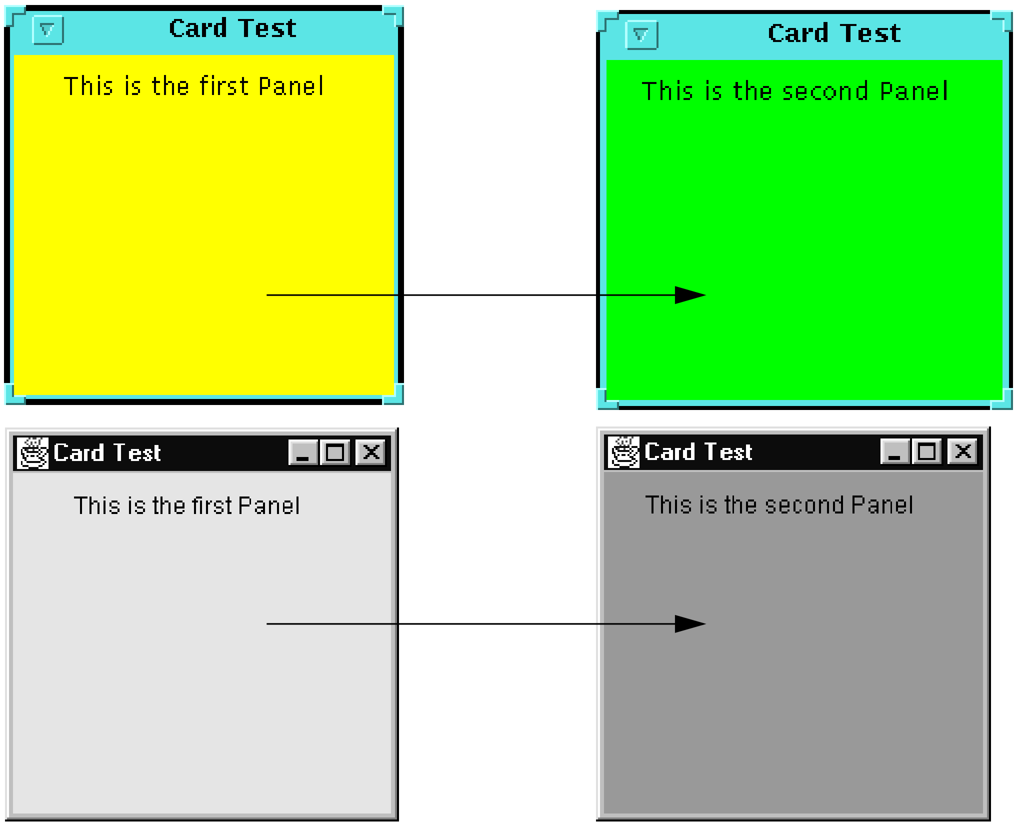
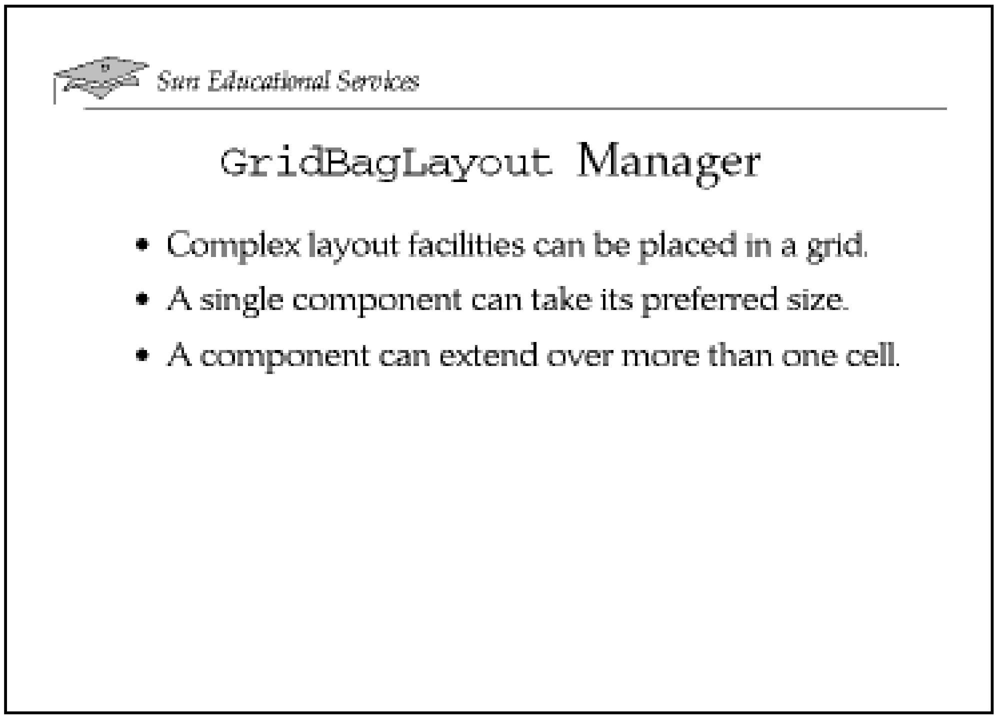


**Figure 8-9** Example of `CardLayout`



Sun Educational Services

GridBagLayout Manager

- Complex layout facilities can be placed in a grid.
- A single component can take its preferred size.
- A component can extend over more than one cell.

## *Layout Managers*

### GridBagLayout *Manager*

In addition to the FlowLayout, BorderLayout, GridLayout, and CardLayout managers, the core java.awt also provides GridBagLayout manager.

The GridBagLayout manager provides complex layout facilities based on a grid but allowing single components to take their preferred sizes within a cell rather than fill the whole cell. The GridbagLayout manager also allows a single component to extend over more than one cell.

✓ GridBagLayout ***is very complex to understand and manipulate. The*** GridBagLayout ***example from the API is covered in Appendix A. If you want to cover it as a module, you should do so now.***

## Creating Panels and Complex Layouts

The following program uses a `Panel` to allow two buttons to be placed in the `NORTH` region of a border layout. This kind of nesting is fundamental to complex layouts. Note that the `Panel` is treated just like another component as far as the `Frame` itself is concerned.

```
import java.awt.*;

public class ExGui3 {
  private Frame f;
  private Panel p;
  private Button bw, bc;
  private Button bfile, bhelp;

  public void go() {
    f = new Frame("GUI example 3");
    bw = new Button("West");
    bc = new Button("Work space region");
    f.add(bw, BorderLayout.WEST);
    f.add(bc, BorderLayout.CENTER);
    p = new Panel();
    bfile = new Button("File");
    bhelp = new Button("Help");
    p.add(bfile);
    p.add(bhelp);
    f.add(p, BorderLayout.NORTH);
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    ExGui3 gui = new ExGui3();
    gui.go();
  }
}
```

## Creating Panels and Complex Layouts

When the example is run, the resulting display looks like this:

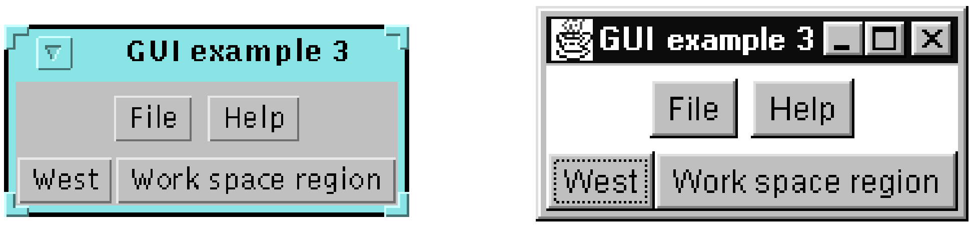


**Figure 8-10** Combining Layout Managers

If the window is resized, it looks like this:

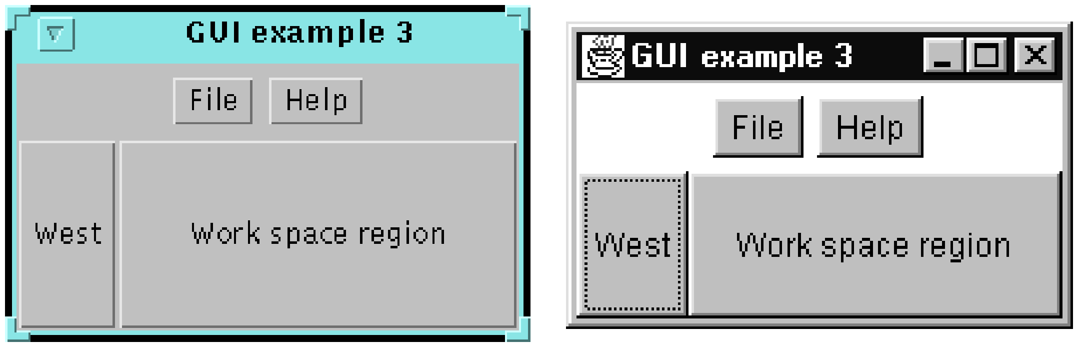
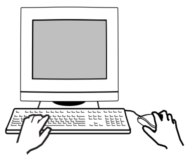


**Figure 8-11** Resized Combination Layouts

The `NORTH` region of the `BorderLayout` is now effectively holding two buttons. In fact, it holds only the single `Panel` but that panel contains the two buttons.

The size and position of the `Panel` is determined by the `BorderLayout` manager, and the preferred size of a `Panel` is determined from the preferred size of the components in that `Panel`. The size and position of the buttons in the `Panel` are controlled by the `FlowLayout` that is associated with the `Panel` by default.

# Exercise: Building Java GUIs



**Exercise objective –** In this lab you will develop two graphical user interfaces.

## Preparation

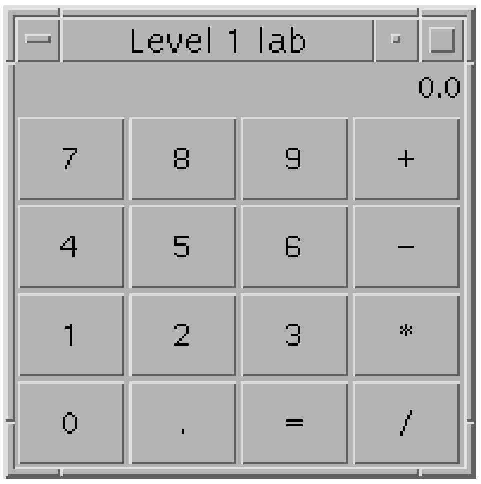
In order to successfully complete this lab, you must understand the purpose of a graphical user interface and know how to create one using layout managers.

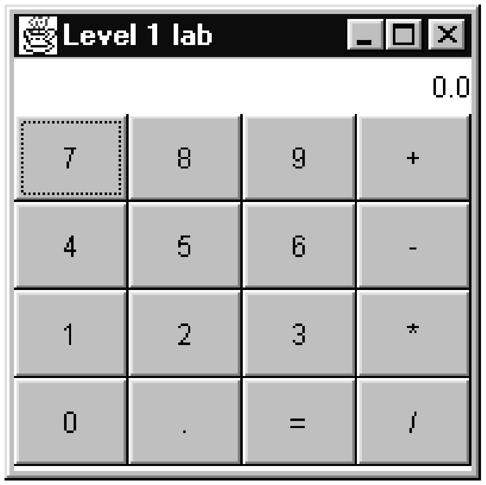
## Tasks

### Level 1 Lab: Create the Calculator GUI

Complete the following step:

1. Create the following GUI:





✓ ***Tell students that they can use either `Label` or `TextField` for the top area of the calculator.***

## Exercise: Building Java GUIs

### Level 3 Lab: Create the Account GUI

Perform the following step:

1. Create a GUI that will provide the front-end user interface for the `Teller.java` application created in Module 5. You might need to research some components that are not described in this module. (They are described later in this course.)

## Exercise: Building Java GUIs

### Exercise Summary

**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

❑ Describe the AWT package and its components

❑ Define the terms *containers*, *components* and *layout managers*, and how they work together to build a GUI

❑ Use layout managers

❑ Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout

❑ Add components to a container

❑ Use the `Frame` and `Panel` containers appropriately

❑ Describe how complex layouts with nested containers work

❑ In a Java program, identify the following:

- ▼ Containers
- ▼ The associated layout managers
- ▼ The layout hierarchy of all components

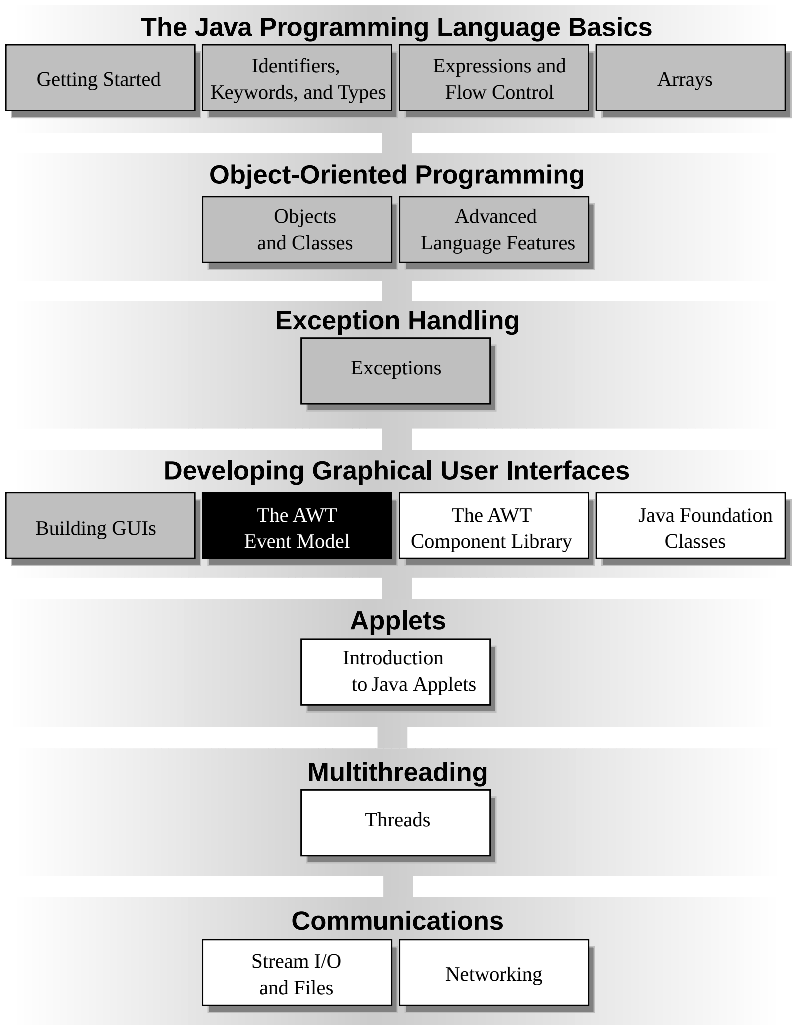
## Think Beyond

You now know how to display a GUI on the computer screen. What is needed to make the GUI useful?

# The AWT Event Model 9

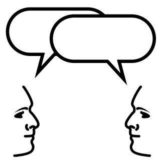
## Course Map

This module covers the event-based GUI user input mechanism.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following questions are relevant to the material presented in this module:

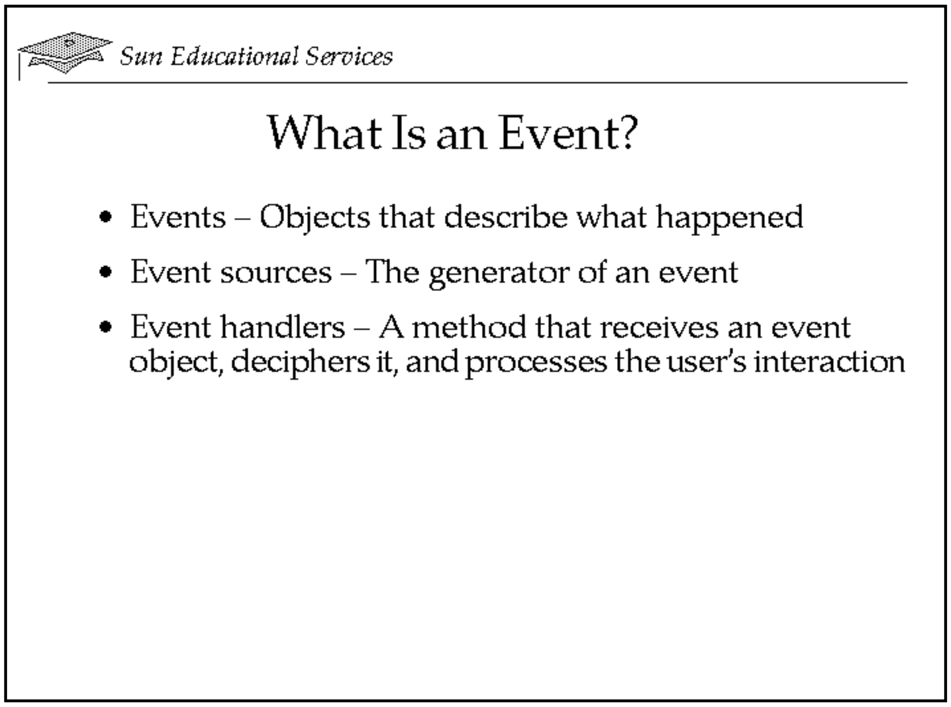
- What parts are required for a GUI to make it useful?
- How does a graphical program handle a mouse click or any other type of user interaction?

✓ ***Part of a GUI is receiving interactive user input. This input can be used, for example, to fill out a text form on the screen, register a mouse click on a map, or shut down a program. This module describes the event-driven mechanism provided by the AWT for receiving interactive user input.***

## Objectives

Upon completion of this module, you should be able to

- Define events and event handling
- Describe the differences between JDK1.0 and later event models
- Write code to handle events that occur in a GUI
- Describe the concept of adapter classes, including how and when to use them
- Determine the user action that originated the event from the event object details
- Create the appropriate interface and event handler methods for a variety of event types.
- Understand inner classes and anonymous classes



Sun Educational Services

### What Is an Event?

- Events – Objects that describe what happened
- Event sources – The generator of an event
- Event handlers – A method that receives an event object, deciphers it, and processes the user's interaction

## What Is an Event?

When the user performs an action at the user interface level (clicks a mouse or presses a key), this causes an *event* to be issued. Events are objects that describe what has happened. A number of different types of event classes exist to describe different categories of user action.

## What Is an Event?

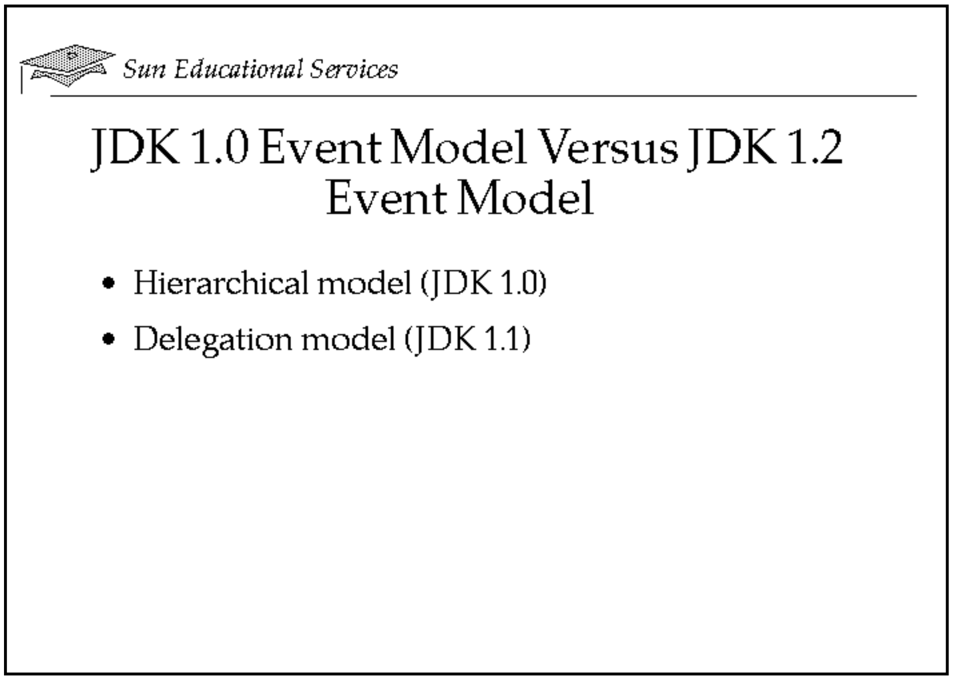
### Event Sources

An *event source* is the generator of an event. For example, a mouse click on a `Button` component will generate an `ActionEvent` with the button as the source. The `ActionEvent` instance is an object that contains information about the events that just took place. It contains:

- `getActionCommand()` – Returns the command name associated with the action.
- `getModifiers()` – Returns any modifiers held during the action.

### Event Handlers

An *event handler* is a method that receives an event object, deciphers it, and processes the user's interaction.


Sun Educational Services

JDK 1.0 Event Model Versus JDK 1.2 Event Model

- Hierarchical model (JDK 1.0)
- Delegation model (JDK 1.1)

## *JDK 1.0 Event Model Versus JDK 1.2 Event Model*

In JDK 1.1, significant changes in the way that events are received and processed were introduced. This section compares the previous event model (JDK 1.0) to the current event model (JDK 1.1 and JDK 1.2).

JDK 1.0 uses a hierarchical event model, while JDK 1.1 and beyond use a delegation event model.

### *Hierarchical Model (JDK 1.0)*

The hierarchical event model is based on containment. Events are sent to the component first, but then propagate up the containment hierarchy. Events that are not handled by the component will *automatically* continue to propagate to the component's container.

✓ ***These are containers, not the*** `Container` ***class. They extend the*** `Container` ***class.***

## JDK 1.0 Event Model Versus JDK 1.2 Event Model

For example, in Figure 9-1, a mouse click on the `Button` object (contained by a `Panel` on a `Frame`) sends an action event to the `Button` first. If it is not handled by the `Button`, the event is then sent to the `Panel`, and if it is not handled there, the event is sent to the `Frame`.

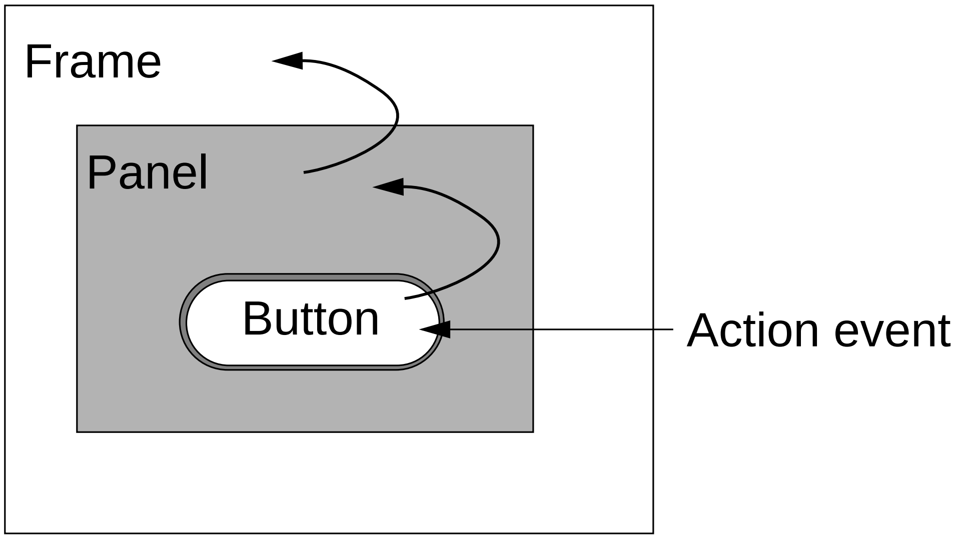
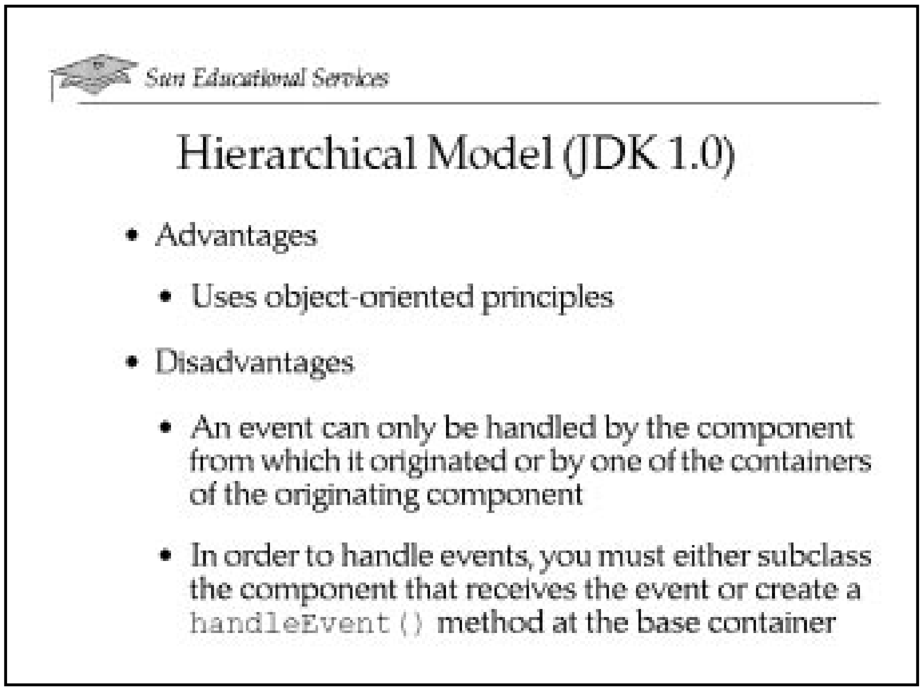


**Figure 9-1** Hirearchical Event Model



Sun Educational Services

## Hierarchical Model (JDK 1.0)

- Advantages
  - Uses object-oriented principles
- Disadvantages
  - An event can only be handled by the component from which it originated or by one of the containers of the originating component
  - In order to handle events, you must either subclass the component that receives the event or create a `handleEvent()` method at the base container

## *JDK 1.0 Event Model Versus JDK 1.2 Event Model*

### *Hierarchal Model (JDK 1.0) (Continued)*

There is an obvious advantage to this model:

- It is simple and uses features of an object-oriented programming environment; after all, Java software components extend from the `java.awt.Component` class, which defines `handleEvent().` Override `handleEvent()` to customize event handling.

There are, however, some disadvantages:

- The event can be handled only by the component from where it originates or by one of the containers that contains it. This restriction violates one of the fundamental principles of object-oriented programming: functionality should reside in the most appropriate class. Often the most appropriate class for handling an event is not a member of the originating component's containment hierarchy.

## DK 1.0 Event Model Versus JDK 1.2 Event Model

### *Hierarchal Model (JDK 1.0) (Continued)*

- A large number of CPU cycles are wasted on unrelated events. Any event unrelated or unimportant to the program would traverse the containment hierarchy before eventually being discarded. There is no simple way to filter events.

✓ ***It is possible to throw a new event in JDK 1.0, but it requires knowledge of the superclass hierarchy and specialized methods like*** `postEvent`***.***

- In order to handle events, you must either subclass the component that receives the event or create a `handleEvent()` method at the base container.

✓ ***This involves quite a bit of software.***

## JDK 1.0 Event Model Versus JDK 1.2 Event Model

### Delegation Model (JDK 1.1 and Beyond)

The delegation event model came into existence with JDK 1.1. With this model, events are sent to the component from which the event originated, but it is up to each component to propogate the event to one or more registered classes called *listeners*. Listeners contain event handlers that can receive and process the event. In this way, the event handler can be in an object separate from the component. Listeners are classes which implement the `EventListener` interface.

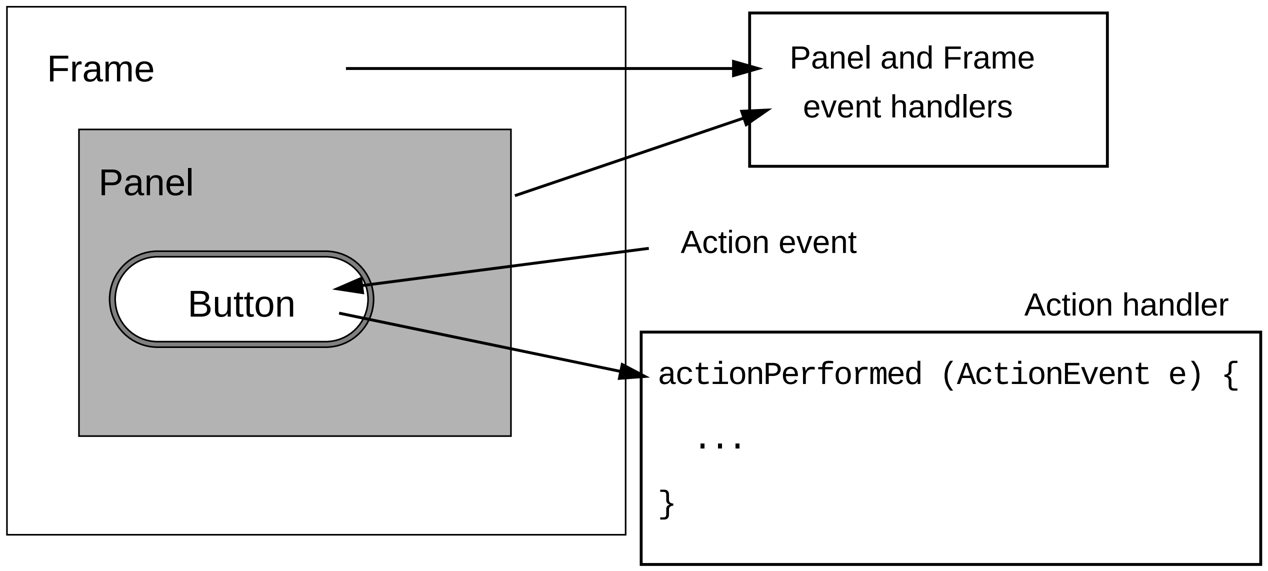

Frame
Panel
Button
Panel and Frame event handlers
Action event
Action handler
actionPerformed (ActionEvent e) {
...
}

**Figure 9-2** Delegation Event Model

Events are objects that are only reported to registered listeners. Every event has a corresponding listener interface which mandates which methods must be defined in a class suited to receiving that type of event. The class that implements the interface defines those methods, and can be registered as a listener.

Events from components which have no registered listeners are not propagated.

## JDK 1.0 Event Model Versus JDK 1.2 Event Model

### Delegation Model (JDK 1.1 and Beyond) (Continued)

For example, here is the code for a simple `Frame` with a single `Button` on it:

```
import java.awt.*;

public class TestButton {
  public static void main(String args[]) {
    Frame f = new Frame("Test");
    Button b = new Button("Press Me!");
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
}
```

The `ButtonHandler` class is the handler class to which the event will be delegated.

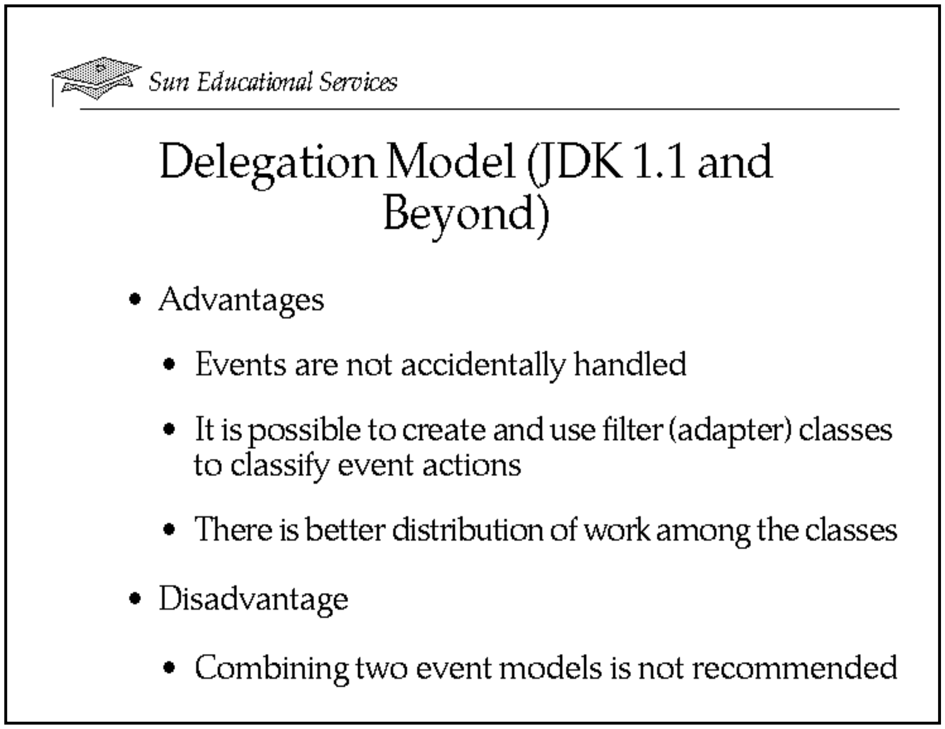
```
import java.awt.event.*;

public class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action occurred");
    System.out.println(
          "Button's label is :"  + e.getActionCommand());
  }
}
```

## JDK 1.0 Event Model Versus JDK 1.2 Event Model

### Delegation Model (JDK 1.1 and Beyond) (Continued)

These two examples have the following characteristics:

- The `Button` class has an `addActionListener(ActionListener)` method.
- The `ActionListener` interface defines a single method, `actionPerformed,` which will receive an `ActionEvent.`
- When a `Button` object is created, it can have an object registered as a listener for `ActionEvents` through the `addActionListener()` method. The registered listener is instantiated from a class that implements the `ActionListener` interface.


Sun Educational Services

Delegation Model (JDK 1.1 and Beyond)

- Advantages
  - Events are not accidentally handled
  - It is possible to create and use filter (adapter) classes to classify event actions
  - There is better distribution of work among the classes
- Disadvantage
  - Combining two event models is not recommended

## *JDK 1.0 Event Model Versus JDK 1.2 Event Model*

### *Delegation Model (JDK 1.1 and Beyond) (Continued)*

- When the `Button` object is clicked on with the mouse, an `ActionEvent` is sent. The `ActionEvent` is received through the `actionPerformed()` method of any `ActionListener` that is registered on the button through its `addActionListener()` method.

- The method `getActionCommand()` of the `ActionEvent` class returns the command name associated with this action. The button click action, for example, returns the label on the `Button` by default.

## JDK 1.0 Event Model Versus JDK 1.2 Event Model

### Delegation Model (JDK 1.1 and Beyond) (Continued)

There are several advantages to this approach:

- Events are not accidentally handled unlike a hierarchical model where it is possible for an event to propagate to a container and get handled at an unexpected level.
- It is possible to use filter (adapter) classes to classify event actions.
- The delegation model is much better for the distribution of work among classes.
- The new event model provides support for JavaBeans™.
- Events do not have to be related to AWT components.

There is also a disadvantage to this approach:

- Although the current JDK supports the JDK 1.0 event model in addition to the delegation model, the JDK 1.0 and JDK 1.1 event models cannot be mixed.

## GUI Behavior

### Event Categories

The general mechanism for receiving events from components has been described in the context of a single type of event. The hierarchy of event classes is as shown in Figure 9-3. Many of the event classes reside in the `java.awt.event` package, but others exist elsewhere in the API.

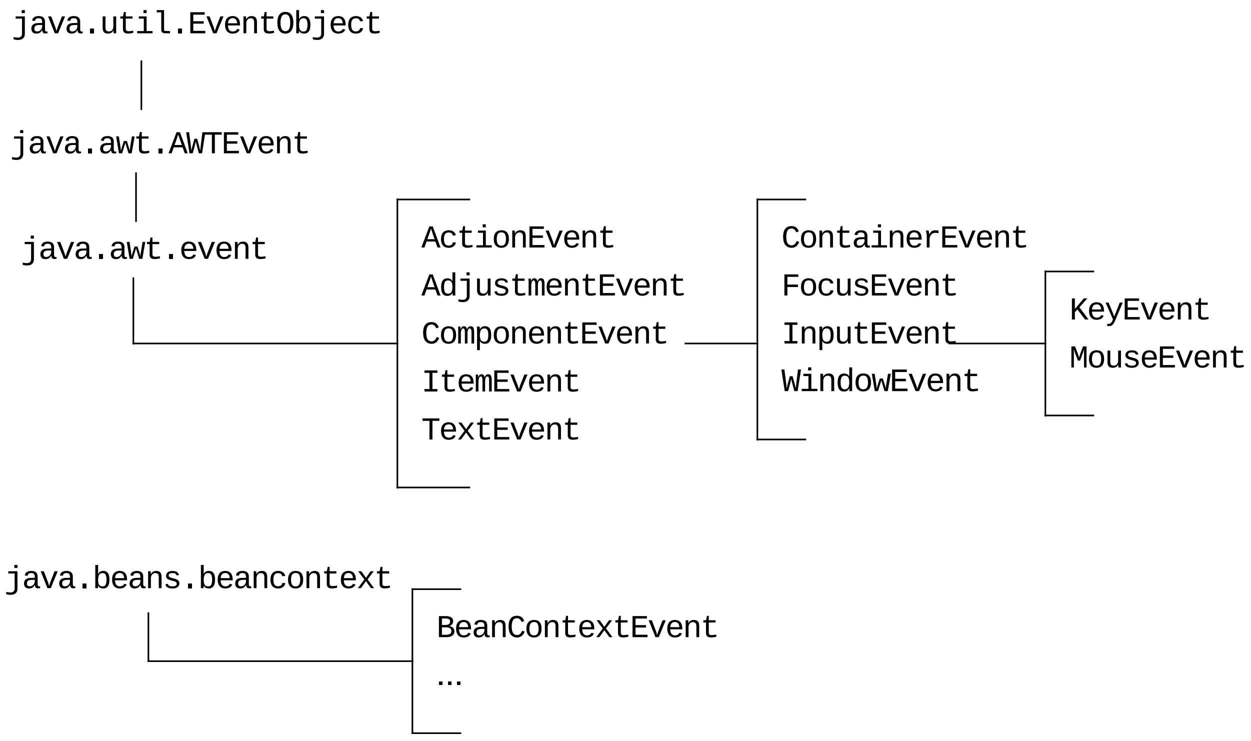


**Figure 9-3** Event Class Hirearchy

For each category of events, there is an interface that has to be implemented by the class of objects that wants to receive the events. That interface demands that one or more methods be defined too. Those methods will be called when particular events arise. Table 9-1 lists the categories, giving the interface name for each category and the methods demanded. The method names are mnemonic, indicating the source or conditions that will cause the method to be called.

## GUI Behavior

**Table 9-1** Method Categories and Interfaces

| Category | Interface Name | Methods |
|---|---|---|
| Action | `ActionListener` | `actionPerformed(ActionEvent)` |
| Item | `ItemListener` | `itemStateChanged(ItemEvent)` |
| Mouse motion | `MouseMotionListener` | `mouseDragged(MouseEvent)` |
| | | `mouseMoved(MouseEvent)` |
| Mouse button | `MouseListener` | `mousePressed(MouseEvent)` |
| | | `mouseReleased(MouseEvent)` |
| | | `mouseEntered(MouseEvent)` |
| | | `mouseExited(MouseEvent)` |
| | | `mouseClicked(MouseEvent)` |
| Key | `KeyListener` | `keyPressed(KeyEvent)` |
| | | `keyReleased(KeyEvent)` |
| | | `keyTyped(KeyEvent)` |
| Focus | `FocusListener` | `focusGained(FocusEvent)` |
| | | `focusLost(FocusEvent)` |
| Adjustment | `AdjustmentListener` | `adjustmentValueChanged (AdjustmentEvent)` |
| Component | `ComponentListener` | `componentMoved(ComponentEvent)` |
| | | `componentHidden(ComponentEvent)` |
| | | `componentResized(ComponentEvent)` |
| | | `componentShown(ComponentEvent)` |

## GUI Behavior

**Table 9-1** Method Categories and Interfaces (Continued)

| Category | Interface Name | Methods |
|---|---|---|
| Window | `WindowListener` | `windowClosing(WindowEvent)` |
| | | `windowOpened(WindowEvent)` |
| | | `windowIconified(WindowEvent)` |
| | | `windowDeiconified(WindowEvent)` |
| | | `windowClosed(WindowEvent)` |
| | | `windowActivated(WindowEvent)` |
| | | `windowDeactivated(WindowEvent)` |
| Container | `ContainerListener` | `componentAdded(ContainerEvent)` |
| | | `componentRemoved(ContainerEvent)` |
| Text | `TextListener` | `textValueChanged(TextEvent)` |

# GUI Behavior

## Complex Example

This section examines a more complex Java code software example. It tracks the movement of the mouse when the mouse button is pressed (*mouse dragging*). It also detects mouse movement even when the buttons are not pressed (*mouse moving*).

The events caused by moving the mouse with or without a button pressed can be picked up by objects of a class that implements the `MouseMotionListener` interface. This interface requires two methods, `mouseDragged()` and `mouseMoved()`. Even if you are only interested in the drag movement, both methods must be provided. The body of the `mouseMoved()` method can be empty, however.

To pick up other mouse events, including mouse clicking, the `MouseListener` interface must be implemented. This interface includes several events including `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, and `mouseClicked`.

When mouse or keyboard events occur, information about the position of the mouse and the key that was pressed is available in the event that it generated. In the first example on event handling, there was a separate class named `ButtonHandler` that handled events. In the following example, events will be handled within the class named `TwoListen`.

## Complex Example (Continued)

This is the code:

```
import java.awt.*;
import java.awt.event.*;

public class TwoListen
      implements MouseMotionListener,
      MouseListener {
  private Frame f;
  private TextField tf;

  public void go() {
    f = new Frame("Two listeners example");
    f.add(new Label ("Click and drag the mouse"),
                      BorderLayout.NORTH);
    tf = new TextField (30);
    f.add (tf, BorderLayout.SOUTH);

    f.addMouseMotionListener(this);
    f.addMouseListener (this);
    f.setSize(300, 200);
    f.setVisible(true);
  }

  // These are MouseMotionListener events
    public void mouseDragged (MouseEvent e) {
    String s =
      "Mouse dragging:  X = " + e.getX() +
                      " Y = " + e.getY();
    tf.setText (s);
  }

  public void mouseEntered (MouseEvent e) {
    String s = "The mouse entered";
    tf.setText (s);
  }

```

### Complex Example (Continued)

```
36   public void mouseExited (MouseEvent e) {
37     String s = "The mouse has left the building";
38     tf.setText (s);
39   }
40
41   // Unused MouseMotionListener method.
42   // All methods of a listener must be present in the
43   // class even if they are not used.
44   public void mouseMoved (MouseEvent e) { }
45
46   // Unused MouseListener methods.
47   public void mousePressed (MouseEvent e) { }
48   public void mouseClicked (MouseEvent e) { }
49   public void mouseReleased (MouseEvent e) { }
50
51   public static void main(String args[]) {
52     TwoListen two = new TwoListen();
53     two.go();
54   }
55 }
```

A number of points in this example are discussed in the following sections.

#### Declaring Multiple Interfaces

The class is declared in lines 5 and 6 using the following:

```
implements MouseMotionListener,
            MouseListener
```

Multiple interfaces can be declared by using comma separation.

## GUI Behavior

### Complex Example (Continued)

#### Listening to Multiple Sources

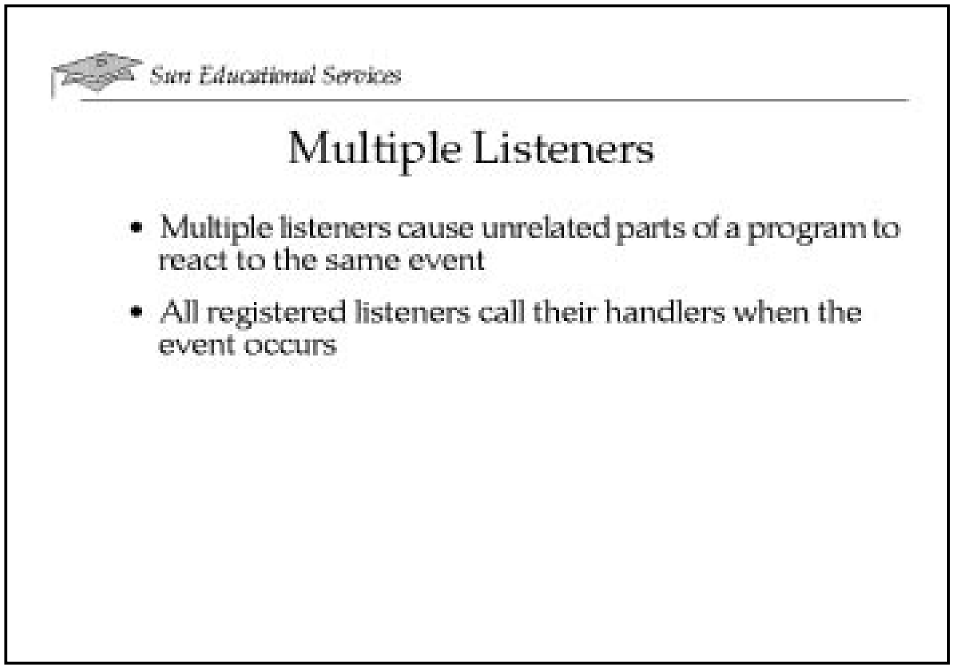
If you issue the following method calls in lines 17 and 18:

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

both types of events cause methods to be called in the `TwoListen` class. An object can "listen" to as many event sources as required; its class need only implement the required interfaces.

#### Obtaining Details About the Event

The event arguments with which handler methods such as `mouseDragged()` are called contain potentially important information about the original event. To determine the details of what information is available for each category of event, check the appropriate class documentation in the `java.awt.event` package.


Sun Educational Services

## Multiple Listeners

- Multiple listeners cause unrelated parts of a program to react to the same event
- All registered listeners call their handlers when the event occurs

## *GUI Behavior*

### *Multiple Listeners*

The AWT event listening framework allows multiple listeners to be attached to the same component. In general, if you want to write a program that performs multiple actions based on a single event, code that behavior into your handler method. However, sometimes a program's design requires multiple unrelated parts of the same program to react to the same event. This might happen if, for instance, a context-sensitive help system is being added to an existing program.

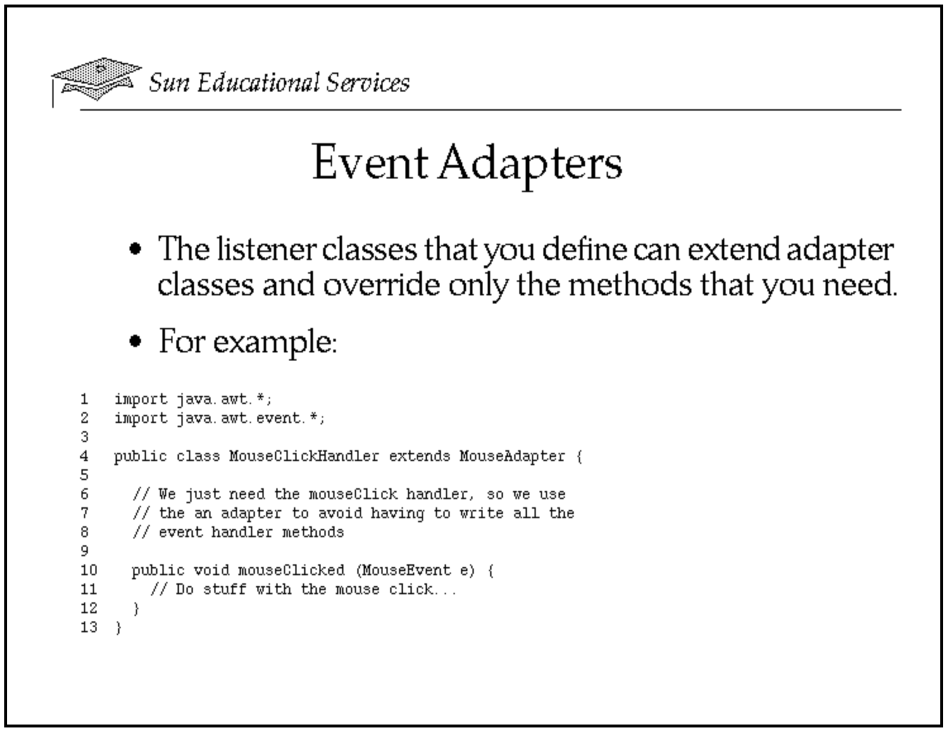
The listener mechanism allows you to call an `addXXXListener()` method as many times as is needed, and you can specify as many different listeners as your design requires. All registered listeners will have their handler methods called when the event occurs.

## *GUI Behavior*

### *Multiple Listeners (Continued)*

**Note –** The order in which the handler methods are called is undefined. Generally, if the order of invocation matters, then the handlers are not unrelated. In this case, register only the first listener, and have that one call the others directly.

✓ ***This is called an event multiplexer.***



Sun Educational Services

## Event Adapters

- The listener classes that you define can extend adapter classes and override only the methods that you need.
- For example:

```
import java.awt.*;
import java.awt.event.*;

public class MouseClickHandler extends MouseAdapter {

  // We just need the mouseClick handler, so we use
  // the an adapter to avoid having to write all the
  // event handler methods

  public void mouseClicked (MouseEvent e) {
    // Do stuff with the mouse click...
  }
}
```

## *Event Adapters*

It is a lot of work to implement all of the methods in each of the listener interfaces, particularly the `MouseListener` interface and `WindowListener` interface.

The `MouseListener` interface, for example, declares the following methods:

- `mouseClicked (MouseEvent)`
- `mouseEntered (MouseEvent)`
- `mouseExited (MouseEvent)`
- `mousePressed (MouseEvent)`
- `mouseReleased (MouseEvent)`

## Event Adapters

As a convenience, the Java programming language provides adapter classes which implement each interface containing more than one method. The methods in these adapter classes are empty.

You can extend an adapter class and override only those methods that you need. For example:

```
import java.awt.*;
import java.awt.event.*;

public class MouseClickHandler extends MouseAdapter {

  // We just need the mouseClick handler, so we use
  // an adapter to avoid having to write all the
  // event handler methods

  public void mouseClicked (MouseEvent e) {
    // Do stuff with the mouse click...
  }
}
```

**Note –** This is a class, not an interface. This means you can only extend one other class. Because listeners are interfaces, you can extend multiple ones.

**Note –** Be careful when overriding methods. Remember that a declaration like `public void mouseClicked(MouseEvent e)` is legal, and really a new method, not an overriding one. If your method does nothing with an event, it may be difficult to track because it has the same method name as the parent class method.

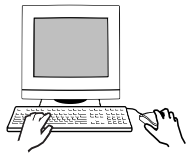
## Anonymous Classes

It is possible to include an entire class definition within the scope of an expression. This approach defines what is called an *anonymous* inner class and creates the instance all at once. Anonymous inner classes are generally used in conjuction with AWT event handling. For example:

```
import java.awt.*;
import java.awt.event.*;

public class AnonTest {
  private Frame f;
  private TextField tf;

  public void go() {
    f = new Frame("Anonymous classes example");
    f.add(new Label("Click and drag the mouse"),
            BorderLayout.NORTH);
    tf = new TextField (30);
    f.add (tf, BorderLayout.SOUTH);

    f.addMouseMotionListener( new MouseMotionAdapter() {
      public void mouseDragged (MouseEvent e) {
        String s = "Mouse dragging:  X = "+ e.getX() +
                    " Y = " + e.getY();
        tf.setText (s);
      }
    }); // <- note the closing parenthesis

    f.addMouseListener (new MouseClickHandler());
    f.setSize(300, 200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    AnonTest obj = new AnonTest();
    obj.go();
  }
}
```

✓ ***Emphasize to your students that the use of anonymous classes is not recommended for code reuse and readability!***

## Exercise: Working With Events



**Exercise objective –** You will write, compile, and run the revised Calculator GUI and Account GUI codes to include event handlers.

### Preparation

In order to successfully complete this lab, you must have a clear understanding of how the event model works.

### Tasks

#### *Level 1 Lab: Create a Calculator GUI, Part II*

Complete the following step:

1. Using the GUI code you created from the Module 8 exercise, write an event code that connects the calculator's user interface to the appropriate event handlers for the functions on a calculator. Also, add a handler for closing the window.

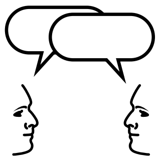
#### *Level 3 Lab: Create an Account GUI, Part II*

Perform the following step:

1. Create an AccountEvent class, objects of which are fired when changes to accounts occur. Then fire the events to the BankManager class. Begin with the `Teller.java` GUI code you created for the Module 8 lab.

## Exercise: Working With Events

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Define events and event handling
- ❑ Describe the differences between JDK1.0 and later event models
- ❑ Write code to handle events that occur in a GUI
- ❑ Describe the concept of adapter classes, including how and when to use them
- ❑ Determine the user action that originated the event from the event object details
- ❑ Create the appropriate interface and event handler methods for a variety of event types.
- ❑ Understand inner classes and anonymous classes

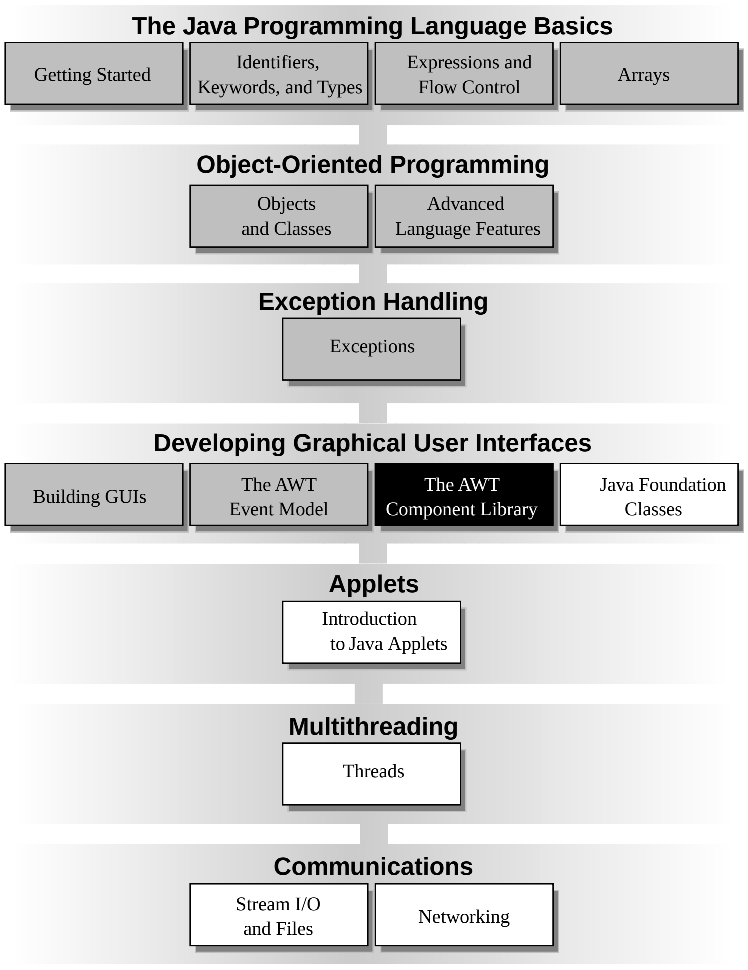
## Think Beyond

You now know how to set up a Java software GUI for both graphic output and interactive user input. However, only a few of the components from which GUIs can be built have been described. What other components would be useful in a GUI?

# The AWT Component Library 10

## Course Map

The JDK offers many components from which GUIs can be built. This module examines some of these AWT components, and covers non-component AWT classes such as Color and Font, as well as GUI printing.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

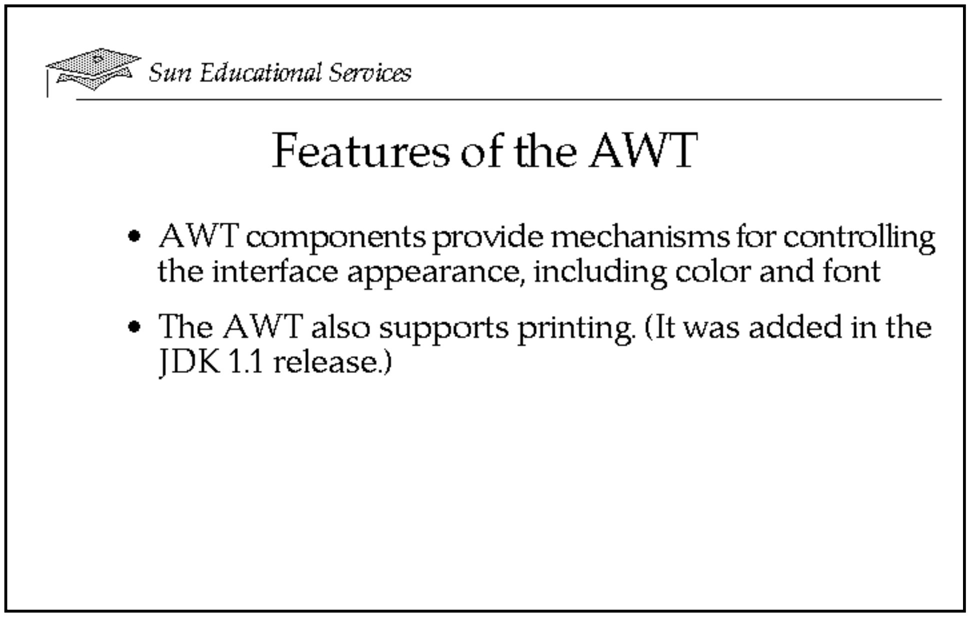
**Discussion –** The following question is relevant to the material presented in this module:

- You now know how to set up a Java software GUI for both graphic output and interactive user input. However, only a few of the components from which GUIs can be built have been described. What other components would be useful in a GUI?

## Objectives

Upon completion of this module, you should be able to

- Recognize the key AWT components
- Given a user interface description, use the AWT components to build a user interface.
- Given an AWT program, change the colors and fonts of the AWT component.
- Use the Java printing mechanism to print a user interface.



Sun Educational Services

## Features of the AWT

- AWT components provide mechanisms for controlling the interface appearance, including color and font
- The AWT also supports printing. (It was added in the JDK 1.1 release.)

## *Features of the AWT*

The AWT provides a wide variety of standard features. This module introduces many of the components that are available to you, and outlines any particular anomalies that you might need to know about.

Initially, the components of the AWT are described. These are used to construct user interfaces. You need to be aware of the full set of GUI components, so that you can choose the appropriate ones when building your own interfaces.

The AWT components provide mechanisms for controlling the interface appearance, including color and the font used for text display.

Additionally, the AWT supports printing. This is a facility that was added with the transition to JDK 1.1.

## Button

You have already become familiar with the Button component. It provides a basic “push to activate” user interface component. It can be constructed with a text label that informs the user of its use.

```
  f = new Frame("Sample Button");
  b = new Button("Sample");
  b.addActionListener(this);
  f.add(b);
```

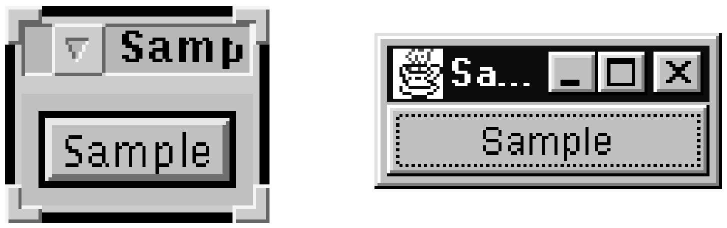


**Figure 10-1** Button

The `actionPerformed()` method of any class implementing the `ActionListener` interface, which is registered as a listener, is called when the button is “pressed” by a mouse click.

```
public void actionPerformed( ActionEvent ae) {
  System.out.println("Button press received.");
  System.out.println("Button's action command is: " +
      ae.getActionCommand());
}
```

The `getActionCommand()` method of the `ActionEvent` that is issued when the button is pressed returns the label string by default. The action command or label is changed by using the button’s `setActionCommand()` method.

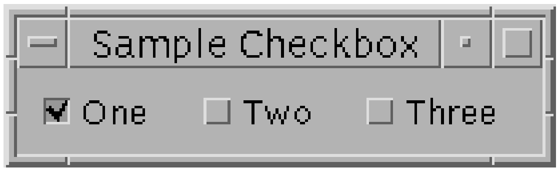
```
  b = new Button("Sample");
  b.setActionCommand("Action Command Was Here!");
  b.addActionListener(this);
  f.add(b);
```

**Note –** The complete source for `SampleButton` and `ActionCommandButton` can be found in the `examples` directory.

## Checkbox

The Checkbox component provides a simple "on/off" input device with a text label beside it.

```
f = new Frame("Sample Checkbox");
one = new Checkbox("One", true);
two = new Checkbox("Two", false);
three = new Checkbox("Three", false);

one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);

f.setLayout(new FlowLayout());
f.add(one);
f.add(two);
f.add(three);
```



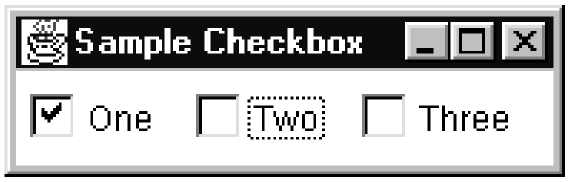


**Figure 10-2** Checkbox

Selection or deselection of a checkbox is sent to the ItemListener interface. The ItemEvent that is passed contains the method getStatechange(), which returns ItemEvent.DESELECTED or ItemEvent.SELECTED, as appropriate. The method getItem() returns the affected checkbox as a String object that represents its label.

```
public void itemStateChanged(ItemEvent ev) {
  String state = "deselected";
  if (ev.getStateChange() == ItemEvent.SELECTED) {
    state = "selected";
  }
  System.out.println (ev.getItem() + " " + state);
}
```

## Checkbox Group – Radio Buttons

CheckboxGroup provides the means to group multiple Checkbox items into a mutual exclusion set, so that only one Checkbox in the set has the value true at any time. The Checkbox with the value true is the currently selected Checkbox. You can create each Checkbox of a group using a constructor that takes an additional argument, a CheckboxGroup. It is this CheckboxGroup object that connects the Checkbox items together into a set. If you do this, the appearance of the Checkbox items is changed and all that are related to the same CheckboxGroup will exhibit "radio button" behavior.

```
f = new Frame("CheckBoxGroup");
cbg = new CheckboxGroup();
one = new Checkbox("One", cbg, false);
two = new Checkbox("Two", cbg, false);
three = new Checkbox("Three", cbg, true);

f.setLayout(new FlowLayout());

one.addItemListener(this);
two.addItemListener(this);
three.addItemListener(this);

f.add(one);
f.add(two);
f.add(three);
```

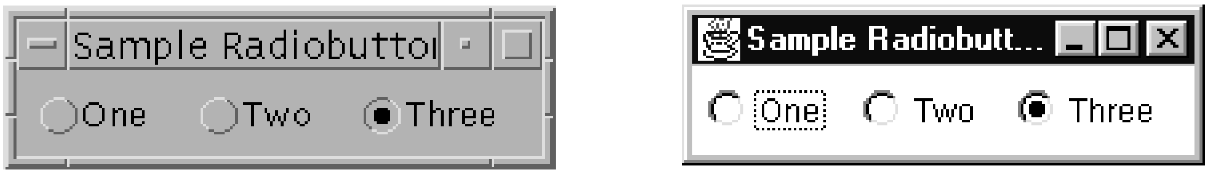


**Figure 10-3** CheckboxGroup

## Choice

The Choice component provides a simple “select one from this list” type of input. For example:

```
f = new Frame("Sample Choice");
choice = new Choice();
choice.addItem("First");
choice.addItem("Second");
choice.addItem("Third");
choice.addItemListener(this);
f.add(choice, BorderLayout.CENTER);
```

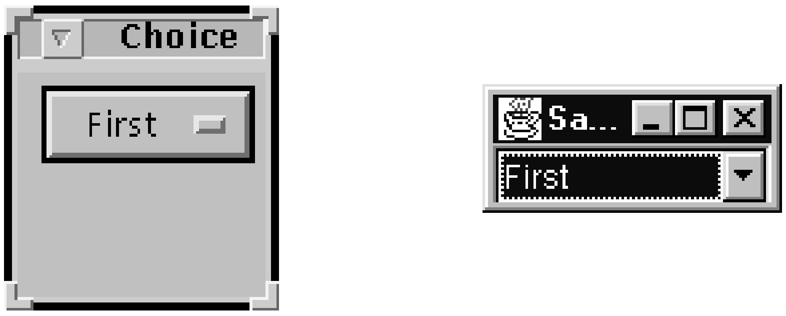


**Figure 10-4** Choice

When the Choice is clicked on, it displays a list of items that have been added to it. Notice that the items added are String objects.

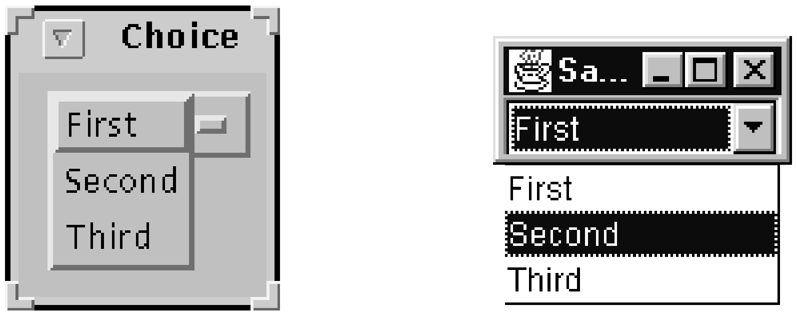


**Figure 10-5** Choice With Items

The ItemListener interface is used to observe changes in the Choice. The details are the same as those for the Checkbox.

## Canvas

A Canvas provides a blank (background colored) space. It has a preferred size of zero by zero, unless you explicitly specify a size using setSize(). To specify the size, place it in a layout manager that specifes the size.

The space can be used to draw, write text, or receive keyboard or mouse input. A later module will show you how to draw effectively in the AWT.

Generally a Canvas is used either to provide general drawing space or to provide a working area for a custom component.

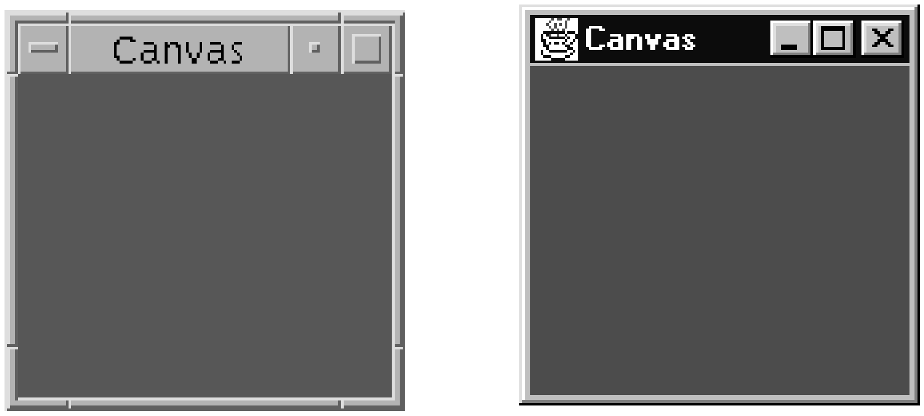


**Figure 10-6** Canvas

The Canvas can "listen" to all the events that are applicable to a general component. In particular, you might want to add KeyListener, MouseMotionListener, or MouseListener objects to it to allow it to respond in some way to user input.

---

**Note –** To receive key events in a Canvas, it is necessary to call the requestFocus() method of the Canvas. If this is not done, it will generally not be possible to "direct" the keystrokes to Canvas. Instead, the keystrokes will go to another component, or perhaps be lost entirely.

---

✓ ***A*** Canvas ***is usually extended for use as a drawing component.***

## Canvas

Here is an example of a `Canvas`. This program changes the color of the `Canvas` each time a key is pressed.

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class MyCanvas extends Canvas
      implements KeyListener{
  private int index;
  Color colors[] = { Color.red, Color.green, Color.blue };

  public void paint(Graphics g) {
    g.setColor(colors[ index ]);
    g.fillRect(0, 0, getSize().width, getSize().height);
  }

  public void keyTyped(KeyEvent ev) {
    index++;
    if ( index == colors.length ) {
      index = 0;
    }
    repaint();
  }

  // Unused KeyListener methods
  public void keyPressed(KeyEvent ev) { }
  public void keyReleased(KeyEvent ev) { }

  public static void main(String args[]) {
    Frame f = new Frame("Canvas");
    MyCanvas mc = new MyCanvas();
    mc.setSize(150, 150);
    f.add(mc, BorderLayout.CENTER);
    mc.requestFocus();
    mc.addKeyListener(mc);
    f.pack();
    f.setVisible(true);
  }
}
```

## Label

A `Label` object displays a single line of static text. The program can change the text, but the user cannot. No special borders or other decorations are used to delineate a `Label`.

```
Frame f = new Frame("Label");
Label lb = new Label("Hello");
f.add(lb);
```

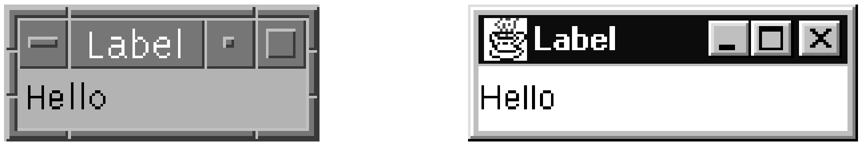


**Figure 10-7** `Label`

`Label` is not usually expected to handle events, but can do so in the same manner as a `Canvas`. That is, keystrokes can only be picked up reliably by calling `requestFocus()`.

## TextField

The TextField is a single line text input device. For example:

```
Frame f = new Frame("TextField");
TextField tf = new TextField("Single line"″, 30);
tf.addActionListener(this);
f.add(tf);
```

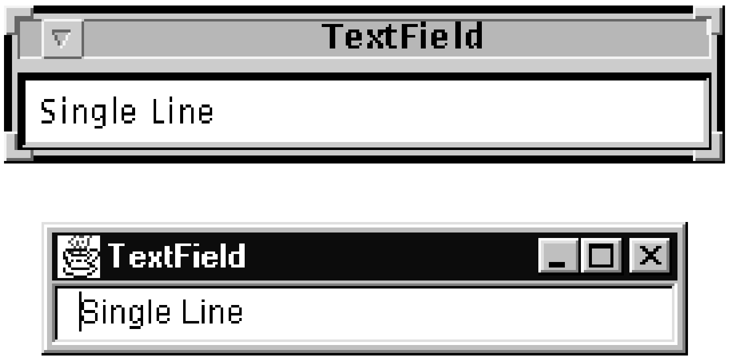


**Figure 10-8** TextField

Because only one line is possible, an ActionListener can be informed, via actionPerformed(), when the Enter or Return key is pressed. Other component listeners can be added if desired.

---

**Note –** The second argument in the TextField constructor is for how many characters are visible. There is no limit on the number of characters allowed in the TextField. Scrolling occurs when the text overflows.

---

## TextField

Masking certain keys from input is sometimes used in TextField applications. The following code creates a TextField that ignores the typing of digits:

```
import java.awt.*;
import java.awt.event.*;

public class SampleTextField {
  private Frame f;
  private TextField tf;

  public void go() {
    f = new Frame("TextField");
    tf = new TextField("Single Line", 30);
    tf.addKeyListener( new NameHandler() );
    f.add(tf, BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

  class NameHandler extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
      char c = e.getKeyChar();
      if ( Character.isDigit(c)) {
        e.consume();
      }
    }
  }

  public static void main (String args[]) {
    SampleTextField txtf = new SampleTextField();
    txtf.go();
  }
}
```

## TextArea

The `TextArea` is a multiple-line, multiple-column text input device. You can set it to read-only, using the method `setEditable(boolean)`. It displays horizontal and vertical scrollbars.

The following example sets up a 4 row X 30 character text area containing "Hello!" initially.

```
f = new Frame("TextArea");
ta = new TextArea("Hello!", 4, 30);
f.add(ta, BorderLayout.CENTER);
```

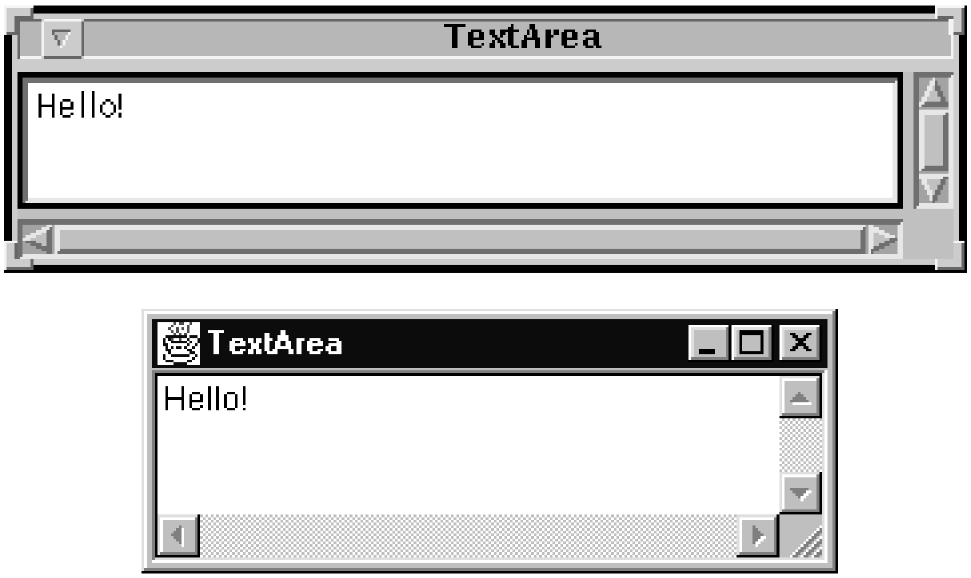
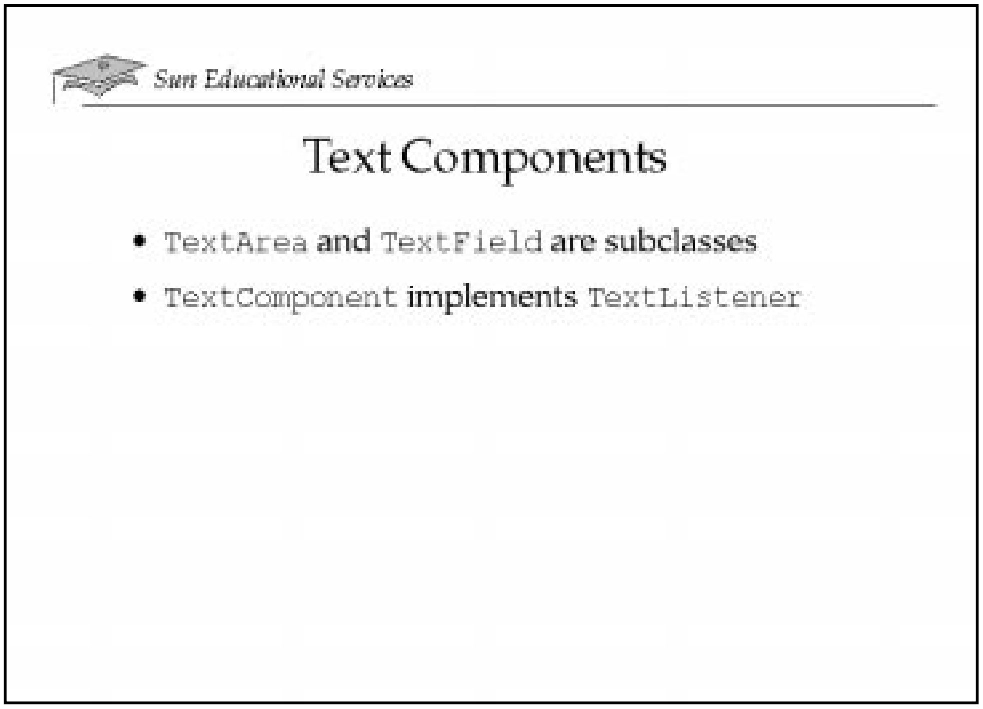


**Figure 10-9** `TextArea`

The listener you specify with `addTextListener()` will receive notification of key strokes in the same way that `TextField` does.

You can add general component listeners to the text area, but because the text is multi-line, pressing the Enter key puts another character into the buffer. If you need to recognize "completion of input," you can put an Apply or Commit button next to the text area to allow the user to indicate this.


Sun Educational Services

Text Components

- TextArea and TextField are subclasses
- TextComponent implements TextListener

## *Text Components*

Both TextArea and TextField are documented in two parts. If you look up a class called TextComponent you will find several methods that TextArea and TextField have in common; for example, setEditable(). This is because TextArea and TextField are both subclasses of TextComponent.

You have seen that the constructors for both TextArea and TextField classes allow you to specify the number of columns for the display. Remember that the size of a displayed component is the responsibility of a layout manager, so these preferences might be ignored. Furthermore, the number of columns is interpreted in terms of the average width of characters in the font that is being used. The number of characters that are actually displayed might vary radically if a proportionally spaced font is used.

Since TextComponent implements TextListener, classes like TextField, TextArea, and any other subclasses have built-in support for keystroke events.

## List

A `List` presents text options which are displayed in a region that allows several items to be viewed at one time. The `List` is scrollable and supports both single- and multiple-selection modes. For example:

```
List lst = new List(4, true);
lst.add("Hello");
lst.add("there");
lst.add("how");
```

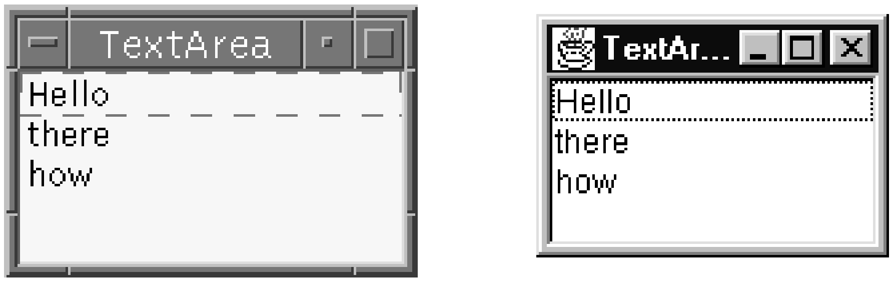


**Figure 10-10** `List`

The numeric argument to the constructor defines the preferred height of the list in terms of the number of visible rows. As always, this value can be overridden by a layout manager. A true boolean argument indicates that the list should allow the user to make multiple selections.

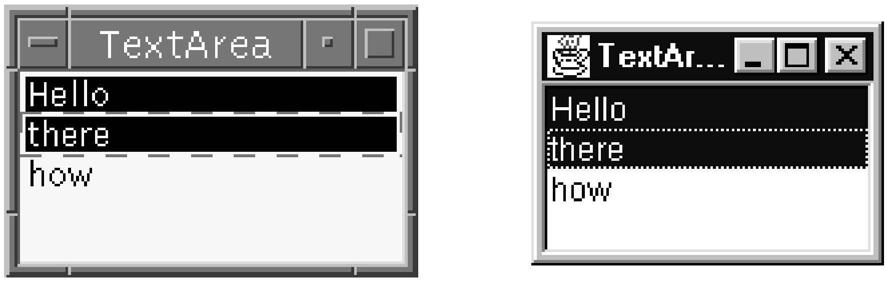


**Figure 10-11** `List` With Items Selected

When an item is selected or deselected, AWT sends an instance of `ItemEvent` to the list. When the user double-clicks on an item in a scrolling list, an `ActionEvent` is generated by the list in both single- and multiple-selection modes. Items are selected from the list according to platform conventions. For a UNIX® / Motif environment, this means that a single click will highlight an entry in the list, but a double click is needed to trigger the action of the list.

## Dialog

A `Dialog` component is associated with a `Frame`. It is a free-standing window with some decorations. It differs from a `Frame` in that fewer decorations are provided and you can request a “modal” dialog which causes it to store all forms of input until it is closed.

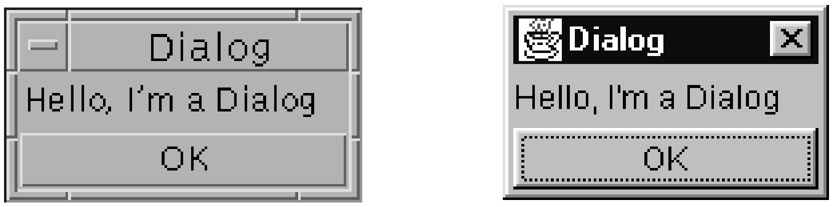


**Figure 10-12** `Dialog`

`Dialog` is either modeless or modal. A modeless `Dialog` means a user can interact with both the `Frame` and the `Dialog` at the same time. A modal `Dialog` blocks input to the remainder of the application, including the `Frame`, until the `Dialog` box is dismissed.

Since `Dialog` subclasses `Window`, its default layout manager is `BorderLayout.`

```
d = new Dialog(f, "Dialog", true);
d.setLayout(new GridLayout(2,1));
dl = new Label("Hello, I'm a Dialog");
db1 = new Button("OK");
d.add(dl);
d.add(db1);
d.pack();
```

The first argument in the `Dialog` constructor designates the owner of the `Dialog` that is being constructed. In the previous example, `f` is the `Frame` that owns the `Dialog`.

## Dialog

A `Dialog` is usually not made visible to the user when it is first created. They are generally displayed in response to some user interface action, such as the pressing of a button.

```
public void actionPerformed(ActionEvent ev) {
  d.setVisible(true);
}
```

---

**Note –** Treat a `Dialog` as a reusable device. That is, do not destroy the individual object when it is dismissed from the display; keep it so it can be used later. The garbage collector can make it too easy to waste memory. Remember, creating and initializing objects takes time and should not be done without some thought.

---

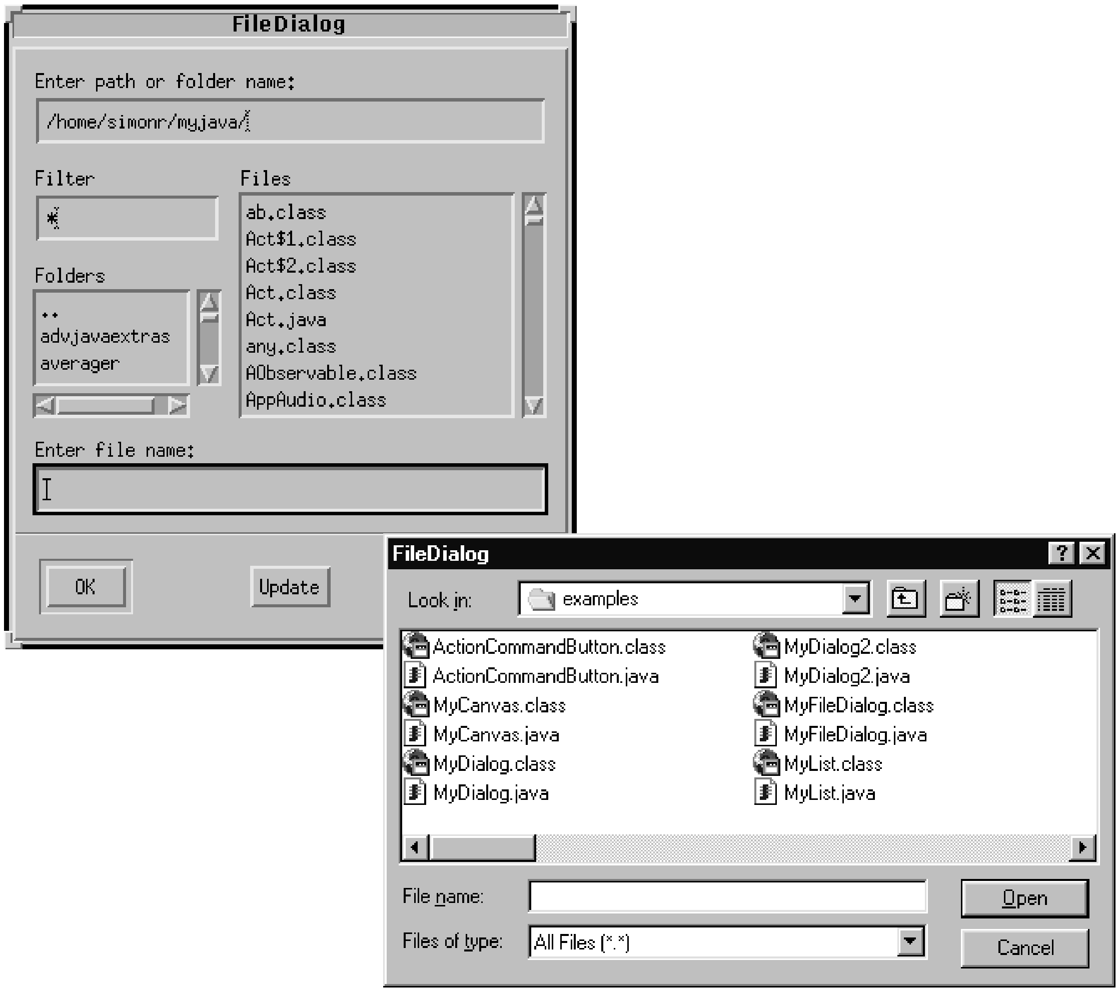
To hide a `Dialog`, you must call `setVisible(false)`. This can be done by adding a `WindowListener` to it and awaiting a call to the `windowClosing()` method in that listener. This parallels the handling of closing a `Frame`.

✓ ***Handling window events is more complicated. The*** `listener` ***class can extend*** `WindowAdapter` ***or implement*** `WindowListener`***. The*** `PaintGUI` ***class in this module's solution directory contains an example of Window handling code.***

## FileDialog

`FileDialog` is an implementation of a file selection device. It has its own free standing window, with window elements, and allows the user to browse the file system and select a particular file for further operations. For example:

```
FileDialog d = new FileDialog(parentFrame, "FileDialog");
d.setVisible(true);  // block here until OK selected
String fname = d.getDirectory() + d.getFile();
```



Enter path or folder name: /home/simonr/myjava/
Filter: *
Folders: .. advjavaextras averager
Files: ab.class Act$1.class Act$2.class Act.class Act.java any.class AObservable.class AppAudio.class
Enter file name:

Look in: examples
ActionCommandButton.class
ActionCommandButton.java
MyCanvas.class
MyCanvas.java
MyDialog.class
MyDialog.java
MyDialog2.class
MyDialog2.java
MyFileDialog.class
MyFileDialog.java
MyList.class
MyList.java
File name:
Files of type: All Files (*.*)

**Figure 10-13** `FileDialog`

In general, it is not necessary to handle events from the `FileDialog`. The `setVisible(true)` call blocks events until the user selects OK, at which point the name of the selected file is requested. This information is returned as a `String`.

## ScrollPane

ScrollPane provides a general container that cannot be used as a free standing component. It should always be associated with a container (for example, a Frame). It provides a viewport onto a larger region and scrollbars to manipulate that viewport. For example:

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));
.
.
.
sp.add(p);
f.add(sp, BorderLayout.CENTER);
f.setSize(100, 100);
f.setVisible(true);
```

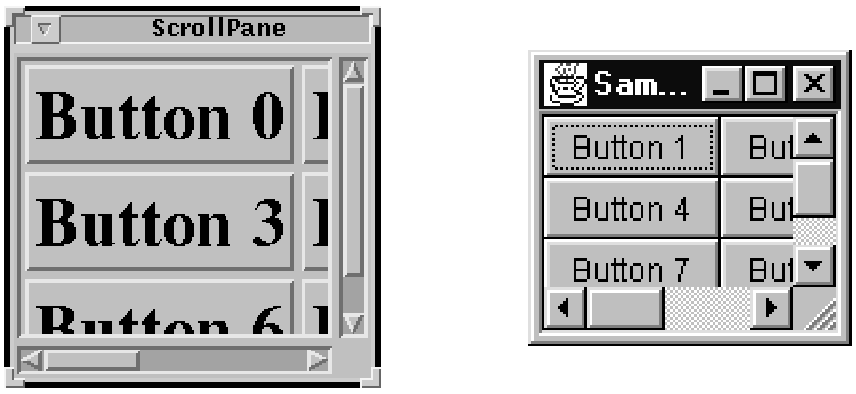
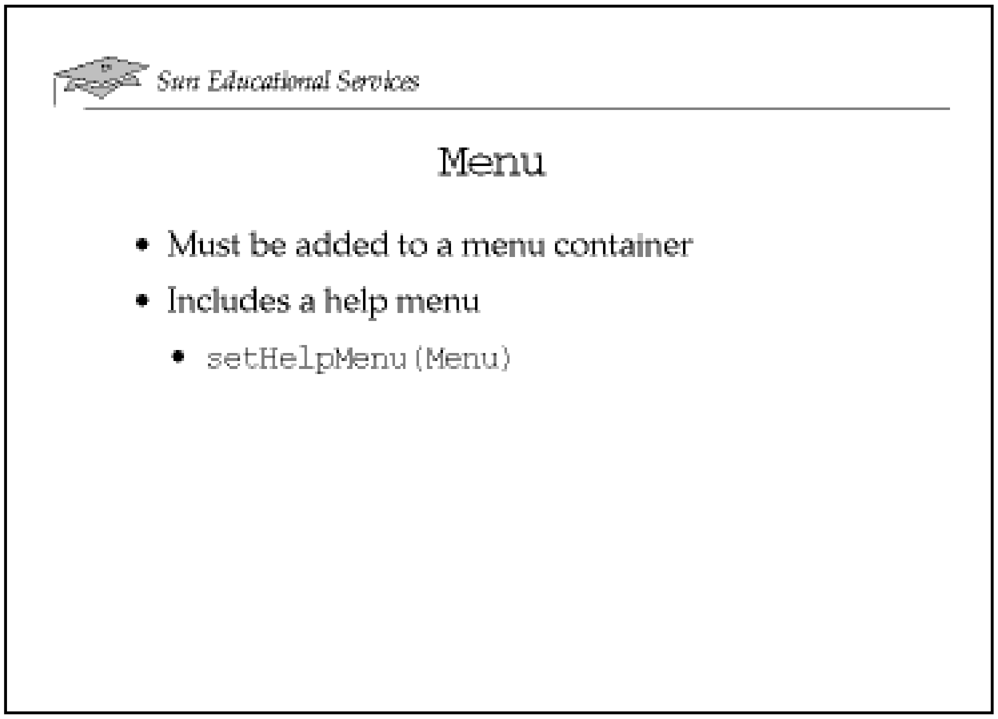


**Figure 10-14** ScrollPane

The ScrollPane creates and manages the scroll bars and holds a single component. You cannot control the layout manager it uses. Instead, you can add a Panel to the scroll pane, configure the layout manager of that panel, and place your components in that panel.

Generally, you do not handle events on a ScrollPane; events are handled through the components that they contain.


Sun Educational Services

## Menu

- Must be added to a menu container
- Includes a help menu
  - `setHelpMenu(Menu)`

## *Menus*

A `Menu` is different from other components in one crucial way: you cannot add a `Menu` to ordinary containers and have them laid out by the layout manager. You can add menus only to a *menu container*. You can start a menu "tree" by putting a menu bar in a `Frame`, using the `setMenuBar()` method. From that point, you can add menus to the menu bar and menus or menu items to the menus.

Pop-up menus are an exception since they appear as floating windows and therefore do not require layout.

## Menus

### The Help Menu

One particular feature of the menu bar is that you can designate one menu to be the Help menu. This is done with the method `setHelpMenu(Menu)`. The menu to be treated as the Help menu must be added to the menu bar; it will then be treated in the same way as the Help menu for the local platform. For X/Motif type systems, this involves flushing the menu entry to the right-hand end of the menu bar.

## MenuBar

A MenuBar component is a horizontal menu. It can be added only to a Frame object, and it forms the root of all menu trees. A Frame can display one MenuBar at a time. However, you can change the MenuBar based on the state of the program so that different menus can appear at various points. For example:

```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

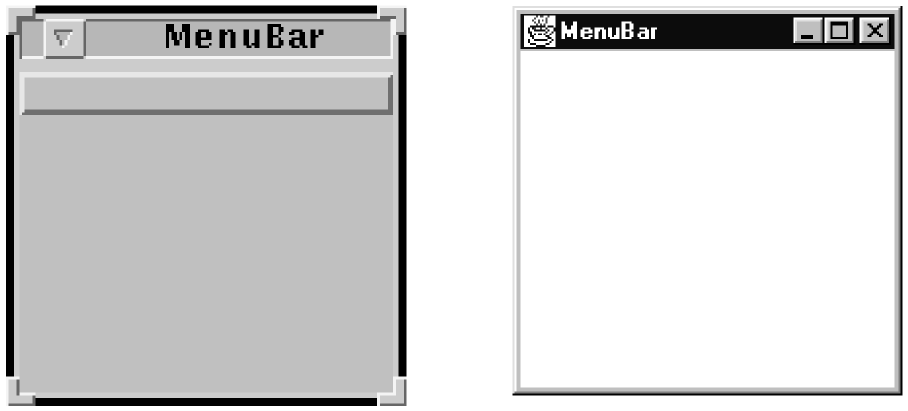


**Figure 10-15** MenuBar

The MenuBar does not support listeners. As part of the normal menu behavior, anticipated events that occur in the region of a menu bar will be processed automatically.

## Menu

The Menu component provides a basic pull-down menu. It can be added either to a MenuBar or to another Menu. For example:

```
f = new Frame("Menu");
mb = new MenuBar();
m1 = new Menu("File");
m2 = new Menu("Edit");
m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
```

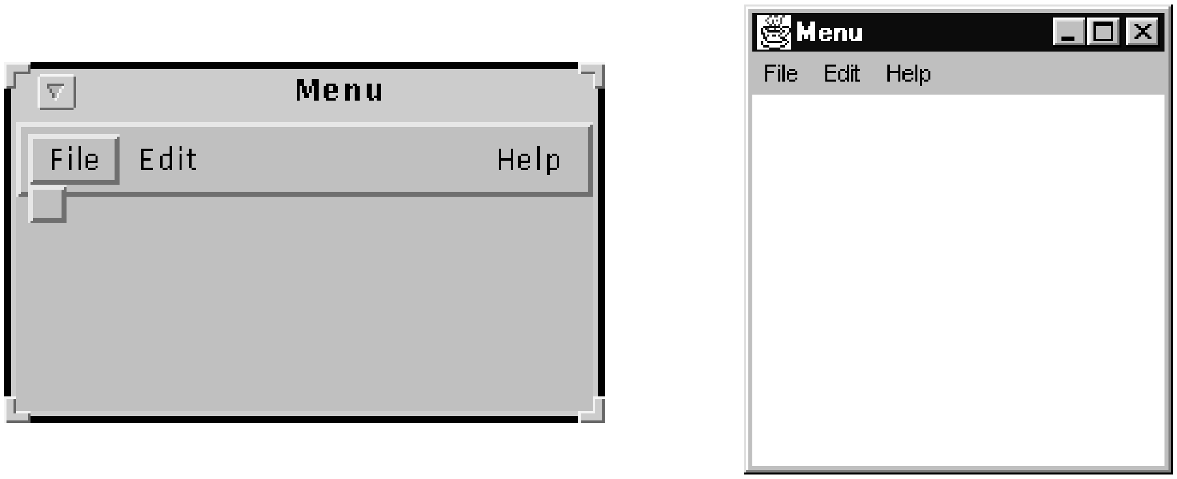


**Figure 10-16** Menu

**Note –** The menus shown here are empty, which accounts for the appearance of the File menu.

You *can* add an ActionListener to a Menu object, but this would be unusual. Normally, menus are used to display and control menu items, which are discussed next.

## MenuItem

MenuItem components are the text *leaf* nodes of a menu tree. They are generally added to a menu to complete it. For example:

```
mi1 = new MenuItem("New");
mi2 = new MenuItem("Save");
mi3 = new MenuItem("Load");
mi4 = new MenuItem("Quit");
mi1.addActionListener(this);
mi2.addActionListener(this);
mi3.addActionListener(this);
mi4.addActionListener(this);
m1.add(mi1);
m1.add(mi2);
m1.add(mi3);
m1.addSeparator();
m1.add(mi4);
```

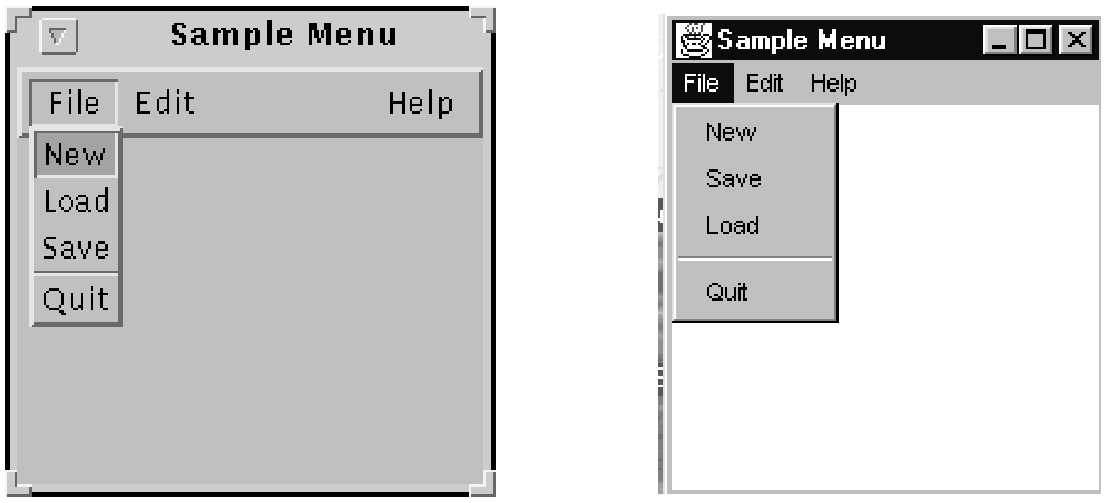


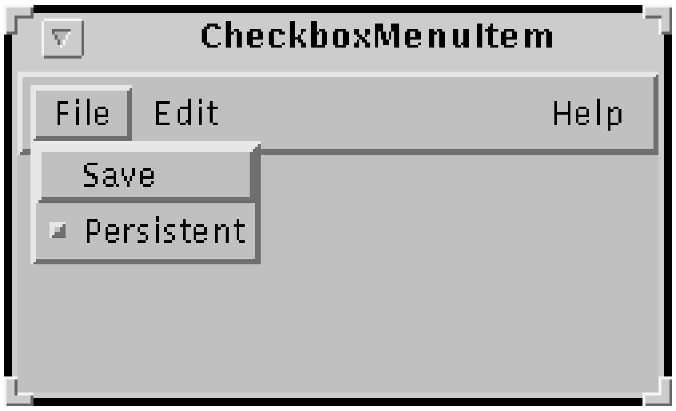
**Figure 10-17** MenuItem

Usually an ActionListener is added to a MenuItem object to provide behavior for the menus.

## CheckboxMenuItem

`CheckboxMenuItem` is a checkable menu item, so you can have selections (*on* or *off* choices) listed in menus. For example:

```
mb = new MenuBar();
m1 = new Menu("File");
m2 = new Menu("Edit");
m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
f.setMenuBar(mb);
.....
mi2 = new MenuItem("Save");
mi2.addActionListener(this);
m1.add(mi2);
......
mi5 = new CheckboxMenuItem("Persistent");
mi5.addItemListener(this);
m1.add(mi5);
```



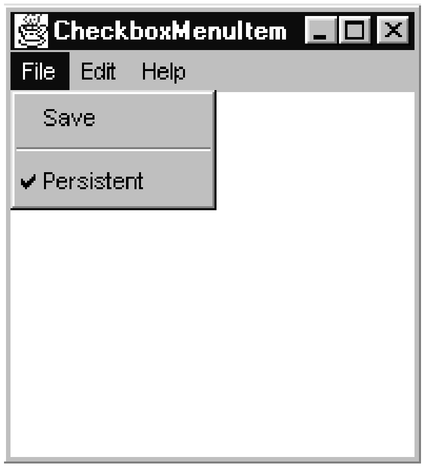


**Figure 10-18** `CheckboxMenuItem`

The `CheckboxMenuItem` should be monitored via the `ItemListener` interface. The `itemStateChanged()` method is called when the checkbox state is modified.

## PopupMenu

The PopupMenu provides a stand-alone menu that can be displayed on any component. You can add items or menus to a pop-up menu. For example:

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem ld = new MenuItem("Load");
b.addActionListener(this);
f.add(b,BorderLayout.CENTER);
p.add(s);
p.add(ld);
f.add(p);
```

For the PopupMenu to be displayed, you must call the show() method. The show() method requires a component reference to act as the origin for the x and y coordinates. Usually you would use the trigger component for this. In this case, the trigger is the Button b.

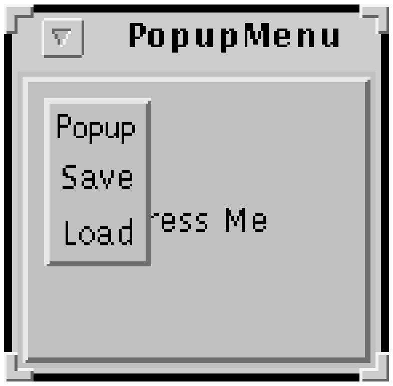


**Figure 10-19** PopupMenu

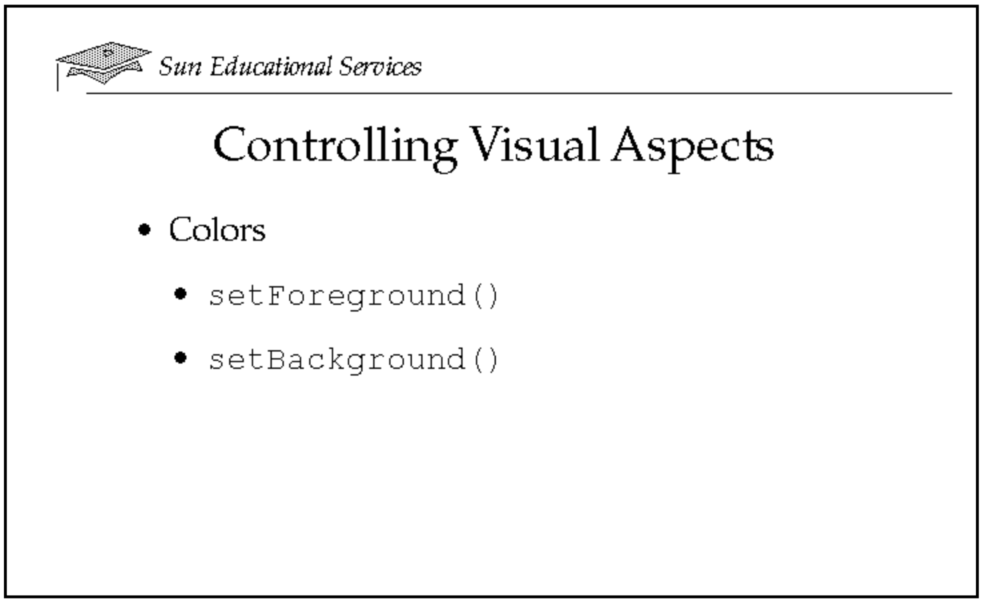
✓ ***The Windows version of the popup menu is not available because it does not work properly in JDK1.2. You get a popup menu containing two buttons that are labeled Press Me. This may be an opportunity to discuss compatibility. A bug report has been filed.***

## Popup Menu

```
public void actionPerformed(ActionEvent ev) {

  // display popup at (10,10) relative to b
  p.show(b, 10, 10);
}
```

**Note –** The `PopupMenu` must be added to a "parent" component. This is not the same as adding components to containers. In this example, the pop-up menu has been added to the enclosing `Frame`.

✓ ***At this point, students might have questions about the restrictions on the containment hierarchy.***



Sun Educational Services

Controlling Visual Aspects

- Colors
  - `setForeground()`
  - `setBackground()`

## Controlling Visual Aspects

You can control the colors used for the foreground and the background of AWT components.

✓ ***Some platforms do not allow the colors to be changed on specific components. For example, Windows95/NT does not allow a button's color to be altered.***

### Colors

Two methods are used to set the colors of a component:

- `setForeground()`
- `setBackground()`

Both of these methods take an argument that is an instance of the `java.awt.Color` class. You can use constant colors referred to as `Color.red`, `Color.blue`, and so on. The full range of predefined colors is listed in the documentation page for the `Color` class.

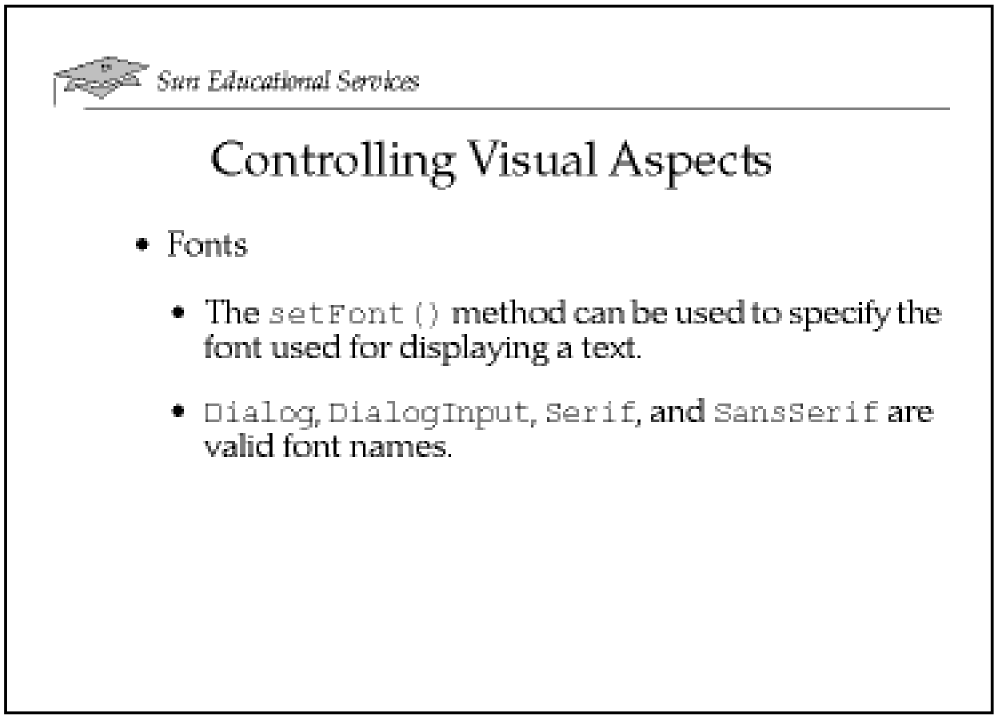
## Controlling Visual Aspects

### Colors (Continued)

In addition, you can create a specific color like this:

```
int r = 255;
int g = 255;
int b = 0;
Color c = new Color(r, g, b);
```

Such a constructor creates a color based on the specified intensities (in a range of 0 to 255 for each) of red, green, and blue.



Sun Educational Services

## Controlling Visual Aspects

- Fonts
  - The `setFont()` method can be used to specify the font used for displaying a text.
  - `Dialog`, `DialogInput`, `Serif`, and `SansSerif` are valid font names.

## Controlling Visual Aspects

### Fonts

The font used for displaying text in a component can be specified by using the method `setFont()`. The argument to this method should be an instance of the `java.awt.Font` class.

No constants are defined for fonts, but you can create a font by specifying the name of the font, the style, and the point size.

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

Valid font names include

- `Dialog`
- `DialogInput`
- `Serif`
- `SansSerif`

# Controlling Visual Aspects

## Fonts

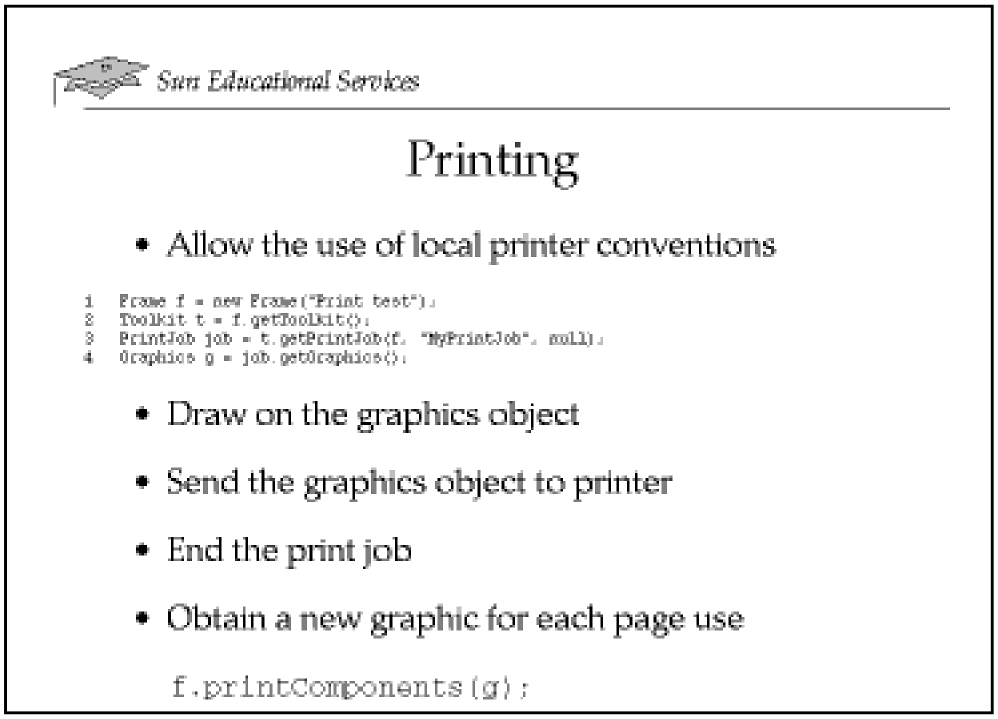
A full list can be determined by looking at the contents of the `java.awt.GraphicsEnvironment` package. The `getToolkit()` method is used to obtain the toolkit after it has been displayed. Alternatively, you can use the default toolkit, which you can acquire by calling `Toolkit.getDefaultToolkit()`.

Font-style constants, which are actually `int` values, are

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

Point sizes should be specified using an `int` value.

✓ ***Java 2D has expanded the available fonts substantially. Java 2D is discussed very briefly in the Java Foundation Classes module.***



Sun Educational Services

## Printing

- Allow the use of local printer conventions

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
Graphics g = job.getGraphics();
```

- Draw on the graphics object
- Send the graphics object to printer
- End the print job
- Obtain a new graphic for each page use

```
f.printComponents(g);
```

## *Printing*

As of JDK1.1, printing is handled in a fashion that closely parallels screen display. A special kind of `java.awt.Graphics` object is obtained so that any draw instructions sent to that graphic are destined for the printer.

The printing system allows the use of local printer control conventions. Users will see a printer selection dialog box when a print operation is started. They can then choose options such as paper size, print quality, and which printer to use. For example:

```
Frame f = new Frame("Print test");
f.setVisible(true);
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
Graphics g = job.getGraphics();
```

These lines create a `Graphics` object that is “connected” to the printer chosen by the user.

## Printing

You can use any of the `Graphics` class's drawing methods to write to the printer. Alternatively, as shown here, you can ask a component to draw itself onto the graphic.

```
f.printComponents(g);
```

The `print()` method asks a component to draw itself in this way, but it only relates to the component that it has been called for. In the case of a container, you can use the `printComponents()` method to draw the container and all of its components on the printer.

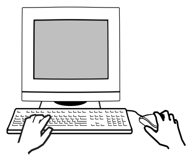
After the page of output is created, use the `dispose()` method to submit that page to the printer.

```
g.dispose();
```

When you have completed the job, call the `end()` method on the print job object. This indicates that the print job is complete and allows the printer spooling system to run the job and release the printer for other jobs.

```
job.end();
```

## Exercise: Creating a Paint Program Layout



**Exercise objective –** In this lab, you will create a more sophisticated GUI application which uses many components.

### Preparation

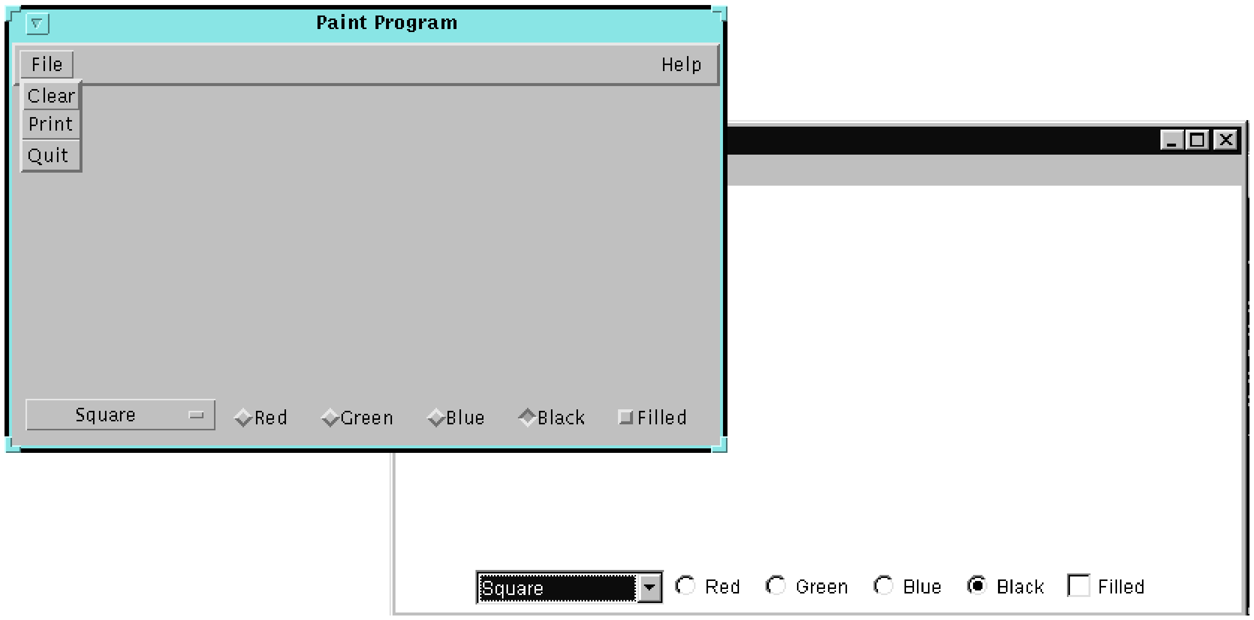
In order to successfully complete this lab, you must understand the purpose of the AWT, its event handlers, and its graphical features.

### Tasks

#### Level 1 Lab: Create the Paint Program Layout

Complete the following step:

1. Using the `java.awt` package, create a Java application, `PaintGUI.java`, that adds the components shown in the following illustration..



## Exercise: Creating a Paint Program Layout

### Tasks

#### Level 2 Lab: Create the Paint Program Layout (Continued)

Perform the following step:

1. Use a dialog for the Help menu.

#### Level 3 Lab: Use the Paint Program

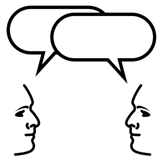
Complete the steps:

1. Create a simple paint program using the layout shown on the previous page as a guide. Use event handling by including a `PaintHandler` class in the GUI.

2. Use `java.awt.PrintJob` to print the graphic created when the Print menu option is selected.

## *Exercise: Creating a Paint Program*

### *Exercise Summary*



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Identify key AWT components
- ❑ Use AWT components to build user interfaces for real programs
- ❑ Control the colors and fonts used by an AWT component
- ❑ Use the Java printing mechanism

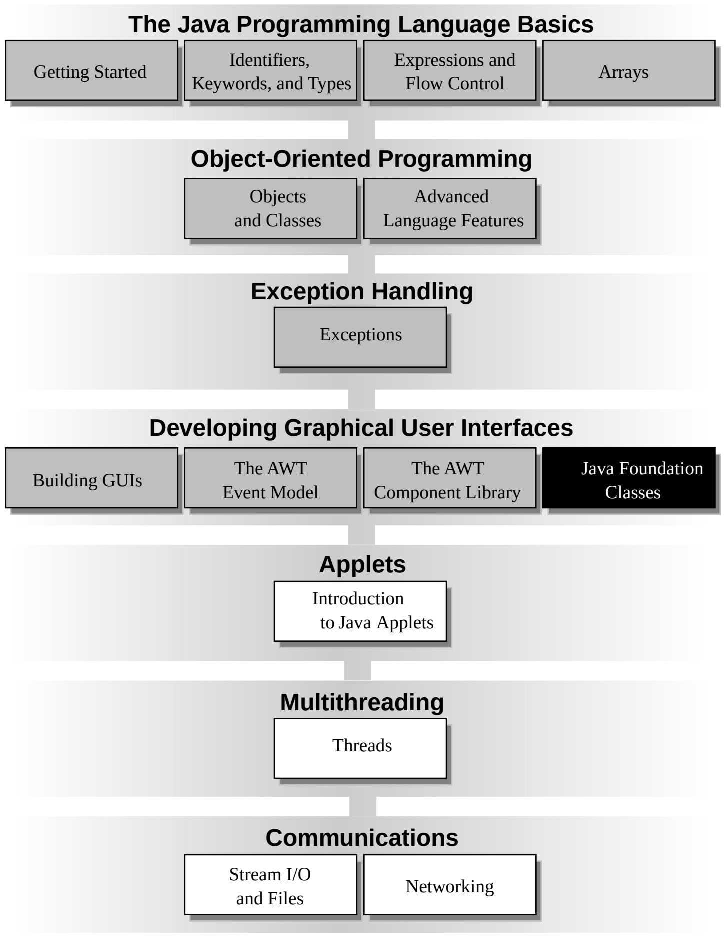
## Think Beyond

What would make the AWT work better?

# Java Foundation Classes 11

## Course Map

JDK 1.2 offers the Java Foundation Classes (JFC), part of which is Swing. Swing is a set of components (written in 100% Pure Java™ for platform independence), layered on top of the AWT. This module introduces JFC, and the implementation of Swing GUIs.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***

**Discussion –** The following questions are relevant to the material presented in this module:

- While the AWT by itself is very useful, it is a part of a new set of classes, called Java Foundation Classes (JFC), that, as a whole, take GUIs to a new level. What exactly is JFC and, in particular, what is Swing? What can Swing do that AWT cannot?

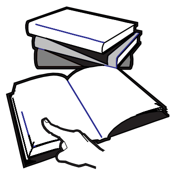
✓ ***The AWT for any particular platform is written partially in the Java programming language and partially in the native code of that platform. The native code portions limit the flexibility of GUI support. JFC classes, written entirely in the Java programming language, interoperate "at the Java level" before calling the AWT for display, to provide a full GUI flexibility not possible with AWT alone.***

## Objectives

Upon completion of this module, you should be able to

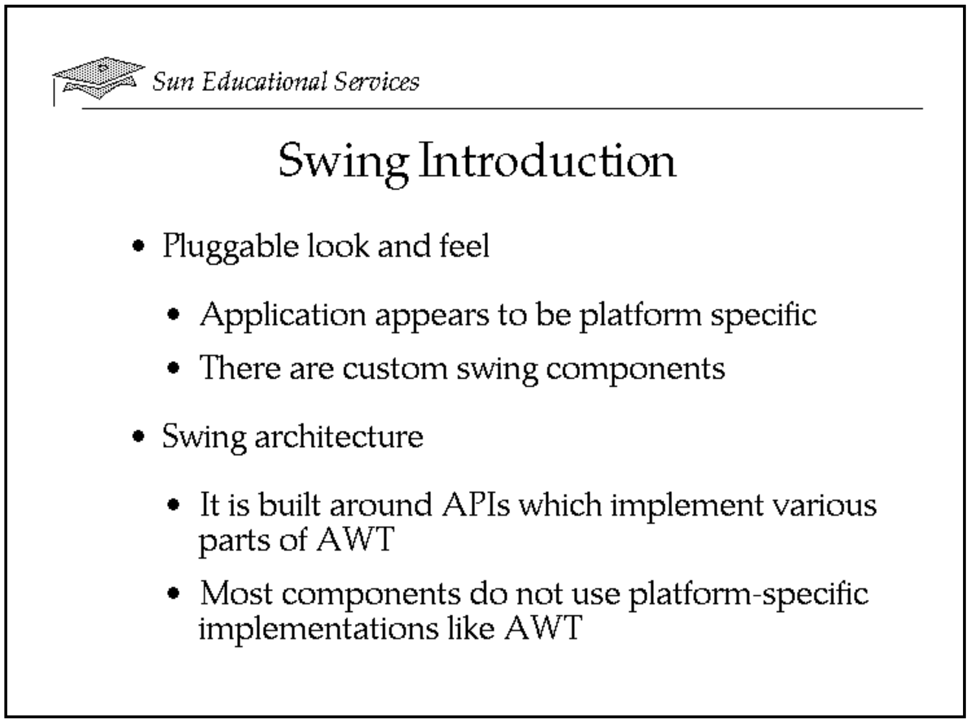
- Identify the key features of Java Foundation Classes
- Describe the key features of `com.sun.java.swing` package
- Identify Swing components
- Define *containers* and *components*, and explain how they work together to build a Swing GUI
- Write, compile and run a basic Swing application
- Use top-level containers such as `JFrame` and `JApplet` effectively

### References



**Additional resources –** following reference can provide additional details on the topics discussed in this module:

- *The Java Tutorial*, an on-line tutorial from Sun Microsystems, available from `http://java.sun.com/docs/books/tutorial`



Sun Educational Services

### Swing Introduction

- Pluggable look and feel
  - Application appears to be platform specific
  - There are custom swing components
- Swing architecture
  - It is built around APIs which implement various parts of AWT
  - Most components do not use platform-specific implementations like AWT

## *Swing Introduction*

The *Java Foundation Classes* (JFC) are a comprehensive set of GUI components and services which dramatically simplify the development and deployment of robust Java applications.

JFC, an integral part of JDK 1.2 is primarily composed of five APIs: AWT, Java 2D, Accessibility, Drag and Drop, and Swing. It provides a full set of application development packages to assist the developer in designing complex interactive applications.

The *AWT* components, as discussed in the earlier modules, provide a variety of GUI tools for a wide class of Java applications.

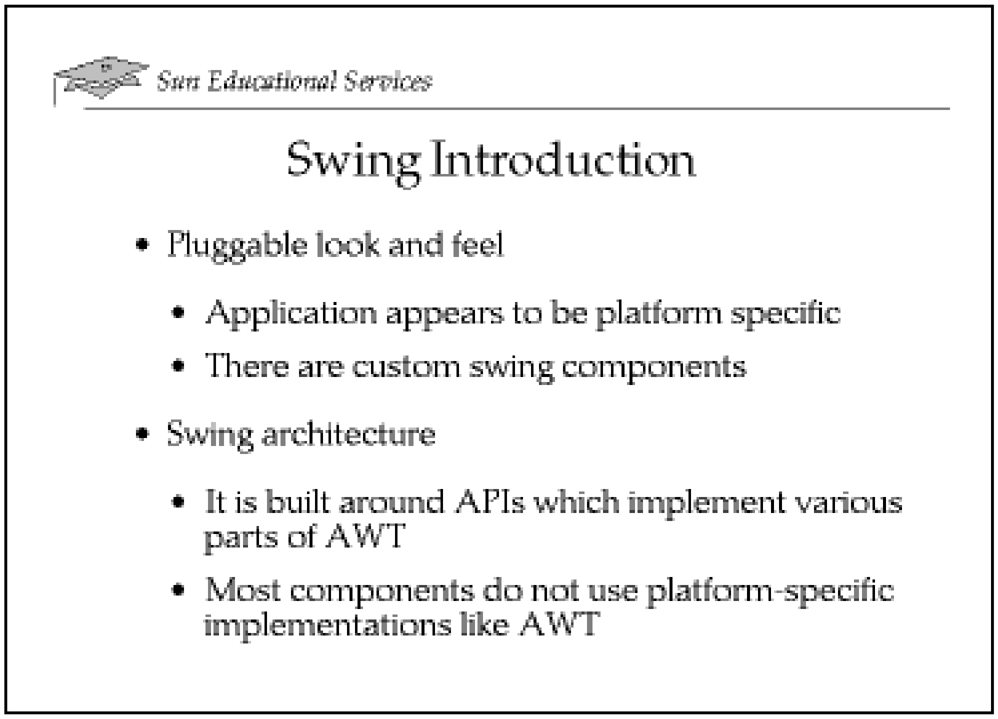
*Java 2D* is a graphics API designed to provide Java applications with an advanced set of classes for two-dimensional (2D) graphics and imaging. The Java 2D API extends the capabilities of the `java.awt` and the `java.awt.image` packages and provides a rich set of paint styles, mechanisms for defining complex shapes, methods, and classes for fine-tuning the rendering process. An extended font set is included in this API.

## Introduction

The *Accessibility* API provides an advanced set of tools to assist in developing applications that use non-traditional means for input and output. It provides an interface for assistive technologies such as screen readers, screen magnifiers, audible text readers (speech processing), and so on.

*Drag and Drop* technology provides interoperability between Java and native applications to exchange data across Java applications and those applications that do not support Java technology.

This JFC module focuses primarily on *Swing*. Swing is designed for forms-based application development. It provides a rich set of components and a framework to specify how GUIs are presented visually independently of the platform.


Sun Educational Services

### Swing Introduction

- Pluggable look and feel
  - Application appears to be platform specific
  - There are custom swing components
- Swing architecture
  - It is built around APIs which implement various parts of AWT
  - Most components do not use platform-specific implementations like AWT

## Swing Introduction

Swing provides a full set of GUI components written entirely in the Java programming language for portability.

### Pluggable Look and Feel

Pluggable look and feel enables developers to build applications which will execute on any platform as if it were developed for that specific platform. A program executed in the Windows environment behaves as if it were developed for this environment; and the same program executed on the UNIX platform behaves as if it were developed for the UNIX environment.

Developers can create their own custom Swing components, with any kind of look and feel they want to design. This increases the consistency of applications and applets deployed across platforms. An entire application's GUI will be able to switch from one look and feel to a different one at runtime.

## Swing – Introduction

### Swing Architecture

Swing provides a more comprehensive set of components than the AWT, introducing new features and rich capabilities. The Swing APIs are built around a number of APIs which implement various parts of the AWT. This ensures that all of the earlier AWT components can still be used. Most Swing components do not use any of the platform-specific implementations that the AWT does, thus giving Swing its customization and pluggable look and feel features.

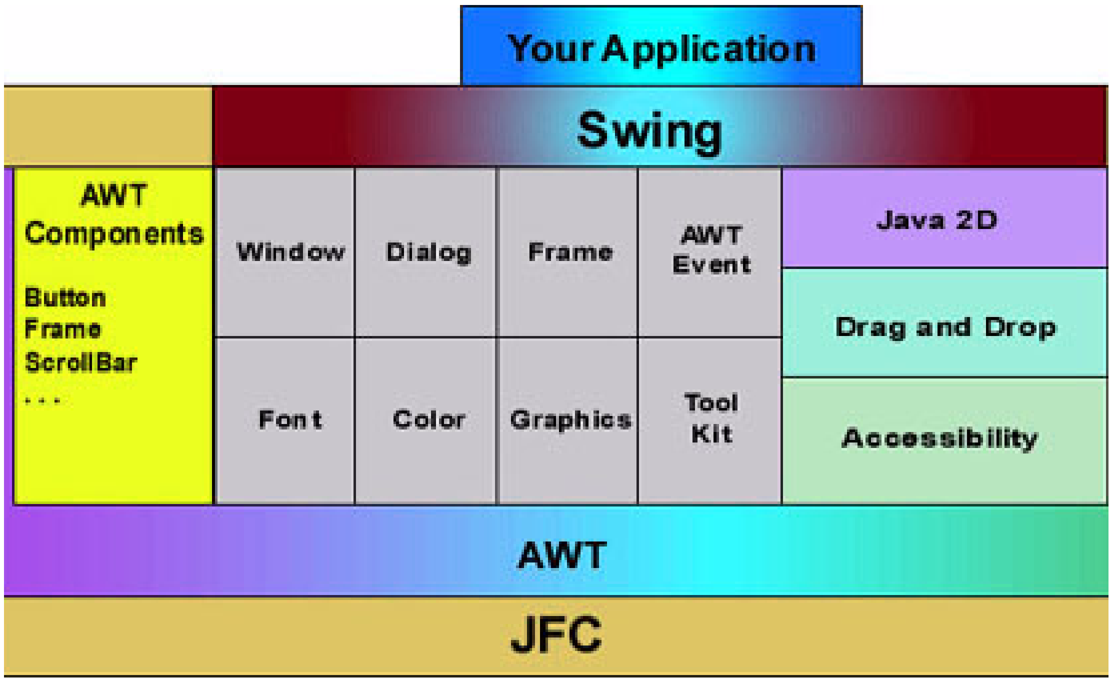


**Figure 11-1** Java Foundation Classes

The above diagram illustrates the interrelationship between various parts of JFC. Java 2D, Drag and Drop, and Accessibility APIs are part of AWT and JFC, but they are not part of Swing. This is because these components use some native code, whereas Swing does not.

Swing is built around a new component called the `JComponent`, which extends from the AWT's `Container` class.

# Swing – Introduction

## The Swing Hierarchy

The following diagram illustrates the Swing component hierarchy:

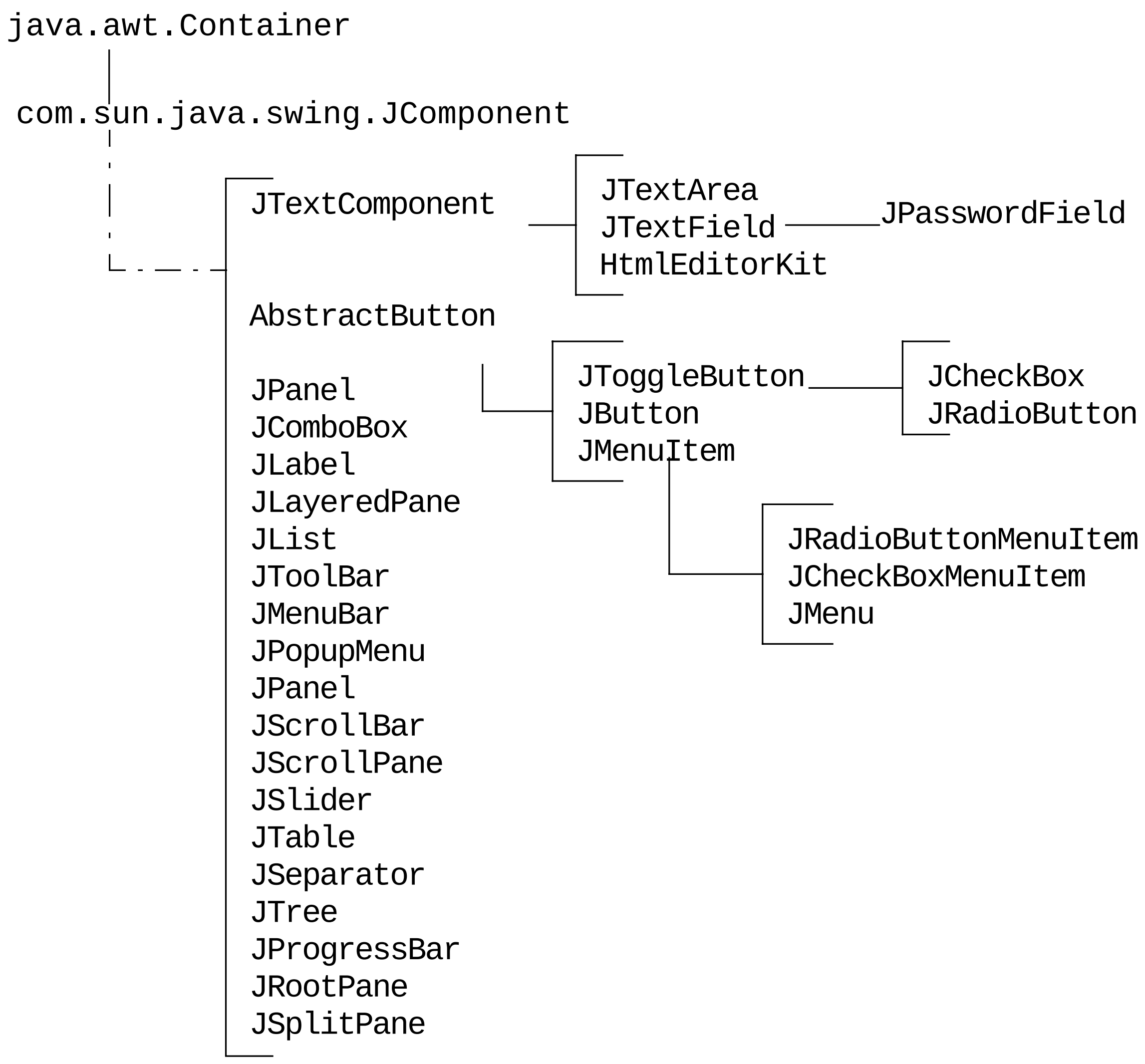


**Figure 11-2** Swing Component Hirearchy

Swing GUIs use two kinds of classes, GUI classes and non-GUI support classes. The GUI classes are visual and descendents of `JComponent`, and are thus called "J" classes. The non-GUI classes provide services and perform vital functions for GUI classes; hence they do not produce any visual output.

---

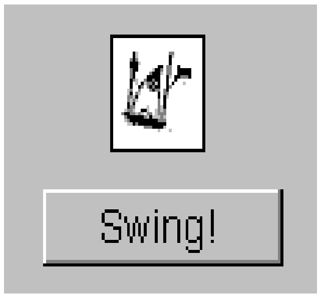
**Note –** Swing's event handling classes are examples of non-GUI classes.

---

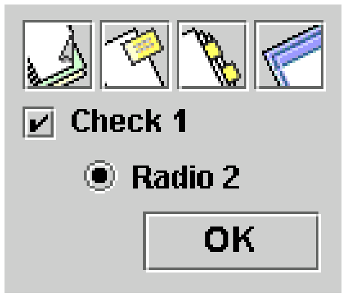
# Swing – Introduction

## Swing Components

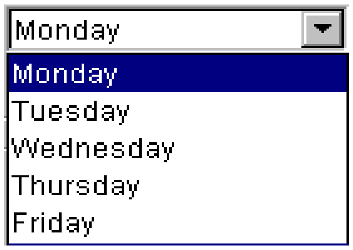
The Swing components primarily provide components for text handling, buttons, labels, lists, panes, combo boxes, scroll bars, scroll panes, menus, tables, and trees. Some of the components appear as follows:



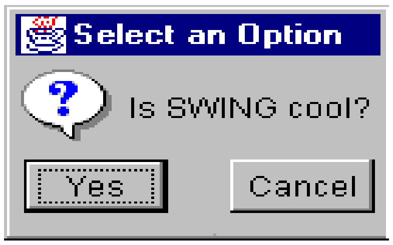
`JApplet`



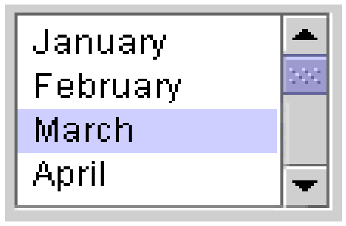
`JButton, JToggleButton`



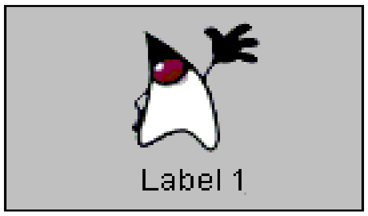
`JComboBox`



`JOptionPane`



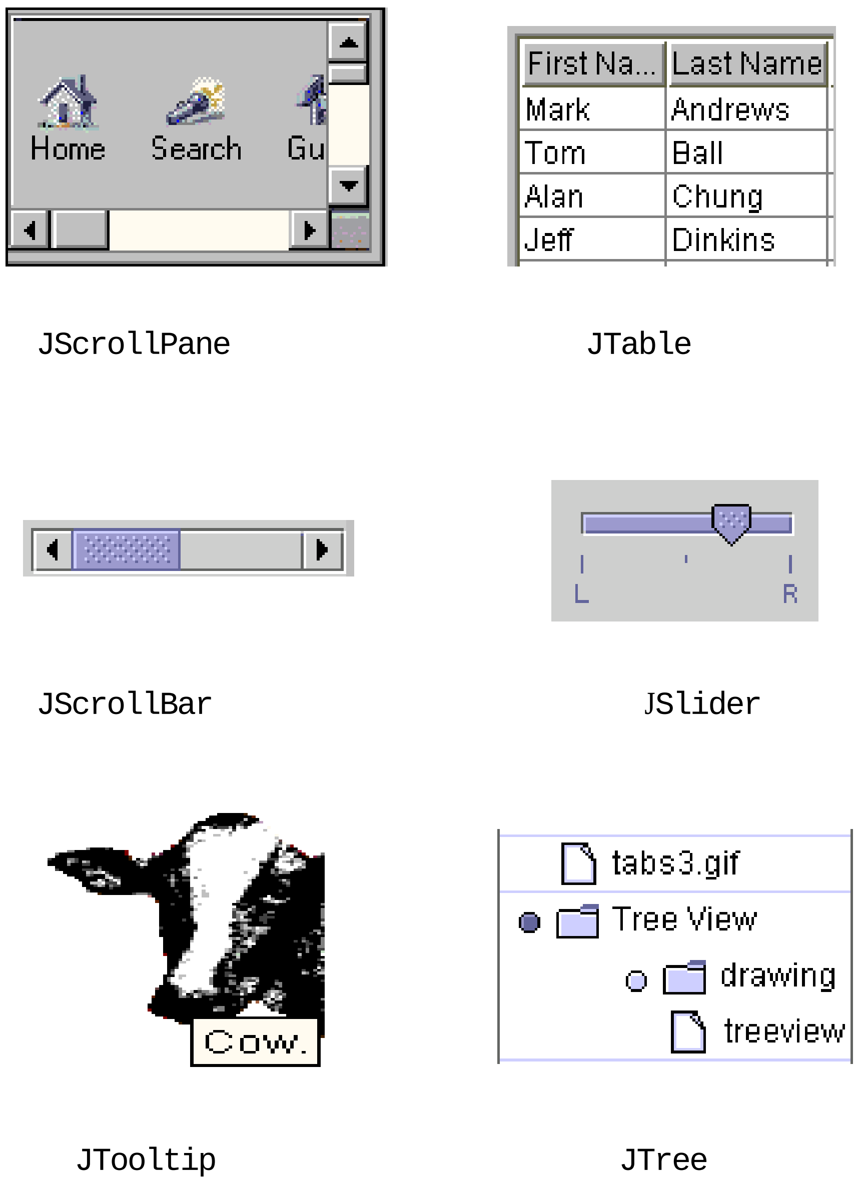
`JList`



`JLabel`

**Figure 11-3** Swing Components

## Swing Components (Continued)



| First Na... | Last Name |
|---|---|
| Mark | Andrews |
| Tom | Ball |
| Alan | Chung |
| Jeff | Dinkins |

**Figure 11-4** Swing Components (Continued)

## Basic Swing Application

The output of the `HelloSwing` application displays the window as shown in Figure 11-5.

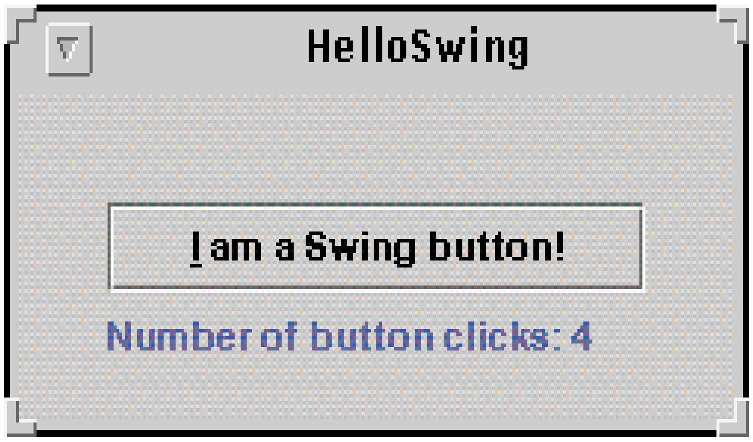


**Figure 11-5** `HelloSwing` Application

Each time the user clicks on a button, the label is updated.

## Basic Swing Application

### HelloSwing

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.accessibility.*;

public class HelloSwing implements ActionListener {
  private JFrame jFrame;
  private JLabel jLabel;
  private JPanel jPanel;
  private JButton jButton;
  private AccessibleContext accContext;

  private String labelPrefix = "Number of button clicks: ";
  private int numClicks = 0;

  public void go() {
    // Here is how you can set up a particular
    // lookAndFeel.  Not necessary for default.
    //
    // try {
    //     UIManager.setLookAndFeel(
    //      UIManager.getLookAndFeel());
    // } catch (UnsupportedLookAndFeelException e) {
    //     System.err.println("Couldn't use the " +
    //       "default look and feel " + e);
    // }

    jFrame = new JFrame("HelloSwing");
    jLabel = new JLabel(labelPrefix + "0");

    jButton = new JButton("I am a Swing button!");

    // Create a shortcut: make ALT-A be equivalent
    // to pressing mouse over button.
    jButton.setMnemonic('i');

    jButton.addActionListener(this);
```

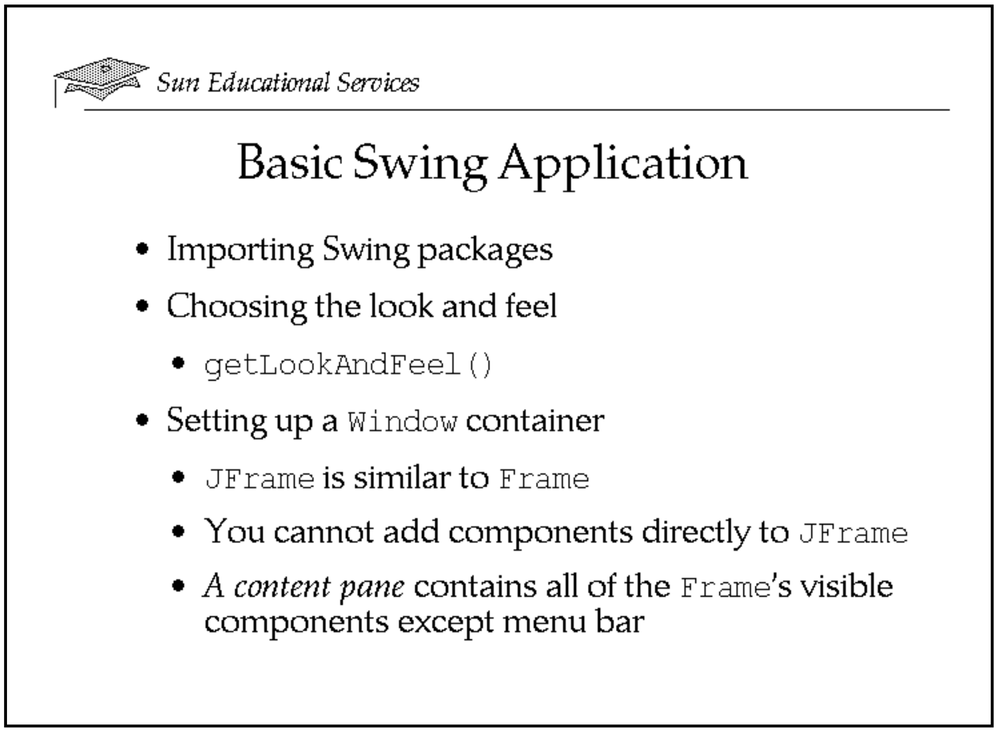
## Basic Swing Application

### HelloSwing

```

    // Add support for accessibility.
    accContext = jButton.getAccessibleContext();
    accContext.setAccessibleDescription(
      "Pressing this button increments " +
      "the number of button clicks");

    // Set up pane.
    // Give it a border around the edges.
    jPanel = new JPanel();
    jPanel.setBorder(
      BorderFactory.createEmptyBorder(30,30,10,30));

    // Arrange for compts to be in a single column.
    jPanel.setLayout(new GridLayout(0, 1));

      // Put compts in pane, not in JFrame directly.
    jPanel.add(jButton);
    jPanel.add(jLabel);
    jFrame.setContentPane(jPanel);

    // Set up a WindowListener inner class to handle
    // window's quit button.
    WindowListener wl = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    jFrame.addWindowListener(wl);

    jFrame.pack();
    jFrame.setVisible(true);
  }

  // Button handling.
  public void actionPerformed(ActionEvent e) {
    numClicks++;
    jLabel.setText(labelPrefix + numClicks);
  }
```

## HelloSwing

```

  public static void main(String[] args) {
    HelloSwing helloSwing = new HelloSwing();
    helloSwing.go();
  }
}

```


Sun Educational Services

## Basic Swing Application

- Importing Swing packages
- Choosing the look and feel
  - `getLookAndFeel()`
- Setting up a `Window` container
  - `JFrame` is similar to `Frame`
  - You cannot add components directly to `JFrame`
  - *A content pane* contains all of the `Frame`'s visible components except menu bar

## *Basic Swing Application*

### *Importing Swing Packages*

The line `import com.sun.java.swing.*` imports the entire Swing package, which includes the standard Swing components and functionality.

### *Choosing the Look and Feel*

Lines 20–26 of `HelloSwing` formats the application's look and feel. The method `getLookAndFeel()` returns the Windows look and feel on a Windows environment. On machines running the Solaris operating system, it returns a common desktop environment (CDE)/Motif look and feel. These lines are not neccessary for this example because they are the default value.

# Basic Swing Application

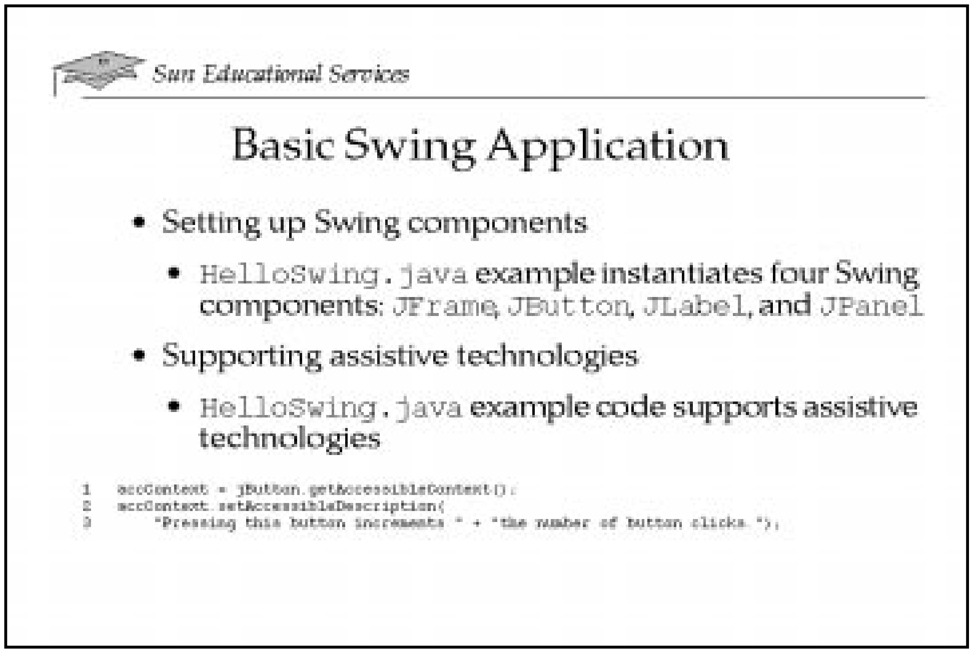
## Setting up Windows

Swing programs implement their primary windows with `JFrame` objects. The `JFrame` class is a subclass of AWT's `Frame` class. It also adds some features found only in Swing. The `HelloSwing` code that deals with its `JFrame` is

```
public HelloSwing() {
  JFrame jFrame;
  JPanel jPanel;
        .....
  jFrame = new JFrame("HelloSwing");
  jPanel = new JPanel();
        .......
  jFrame.setContentPane(jPanel);
```

The code looks much like the code for using a `Frame`. The only difference being that you cannot add components to a `JFrame` directly. Instead, you either add components to the `JFrame`'s content pane, or you provide a new content pane.

A *content pane* is a `Container` that contains all of the frame's visible components except for the menu bar (if there is one). To get a `JFrame`'s content pane, use the `getContentPane()` method. To set its content pane (as shown in the preceding example), use the `setContentPane()` method.



Sun Educational Services

## Basic Swing Application

- Setting up Swing components
  - `HelloSwing.java` example instantiates four Swing components: `JFrame`, `JButton`, `JLabel`, and `JPanel`
- Supporting assistive technologies
  - `HelloSwing.java` example code supports assistive technologies

```
accContent = jButton.getAccessibleContext();
accContext.setAccessibleDescription(
    "Pressing this button increments " + "the number of button clicks.");
```

## *Basic Swing Application*

### *Setting up Swing Components*

The `HelloSwing` program explicitly instantiates four Swing components: a `JFrame`, a `JButton`, a `JLabel` and a `JPanel`. `HelloSwing` uses the code in lines 33–45 to initialize the `JButton`.

✓ ***Emphasize the fact that keyboard navigation is a feature of JFC and that it was not present earlier. Also mention that Swing adds automatic window-close handling to JFrame so that you do not always have to implement a window listener.***

Line 31 creates the button. Line 35 sets the ALT-I key combination as a shortcut used to simulate a button click. Line 37 registers an event handler for the click. Lines 40–43 describe a button, so that assistive technologies can provide information on the button's functionality.

Lines 47–57 initialize the `JPanel`. They create the `JPanel` object, give it a border, and set its layout manager to one that will put the panel's contents in a single column. Finally, a button and a label are added to the `Panel`. The `Panel` in `HelloSwing` uses an invisible border to put extra padding around it.

## Basic Swing Application

### Supporting Assistive Technologies

The only code in HelloSwing.java that exists solely to support the assistive technologies is

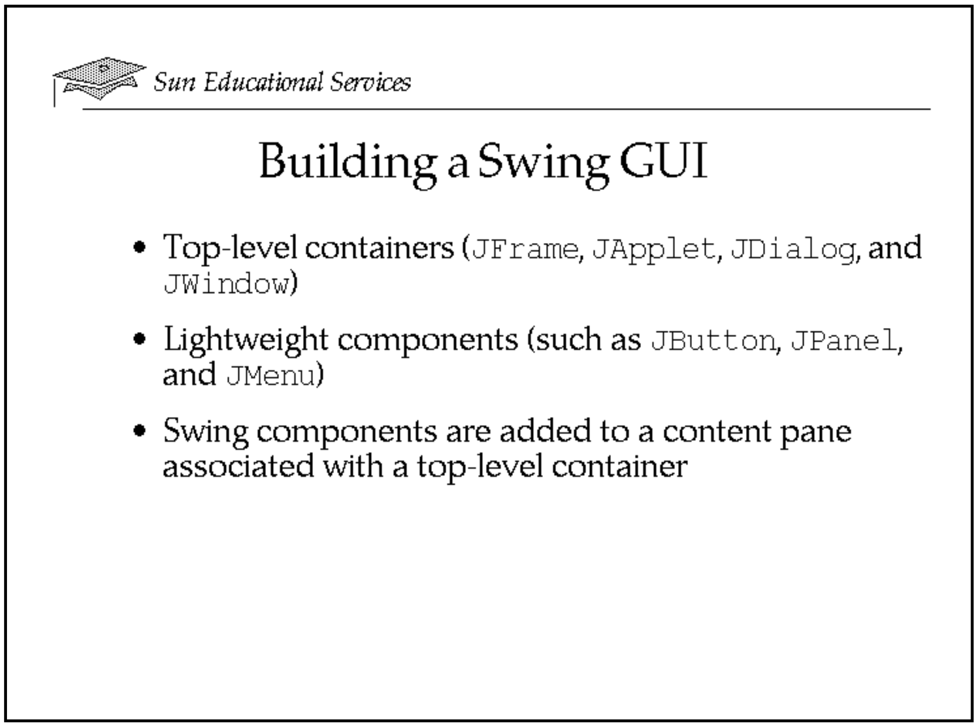
```
accContext = jButton.getAccessibleContext();
accContext.setAccessibleDescription(
   "Pressing this button increments ″ +
   ″the number of button clicks.");
```

The following set of information can also be used by assistive technologies:

```
jButton = new JButton("I’m a Swing button!”);
jLabel = new JLabel(labelPrefix + "0");
jLabel.setText(labelPrefix + numClicks);
```

Accessibility support is built into JFrame, JButton, JLabel, and all other Swing components. Assistive technologies can easily get the text of or the text associated with a specific part of a component.

✓ ***Emphasize the advantages of using assistive technologies.***


Sun Educational Services

## Building a Swing GUI

- Top-level containers (`JFrame`, `JApplet`, `JDialog`, and `JWindow`)
- Lightweight components (such as `JButton`, `JPanel`, and `JMenu`)
- Swing components are added to a content pane associated with a top-level container

## *Building a Swing GUI*

The Swing package defines two types of components:

- Top-level containers (`JFrame`, `JApplet`, `JWindow`, and `JDialog`)
- Lightweight components (*Jeverything-else*, such as `JButton`, `JPanel`, and `JMenu`)

The top-level containers provide the framework in which the lightweight components exist. Specifically, a top-level Swing container provides an area in which lightweight Swing components can draw themselves. Top-level containers are Swing subclasses of their heavyweight AWT component counterpart. These Swing containers rely on the native code of their AWT superclass to properly interface with the hardware.

## Building a Swing GUI

In general, every Swing component should have a top-level Swing container above it in its containment hierarchy. For example, every applet containing Swing components should be implemented as a subclass of `JApplet` (which is itself a subclass of `java.applet.Applet`). Similarly, every main window that contains Swing components should be implemented with a `JFrame`. Typically, if you are using Swing components, you will use only Swing components and Swing containers.

Swing components can be added to a content pane which is associated with a top-level container, but can never be added to the top-level container directly.

Here's a picture of the GUI *containment* hierarchy for a typical Swing program that implements a window containing two buttons, a text field, and a list.

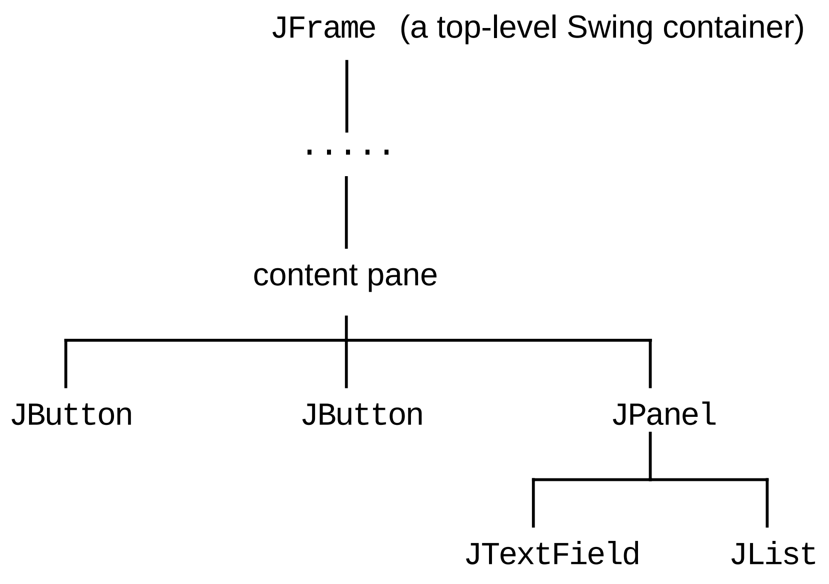


**Figure 11-6** GUI Containment Hirearchy

## Building a Swing GUI

Here's another *containment* hierarchy figure for the same GUI, except that the GUI is in an applet running in a browser:

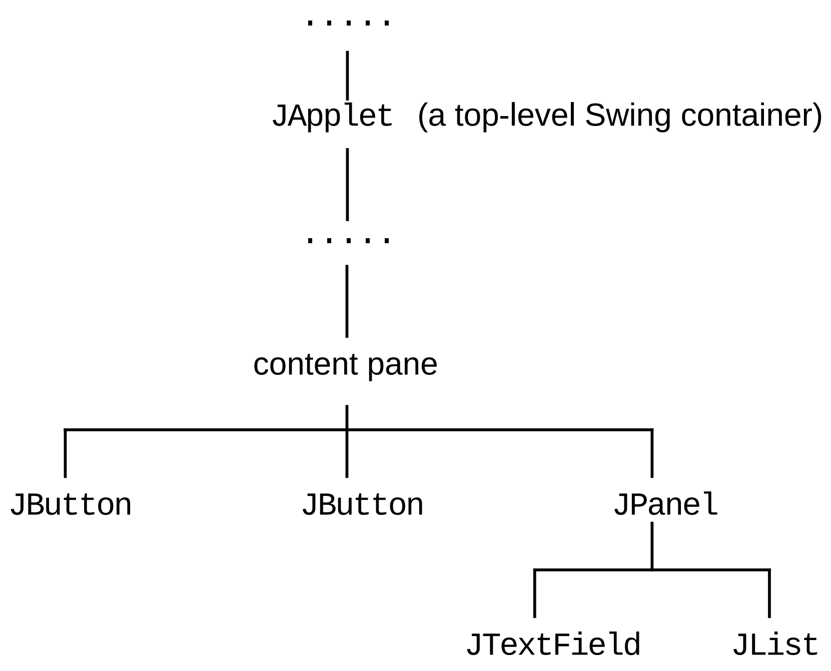


**Figure 11-7** Containment Hirearchy of an Applet Running in a Browser

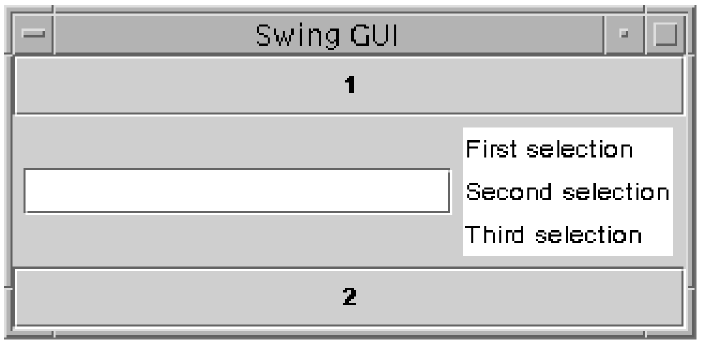
## Building a Swing GUI

Here's the code that constructs the GUI hierarchies shown in the preceding figures:

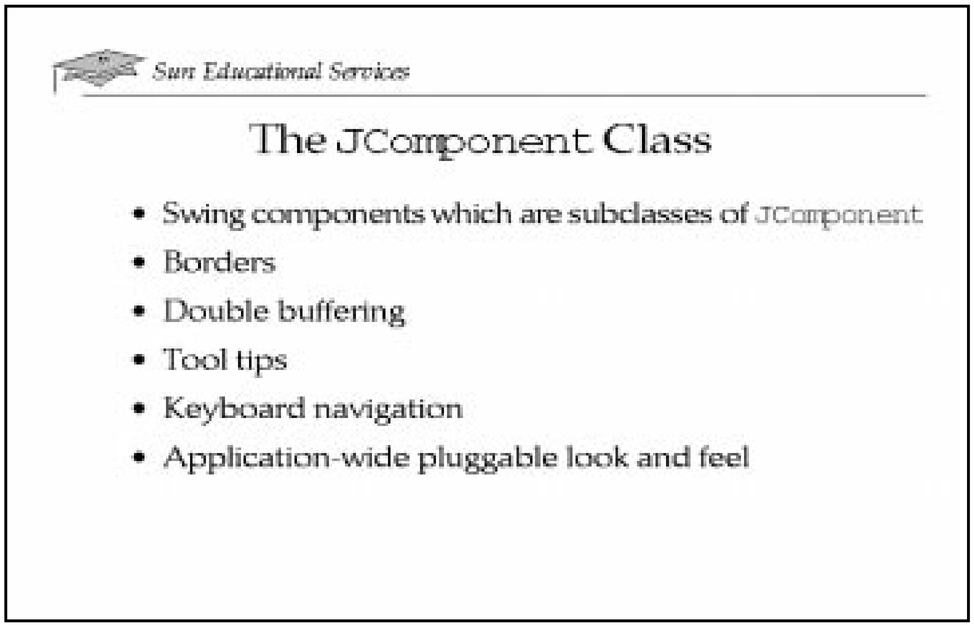
```
import java.awt.*;
import com.sun.java.swing.*;

public class SwingGUI {
  private JFrame topLevel;
  private JPanel jPanel;
  private JTextField jTextField;
  private JList jList;

  private JButton b1;
  private JButton b2;
  private Container contentPane;

  private Object listData[] = {
    new String("First selection"),
    new String("Second selection"),
    new String("Third selection")
  };

  public void go() {
    topLevel = new JFrame("Swing GUI");

    // Set up the JPanel, which contains the text field
    // and list.
    jPanel = new JPanel();
    jTextField = new JTextField(20);
    jList = new JList(listData);

    contentPane = topLevel.getContentPane();
    contentPane.setLayout(new BorderLayout());

    b1 = new JButton("1");
    b2 = new JButton("2");
    contentPane.add(b1, BorderLayout.NORTH);
    contentPane.add(b2, BorderLayout.SOUTH);

    jPanel.setLayout(new FlowLayout());
    jPanel.add(jTextField);
    jPanel.add(jList);
    contentPane.add(jPanel, BorderLayout.CENTER);
```

## Building a Swing GUI

```

    topLevel.pack();
    topLevel.setVisible(true);
  }

  public static void main (String args[]) {
    SwingGUI swingGUI = new SwingGUI();
    swingGUI.go();
  }
}
```



✓ *In general, you should avoid using heavyweight components in Swing GUI's (except for the top-level Swing container that hosts the GUI, of course). The most noticable problem with mixing heavyweight and lightweight components is that when they overlap within a container, the heavyweight component is always drawn on top of (that is, in front of) the lightweight component.*


Sun Educational Services

The JComponent Class

- Swing components which are subclasses of JComponent
- Borders
- Double buffering
- Tool tips
- Keyboard navigation
- Application-wide pluggable look and feel

## *The* `JComponent` *Class*

All Swing components are implemented as subclasses of the `JComponent` class, which inherits from the `Container` class. Swing components inherit the following functionality from `JComponent:`

- Borders

  Using the `setBorder()` method, you can specify the border that a component displays around its edges. You can specify that a component have extra space around its edges using an `EmptyBorder` instance.

- Double buffering

  Double buffering can improve the appearance of a frequently changing component. Now you do not have to write the double buffering code – Swing provides it for you. By default, Swing components are double buffered.

## The JComponent Class

- Tool tips

  By specifying a string with the `setToolTipText()` method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.

- Keyboard navigation

  Using the `registerKeyboardAction()` method, you can enable the user to use the keyboard, instead of the mouse, to maneuver around the GUI. The combination of character and modifier keys that the user must press to start an action is represented by a `KeyStroke` object.

- Application-wide pluggable look and feel

  Each Java application runtime has a `UIManager` object that determines the look and feel of that runtime's Swing components. Subject to security restrictions, you can choose the look and feel used by all Swing components by invoking the `UIManager.setLookAndFeel()` method. Behind the scenes, each `JComponent` object has a corresponding `ComponentGUI` object that performs all drawing, event handling, size determination, and such for that `JComponent`.

## Exercise: Getting Acquainted With Swing

**Exercise objective –** In this lab you write, compile, and run two programs that use Swing components in a GUI.

### Preparation

In order to successfully complete this lab, you must understand how Swing components and AWT components are related.

### Tasks

#### About the Labs

✓ ***For the Level 1 lab, students must be familiar with the concept of associating an image with a button and the concept of tooltip.***

#### *Level 1 Lab: Create a Basic Swing Application*

Complete these steps:

1. Using any text editor, create an application similar to `HelloSwing` discussed earlier.
2. Associate an image-icon with the button. (Hint – You might have to use the `ImageIcon` class.)
3. Associate a tooltip with the button so that when the mouse moves over the button, a tooltip "JFC Button" is displayed.

✓ ***The Level 2 lab introduces*** `JToolBar`***,*** `JMenuBar`***,*** `JMenuItem`***,*** `JMenu`***, and*** `JDialog` ***to distinguish them from their AWT counterparts. Also, part of this example deals with file handling which is covered in a later module.***

## Exercise: Getting Acquainted With Swing

### Tasks

#### Level 2 Lab: Create a Text Editor Using Swing Components

Perform the following steps:

1. Construct an initial `JFrame` to have a `JToolBar`, `TextArea`, and `JLabel`.
2. Associate a `JMenuBar` with the `JFrame`.
3. Set up the first menu on the `JMenuBar`. Create a `JMenu` labeled File with New, Open, Save, and Close `JMenuItem`s.
4. Add a keyboard accelerator to each item. Use the first letter of the label.
5. For each `JMenuItem`, create an anonymous `ActionListener` to process the event and call the appropriate method to handle the respective event.

## Exercise: Creating Swing Applications

### Tasks

#### Level 2 Lab: Create a Text Editor Using Swing Components (Continued)

6. Add a Help `JMenu` with an About `JMenuItem` to a `JMenuBar`. Add a key accelerator for H and A, respectively.

7. Create a modal `JDialog` in the associated event handler for the About `JMenuItem`.

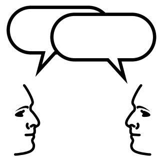
8. Set up four `JButton`s to be placed on the toolbar, labeled New, Open, Save and About.

9. For each button on the toolbar, add a tooltip, with an appropriate message. Also, create an anonymous `ActionListener` to handle the appropriate event.

10. Save and compile the program.

## Exercise: Creating Swing Applications

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Identify the key features of Java Foundation Classes
- ❑ Describe the key features of `com.sun.java.swing` package
- ❑ Identify Swing components
- ❑ Define *containers* and *components*, and explain how they work together to build a Swing GUI
- ❑ Write, compile, and run a basic Swing application
- ❑ Use top-level containers such as `JFrame` and `JApplet` effectively

## Notes

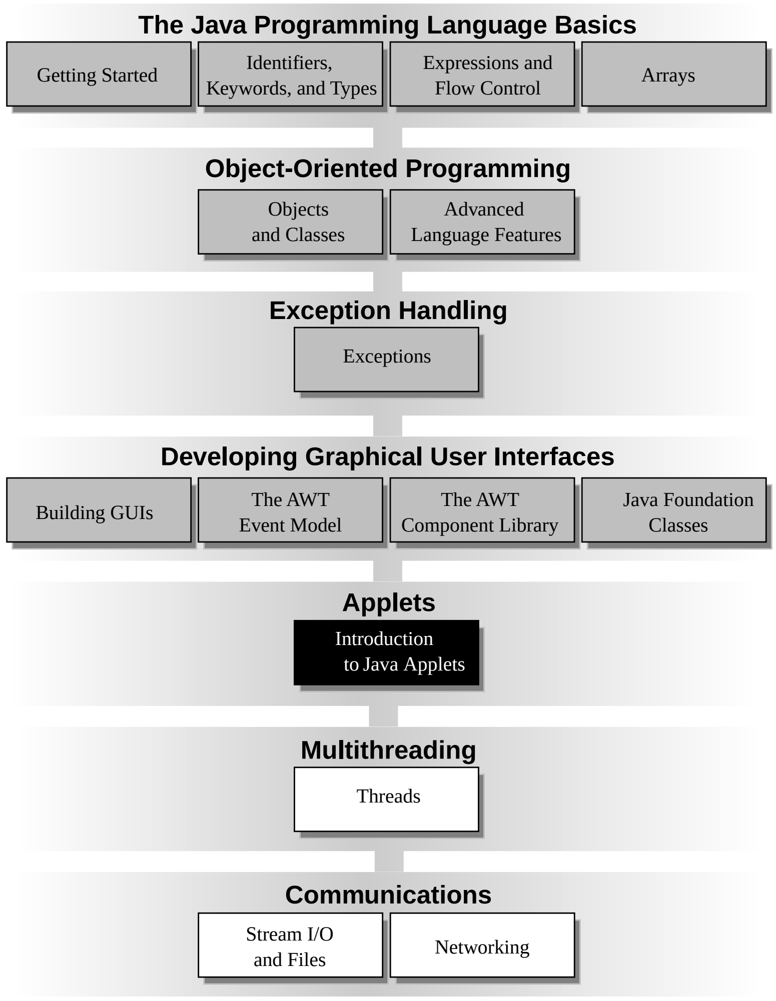
## Think Beyond

You now know how to program GUI applications. Suppose you want to run a GUI application inside a Web browser. How is this done?

# Introduction to Java Applets 12

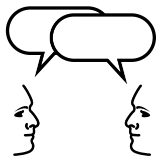
## Course Map

This module discusses the support for applets by the JDK, and how applets differ from applications in terms of program form, operating context, and how they are started.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

- What advantages do applets provide?

✓ ***It is clear from the Java platform's graphics and network security features that the Web is a significant niche for Java programs. Until now, only standalone Java software programs have been discussed. This module discusses what you need to know to run Java software programs using a Web browser, allowing access over the Web.***

✓ ***There are significant differences between application and applet operating contexts; this module discusses the differences between the contexts, and shows how to code applets to operate in their context.***

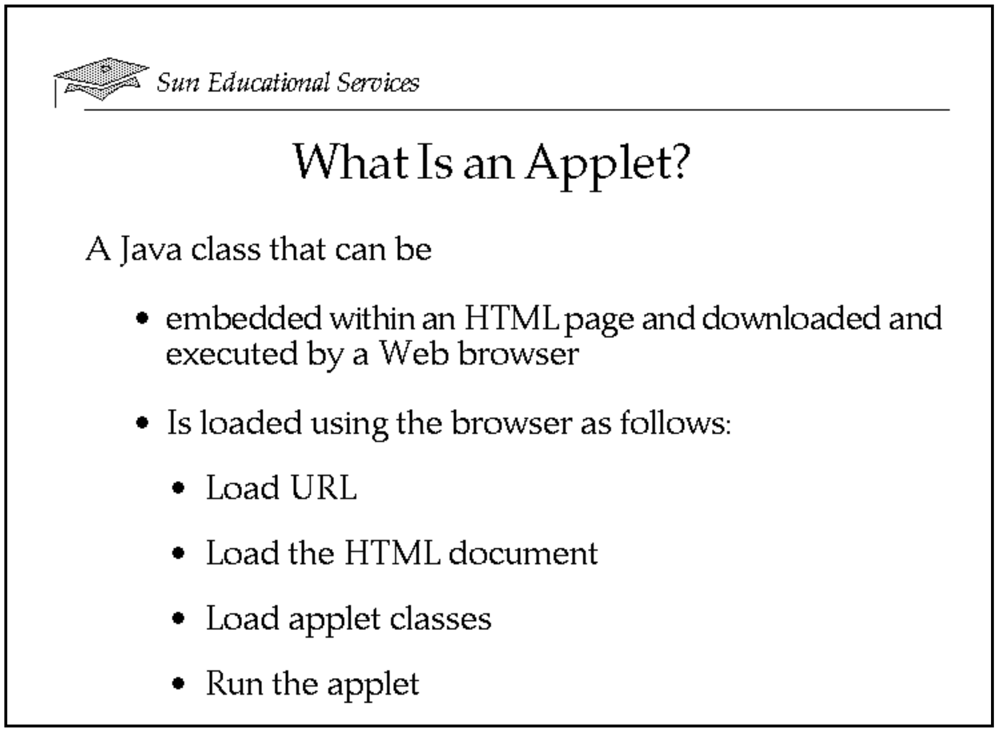
✓ ***Applets interact with their environment differently than applications do. This module discusses how applets are started, and describes enough HTML to start an applet from a Web page.***

✓ ***The "hooks" or places (that is, methods) by which the environment runs an applet differ from those for applications (which have a "main" method). This module explores these "hook" methods, and when and how they are called by their environment. Additionally, the Component*** `paint()` ***mechanism which updates the screen when its content is changed or otherwise needs redisplay is described.***

## Objectives

Upon completion of this module, you should be able to

- Differentiate between a stand-alone application and an applet
- Write an HTML tag to call a Java applet
- Describe the class hierarchy of the applet and AWT classes
- Create the `HelloWorld.java` applet
- List the major methods of an applet
- Describe and use the painting model of AWT
- Use applet methods to read images and files from URLs
- Use `<param>` tags to configure applets



Sun Educational Services

## What Is an Applet?

A Java class that can be

- embedded within an HTML page and downloaded and executed by a Web browser
- Is loaded using the browser as follows:
  - Load URL
  - Load the HTML document
  - Load applet classes
  - Run the applet

## What Is an Applet?

An *applet* is a Java class that can be embedded within an HTML page, and is downloaded and executed by a Web browser. It is a specific type of Java technology container. It differs from an application in the way it is executed. An application is started when its `main()` method is called. The life cycle of an applet is somewhat more complex. This module examines how an applet is run, how it is loaded into the browser, and how it is written.

### Loading an Applet

An applet runs in the environment of a Web browser, so it is not started directly by typing a command. You must create an HTML file that tells the browser what to load and how to run it. You then “point” the browser at the URL that specifies that HTML file. (The format of the HTML file is discussed later in this module.)

## What Is an Applet?

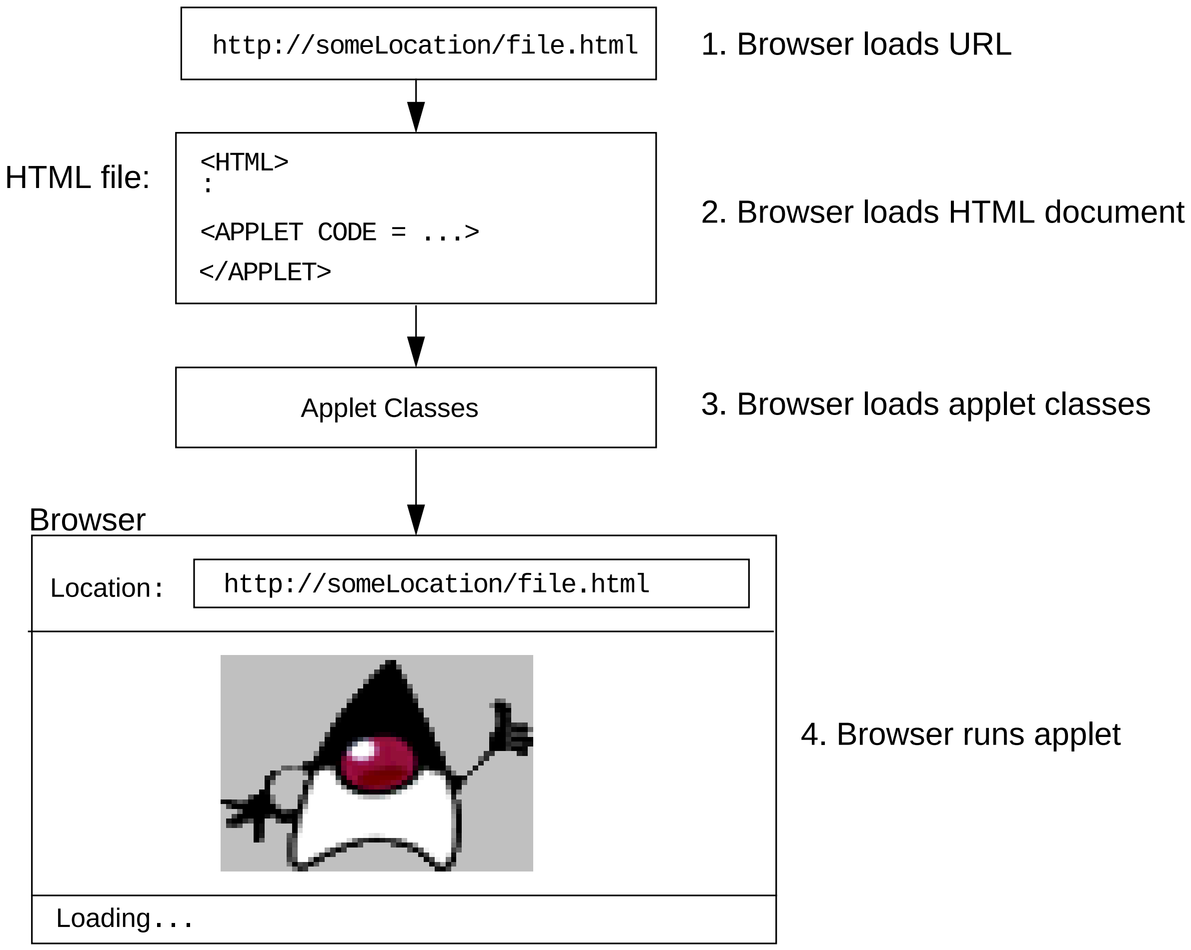
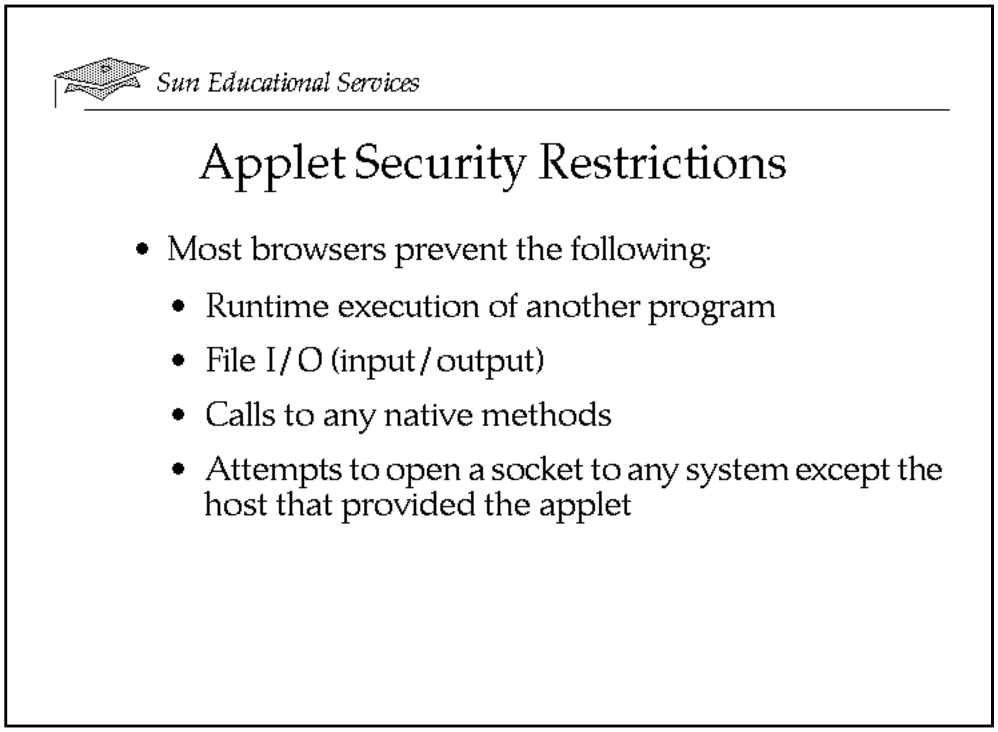


**Figure 12-1** Running an Applet



Sun Educational Services

## Applet Security Restrictions

- Most browsers prevent the following:
  - Runtime execution of another program
  - File I/O (input/output)
  - Calls to any native methods
  - Attempts to open a socket to any system except the host that provided the applet

## What Is an Applet?

### Applet Security Restrictions

Applets are pieces of code that represent an inherently dangerous prospect because they are loaded over a network. What if someone writes a malicious class that reads your password file and sends it over the Internet?

The depth to which security is controlled is implemented at the browser level. Most browsers (including Netscape Navigator) prevent the following by default:

- Runtime execution of another program
- File I/O
- Calls to any native methods
- Attempts to open a socket to any system except the host that provided the applet

## What Is an Applet?

### Applet Security Restrictions

The point of these restrictions is to prevent an applet from violating privacy of, or damaging a remote system, by restricting the applet's access to the files of that system. Preventing execution of another program and disallowing calls to native methods restrict the applet from starting code which runs unchecked by the JVM. The restriction on sockets prevents communication with another program which is stored on a non-trusted host.

JDK1.2 provides a means of specifying a particular "protection domain," or security environment, in which a particular applet is to be run. A remote system checks the originating URL and signature of the applet it downloads against a local file containing entries mapping special applets to special protection domains. Thus special applets coming from particular places can be run with special privileges.

## Writing an Applet

To write an applet, you must create a class using this form:

```
import java.applet.*;

public class HelloWorld extends Applet {
```

The applet's class must be `public` and its name must match the name of the file it is in; in this case, `HelloWorld.java`. Furthermore, the class must be a subclass of the class `java.applet.Applet`.

### Applet Class Hierarchy

The `java.applet.Applet` class is actually a subclass of `java.awt.Panel`. The hierarchy of the applet and AWT classes is as follows:

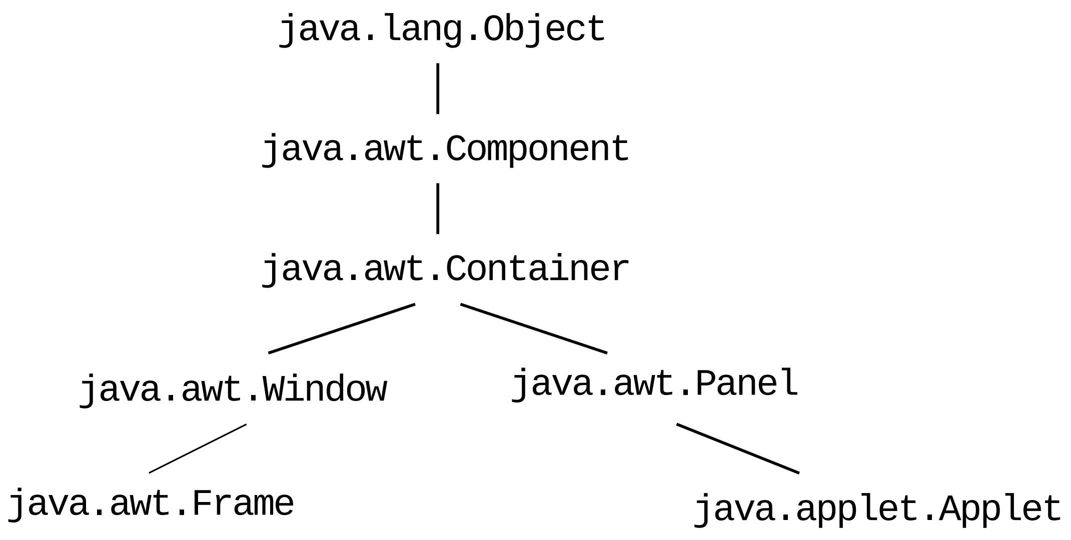
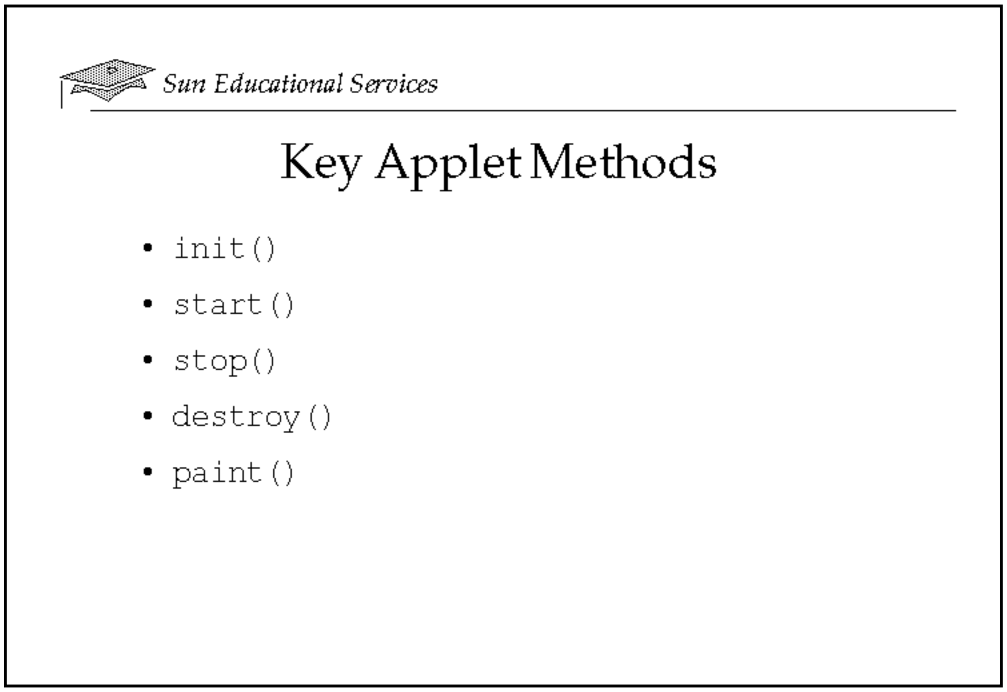


**Figure 12-2** Applet and AWT Class Hirearchies

This hierarchy shows that an applet can be used directly as the starting point for an AWT layout. Because an applet is a `Panel`, it has a `FlowLayout` manager by default. The methods of the `Component`, `Container`, and `Panel` classes are inherited by the `Applet` class.


Sun Educational Services

## Key Applet Methods

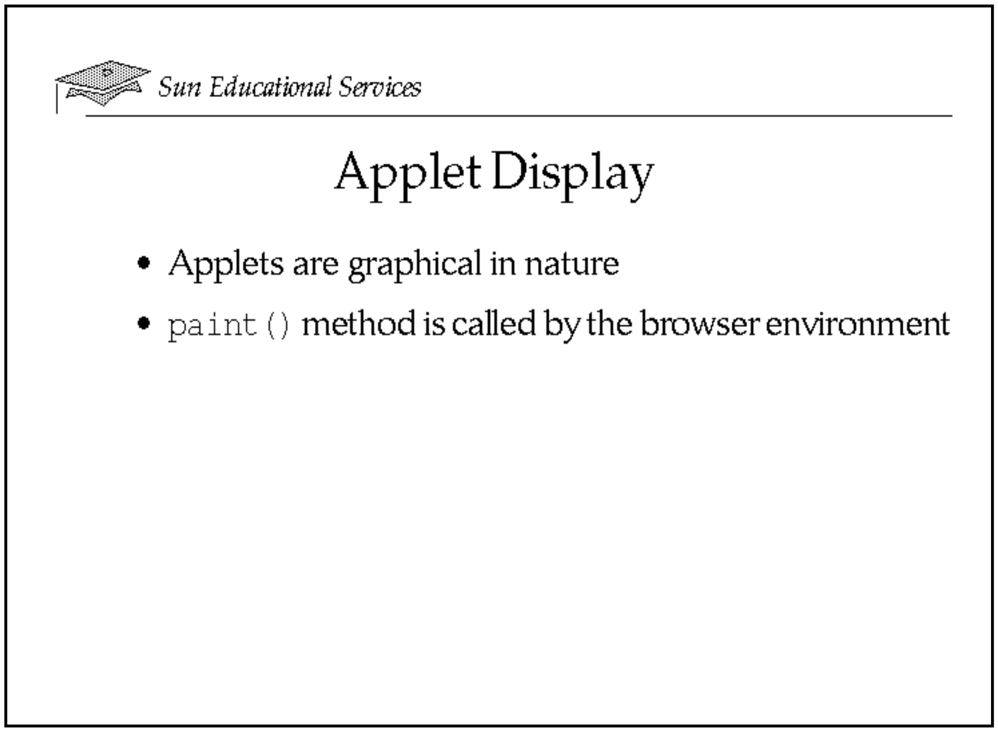
- init()
- start()
- stop()
- destroy()
- paint()

## *Writing an Applet*

### *Key Applet Methods*

In an application, the program is entered when the `main()` method is called. In an applet, however, this is not the case. After the constructor completes its task, the browser calls `init()` to perform basic initialization of the applet. After `init()` completes its task, the browser calls the method `start()`. `start()` will be examined more closely later in this module; in general, it is called when the applet becomes visible.

Both the `init()` and `start()` methods run to completion before the applet becomes "live," and because of this they cannot be used to program ongoing behavior into an applet. In fact, unlike the `main()` method in a simple application, there is no method that is executed continuously throughout the "life" of the applet. You will see later how to do this using threads. Additional methods you write for your applet subclass can include `stop()`, `destroy()` and `paint()`.



Sun Educational Services

## Applet Display

- Applets are graphical in nature
- `paint()` method is called by the browser environment

## Writing an Applet

### Applet Display

Applets are essentially graphical in nature, so although you can issue `System.out.println()` calls, you will not normally do so. Instead, you will create your display in a graphical environment.

You can draw on an applet's panel by creating a `paint()` method. The `paint()` method is called, by the browser environment, whenever the applet's display needs refreshing. This happens, for example, when the browser window is displayed after being minimized or iconified.

Write your `paint()` method so that it works properly whenever it is called. Exposure happens asynchronously and is driven by the environment, not the program.

✓ ***The `paint()` method is not limited to use in applets. Canvases and Frames use `paint()` as well.***

## Writing an Applet

### *The* `paint()` *Method and the Graphics Object.*

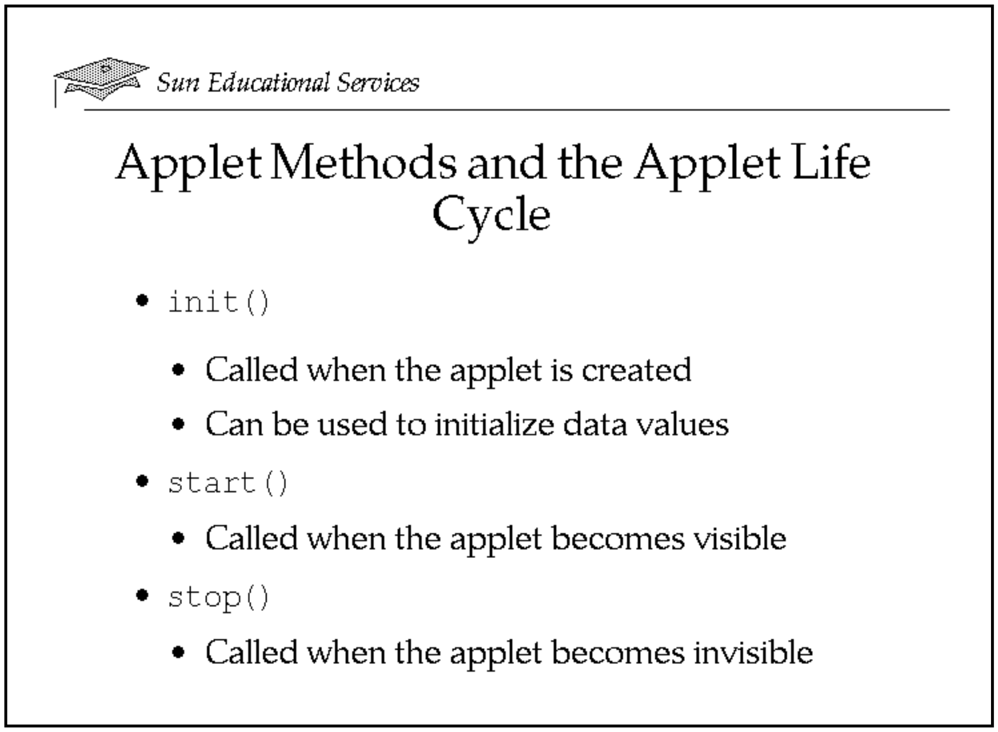
The `paint()` method takes an argument which is an instance of the `java.awt.Graphics` class. The argument is always the graphics context of the panel that makes up the applet (remember, `Applet extends Panel`). You can use this context to draw or write into your applet. The following is an example of an applet that uses a `paint()` method to write some text:

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet {
  public void paint(Graphics g){
    g.drawString("Hello World!", 25, 25);
  }
}
```

---

**Note –** The numeric arguments to the `drawString` method are the x and y pixel coordinates for the start of the text. (0,0) represents the upper left corner. These coordinates refer to the baseline of the font, so writing at y coordinate zero will result in the bulk of your text being off the top of the display, only the descenders like the tail of the letter *p* will be visible.

---



Sun Educational Services

## Applet Methods and the Applet Life Cycle

- `init()`
  - Called when the applet is created
  - Can be used to initialize data values
- `start()`
  - Called when the applet becomes visible
- `stop()`
  - Called when the applet becomes invisible

## *Applet Methods and the Applet Life Cycle*

The applet life cycle is a little more complex than what has been discussed so far. There are three major methods that relate to its life cycle: `init()`, `start()`, and `stop()`.

### init()

This member function is called at the time the applet is created and loaded into a browser capable of supporting Java technology (such as the `appletviewer`). The applet can use this method to initialize data values. The `init()` method runs to completion before `start()` is called.

```
public void init() {
  // set up GUI
}
```

## Applet Methods and the Applet Life Cycle

### start()

Once the `init()`method is completed, the `start()` method executes which causes the applet to become "live." It also runs whenever the applet becomes visible, such as when the browser is restored after being iconized or when the browser returns to the page containing the applet after moving to another URL. This method is used typically to start threads or an animation or to play sounds.

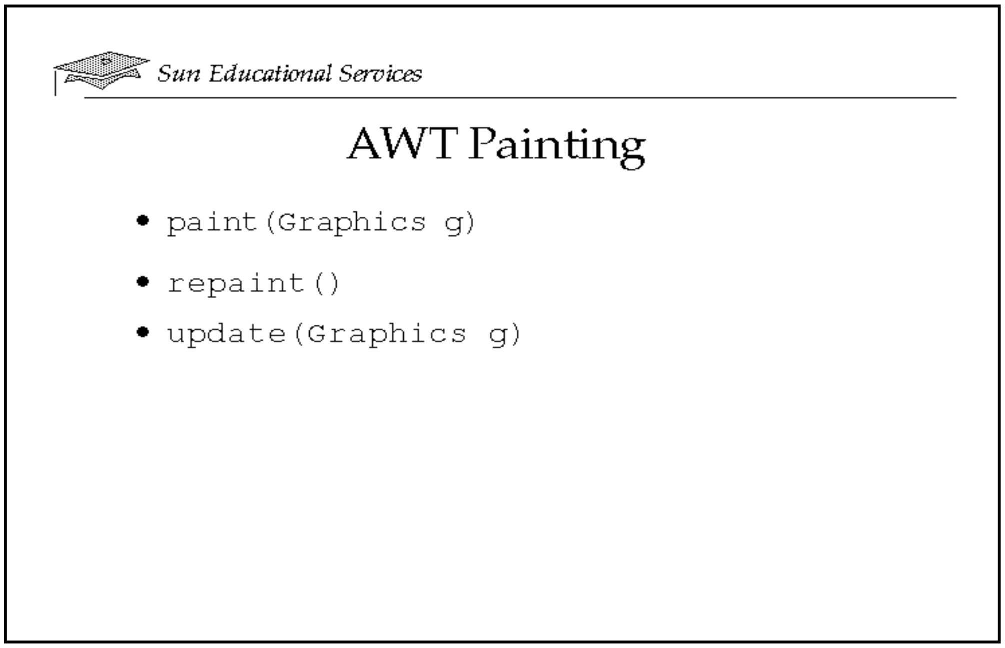
```
public void start() {
  musicClip.play();
}
```

### stop()

The `stop()` method is called when the applet becomes invisible. This happens when the browser is iconified or it follows a link to another URL. The applet uses this method to stop any functionality that should not occupy the CPU when the applet is not on the current browser page.

```
public void stop() {
  musicClip.stop();
}
```

The `start()` and `stop()` methods effectively form a pair. Typically `start()` activates a behavior in an applet and `stop()` deactivates the behavior.



Sun Educational Services

## AWT Painting

- `paint(Graphics g)`
- `repaint()`
- `update(Graphics g)`

## *AWT Painting*

In addition to the basic life cycle methods, an applet has important methods related to its display. These methods are declared and documented for the AWT `Component` class. It is important to adhere to the correct model for display handling using the AWT.

Updating the display is done by a separate thread which is referred to as the *AWT thread*. This thread can be called upon to handle two situations that relate to updating the display.

The first of these conditions is exposure; either when the display is first exposed, or where part of the display has been damaged and must be replaced. Display damage can occur at any time, and your program must be able to update the display at any time.

The second condition is when the program redraws the display with new contents. This redrawing might require the old image be removed first.

## *AWT Painting*

### *The* `paint()` *Method*

Exposure handling occurs automatically and results in a call to the `paint()` method. A facility of the `Graphics` class, called a clip rectangle, is used to optimize the `paint()` method so that updates are not made over the entire area of the graphics unless necessary. Rather, these updates are restricted to the region that has been damaged. Override the `paint()` method to control what is painted on your applet.

```
    public void paint(Graphics g) {...}
```

### *The* `update(Graphics g)` *Method*

`repaint()` actually causes the AWT thread to call another method, `update()`. The `update` method usually clears the current display and calls `paint()`. The `update()` method can be modified, for example, to reduce flicker by calling `paint()` without clearing the display.

### *The* `repaint()` *Method*

A call to `repaint()` notifies the system that you want to change the display.

## *AWT Painting*

### *Method Interaction*

The following diagram shows how the `paint()`, `update()`, and `repaint()` methods are related:

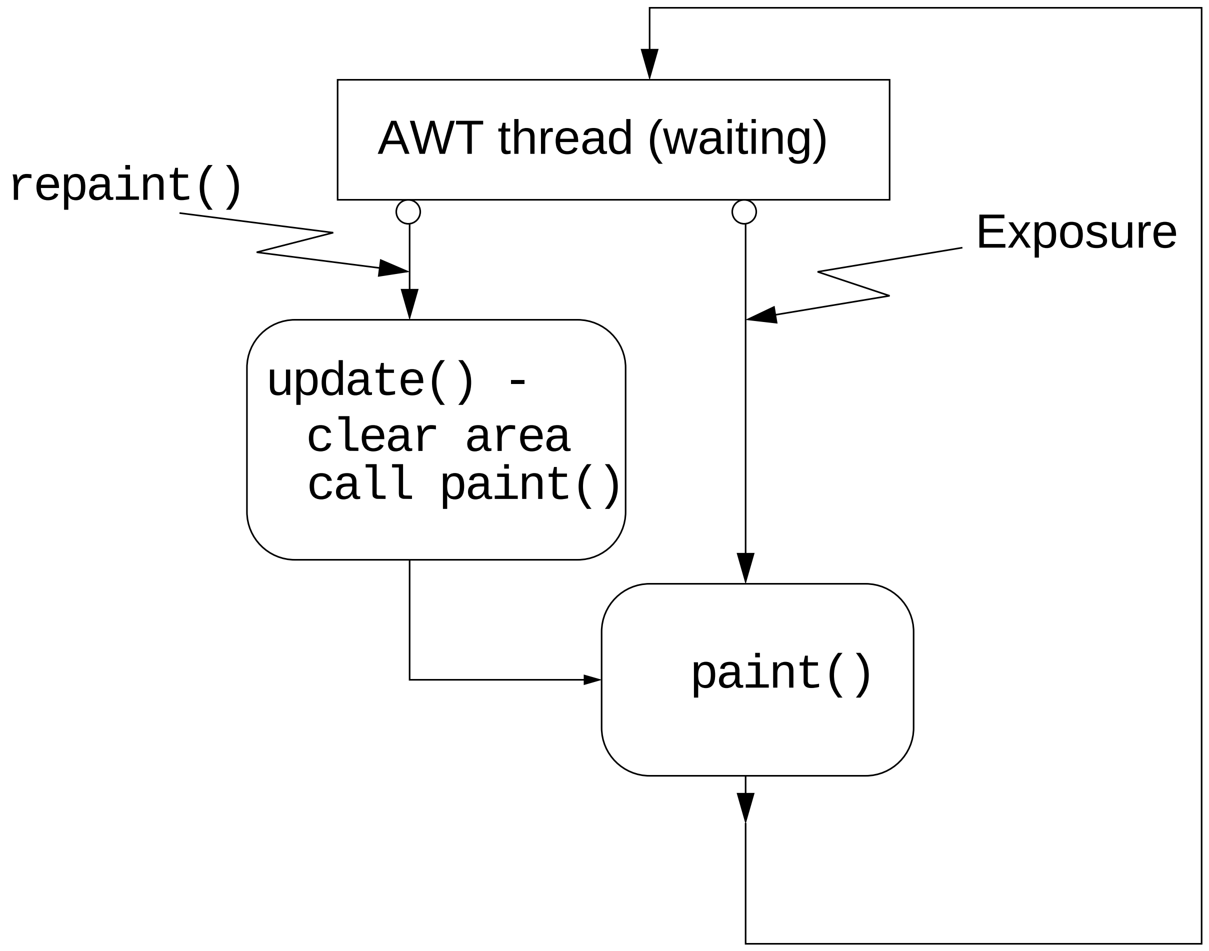
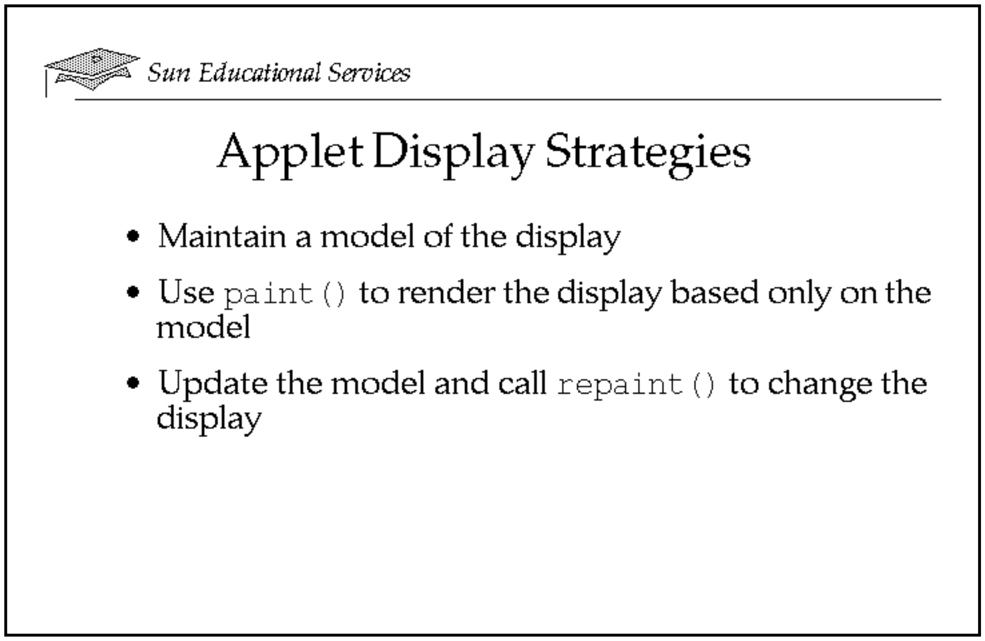


**Figure 12-3** Method Relationships


Sun Educational Services

## Applet Display Strategies

- Maintain a model of the display
- Use `paint()` to render the display based only on the model
- Update the model and call `repaint()` to change the display

## *AWT Painting*

### *Applet Display Strategies*

The applet model requires that you adopt a specific strategy for maintaining your display. You must

- Maintain a model of the display. This model is a definition of what to do to re-render the display. Instructions on how to do this are embodied in the `paint()` method; the data used by these instructions are usually held as class-level variables.
- Have the `paint()` method *render* the display based *only* on the contents of the model. This allows `paint()` to regenerate the display consistently whenever it is called, and handle exposure correctly.

## AWT Painting

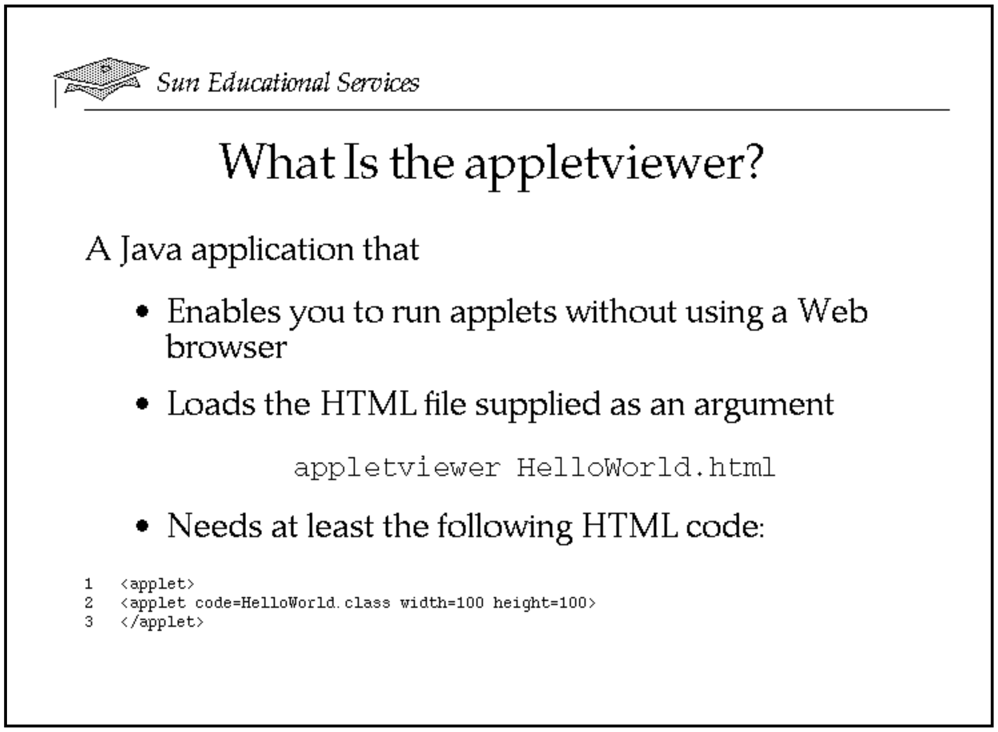
### Applet Display Strategies (Continued)

- Have the program change the display by updating the model and then calling the `repaint()` method so that the `update()` method (and ultimately the `paint()` method) gets called by the AWT thread.

---

**Note –** A single AWT thread handles all component painting and the distribution of input events. Keep `paint()` and `update()` simple to avoid them stalling the AWT thread. In extreme cases, you will need the help of other threads to achieve this. Thread programming is the subject of another module.

---


Sun Educational Services

What Is the appletviewer?

A Java application that

- Enables you to run applets without using a Web browser
- Loads the HTML file supplied as an argument

```
appletviewer HelloWorld.html
```

- Needs at least the following HTML code:

```
<applet>
<applet code=HelloWorld.class width=100 height=100>
</applet>
```

## *The appletviewer*

### *What Is the appletviewer?*

An applet is run usually inside a web browser such as HotJava™ or the Netscape Navigator, which is capable of running Java software programs. To simplify and speed up development, the JDK comes with a tool designed only to view applets, not HTML pages. This tool is the appletviewer.

The appletviewer is a Java application that enables you to run applets without using a Web browser. It resembles a *minimum browser.*

The appletviewer reads the HTML file specified by the URL on the command line. This file must contain the instructions for loading and running one or more applets. The appletviewer ignores all other HTML code. It does not display normal HTML or embedded applets in a text page.

✓ ***Explain that the appletviewer has a subset of the functionality of a browser and is designed so that applets created now will work with minimal changes later.***

## *The appletviewer*

### *Starting Applets With the appletviewer*

The appletviewer posts a frame-like space onto the screen, instantiates an instance of the applet, and posts that applet instance to the Frame.

The appletviewer takes as a command-line argument a URL to an HTML file containing an applet reference. This applet reference is an HTML tag that specifies the code that the `appletviewer` loads; for example:

```
<applet code=HelloWorld.class width=100 height=100>
</applet>
```

✓ ***There can be other tags between*** `applet` ***and*** `/applet`***; these are covered later.***

Notice that the general format of this tag is the same as any other HTML, using the < and > symbols to delimit the instructions. All the parts shown here are required. You must have both `<applet ...>` and `</applet>`. The `<applet ...>` part specifies a code entry and a width and height.

---

**Note –** In general, you should treat applets as being of fixed size and use the size specified in the `<applet>` tag.

---

## Using the appletviewer

### Synopsis

The appletviewer takes a URL to an HTML file containing the `<applet>` tag as a command-line argument.

```
appletviewer [-debug] URLs ...
```

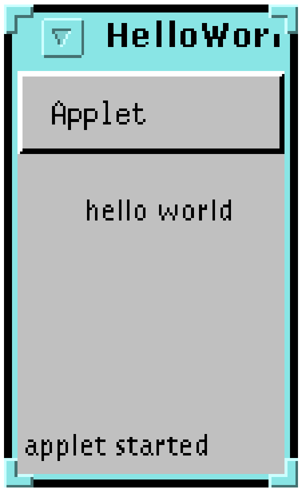
The only valid option to the appletviewer is the `-debug` flag which starts the applet in the Java debugger, `jdb`. Compile your Java code with the `-g` option to see the source code in the debugger.

### Example

The following `appletviewer` command starts the appletviewer:

```
appletviewer HelloWorld.html
```

This creates and displays the small windows in Figure 12-4..



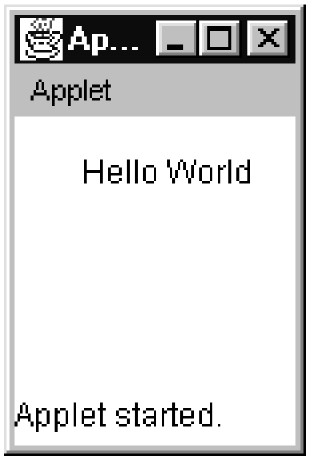


**Figure 12-4** `HelloWorld` Applets

## The applet Tag

### Syntax

The complete syntax for the applet tag is:

```
<applet
   [archive=archiveList]
   code=appletFile.class
   width=pixels height=pixels
   [codebase=codebaseURL]
   [alt=alternateText]
   [name=appletInstanceName]
   [align=alignment]
   [vspace=pixels] [hspace=pixels]
>
[<param name=appletAttribute1 value=value>]
[<param name=appletAttribute2 value=value>]
    . . .
[alternateHTML]
</applet>
```

where

- archive = *archiveList* – This optional attribute describes one or more archives containing classes and other resources that will be "preloaded." The classes are loaded using an instance of an AppletClassLoader with the given codebase. The archives in *archiveList* are separated by a comma (,).
- code = *appletFile.class* – This *required* attribute gives the name of the file that contains the compiled Applet subclass. This can also be in the format *package.appletFile.class*.

---

**Note –** This file is relative to the base URL of the HTML file you are loading it from. It *cannot* include a path name. To change the base URL of the applet, use the <codebase> tag.

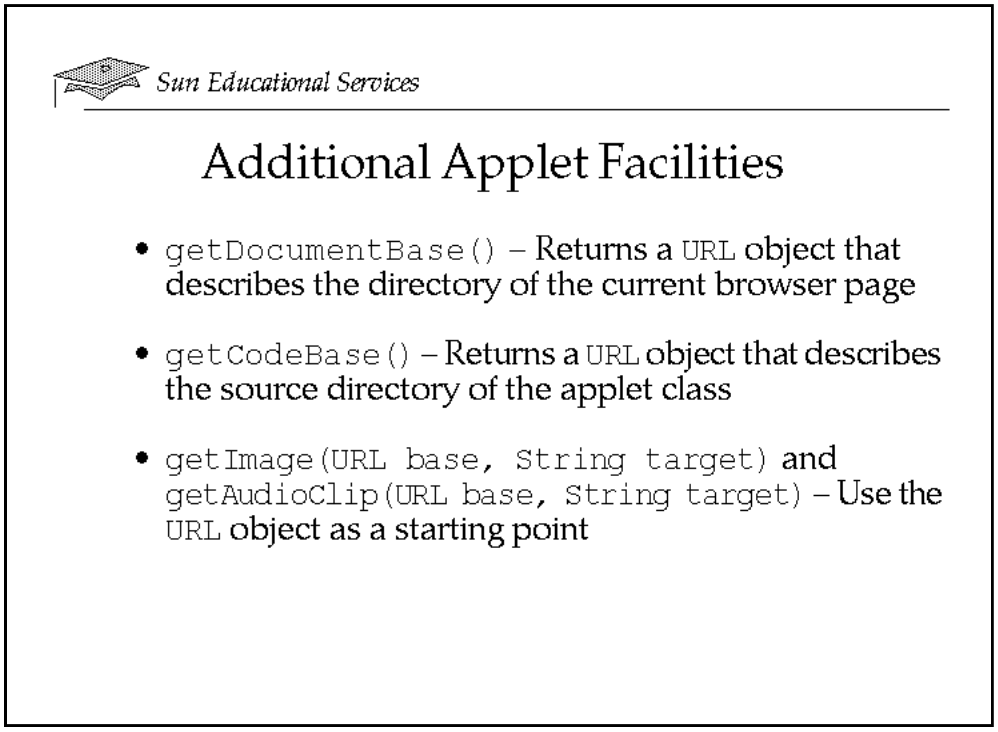
---

- width = *pixels* height = *pixels* – These *required* attributes give the initial width and height (in pixels) of the applet display area, not including any Windows or Dialogs that the applet brings up.

## *The* `applet` *Tag*

### *Description*

- `codebase =` *codebaseURL* – This optional attribute specifies the base URL of the applet—the directory that contains the applet's code. If this attribute is not specified, then the document's URL is used.
- `alt =` *alternateText* – This optional attribute specifies which text to display if the browser can read the applet tag but cannot run Java applets.
- `name =` *appletInstanceName* – This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.
- `align =` *alignment* – This optional attribute specifies the alignment of the applet. The possible values of this attribute are the same as those for the `IMG` tag in basic HTML: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, and `absbottom`.
- `vspace =` *pixels* `hspace =` *pixels* – These optional attributes specify the number of pixels above and below the applet (`vspace`) and on each side of the applet (`hspace`). They are treated the same way as the `IMG` tag's `vspace` and `hspace` attributes.
- <`param name` = *appletAttribute1* `value` = *value*> – This tag provides an applet with a value specified "from the outside," serving the same functional purpose as command-line arguments serve a Java application. Applets access their attributes with the `getParameter()` method, which is covered later in this module in more detail.
- Browsers that are not capable of running Java programs will display any regular HTML included between your `<applet>` and `</applet>` tags; browsers capable of supporting Java technology ignore the HTML code between these two tags.



Sun Educational Services

## Additional Applet Facilities

- getDocumentBase() – Returns a URL object that describes the directory of the current browser page
- getCodeBase() – Returns a URL object that describes the source directory of the applet class
- getImage(URL base, String target) and getAudioClip(URL base, String target) – Use the URL object as a starting point

## *Additional Applet Features*

A number of additional features are available in an applet.

All Java software programs have access to network features using the classes in the `java.net` package that is examined in Module 15. Applets additionally have methods that allow them to determine information about the browser environment in which they have been launched.

The class `java.net.URL` describes URLs and can be used to connect to them. Two methods in the `Applet` class determine the value of significant URLs:

- `getDocumentBase()` returns a URL object that describes the directory of the current browser page (where the HTML file with applet tags resides).
- `getCodeBase()` returns a URL object that describes the source directory of the applet class file itself. Often this is the same as the HTML file directory, but this is not necessarily the case.

## Additional Applet Features

Using the URL as a starting point, you can put sounds and images into your applet.

- `getImage(URL base, String target)` fetches an image from the file named by `target` located at the URL specified by `base`. The returned value is an instance of the class `Image`.
- `getAudioClip(URL base, String target)` fetches a sound from the file named by `target` located at the URL specified by `base`. The returned value is an instance of the class `AudioClip`.

---

**Note –** The `String` target in `getImage(URL, String)` and `getAudioClip(URL, String)`methods can include a relative directory path from the URL. Be cautioned, however, that relative pathnames up the directory hierarchy might not be allowed on some systems.

---

## A Simple Image Test

The following applet retrieves the image file `graphics/surferDuke.gif` relative to the directory path returned by the `getDocumentBase` method and displays it:

```
// Applet which shows an image of Duke in surfing mode

import java.awt.*;
import java.applet.Applet;

public class HwImage extends Applet {
  Image duke;

  public void init() {
    duke = getImage(getDocumentBase(),
        "graphics/surferDuke.gif");
  }

  public void paint(Graphics g) {
    g.drawImage(duke, 25, 25, this);
  }
}
```

The arguments to the `drawImage()` method are:

- The `Image` object to be drawn.
- The x coordinate for the drawing.
- The y coordinate for the drawing.
- The *image observer.* An image observer is an interface that is notified if the image's status changes (such as what happens during loading).

An image that is loaded by `getImage()` will change over time after the call is first issued. This is because the loading is done in the background. Each time more of the image is loaded, the `paint()` method is called again. This call to the `paint()` method happens because the applet was registered as an observer when it passed itself as the fourth argument to `drawImage()`.

## Audio Clips

The Java programming language also has methods to play audio clips. These methods are in the `java.applet.AudioClip` class. You will need the appropriate hardware for your computer to play audio clips.

### Playing a Clip

The easiest way to listen to an audio clip is through an `Applet` `play` method:

```
play(URL soundDirectory, String soundFile);
```

or, more simply:

```
play(URL soundURL);
```

For example,

```
play(getDocumentBase(), "bark.au");
```

will play `bark.au` which exists in the same directory as the HTML file.

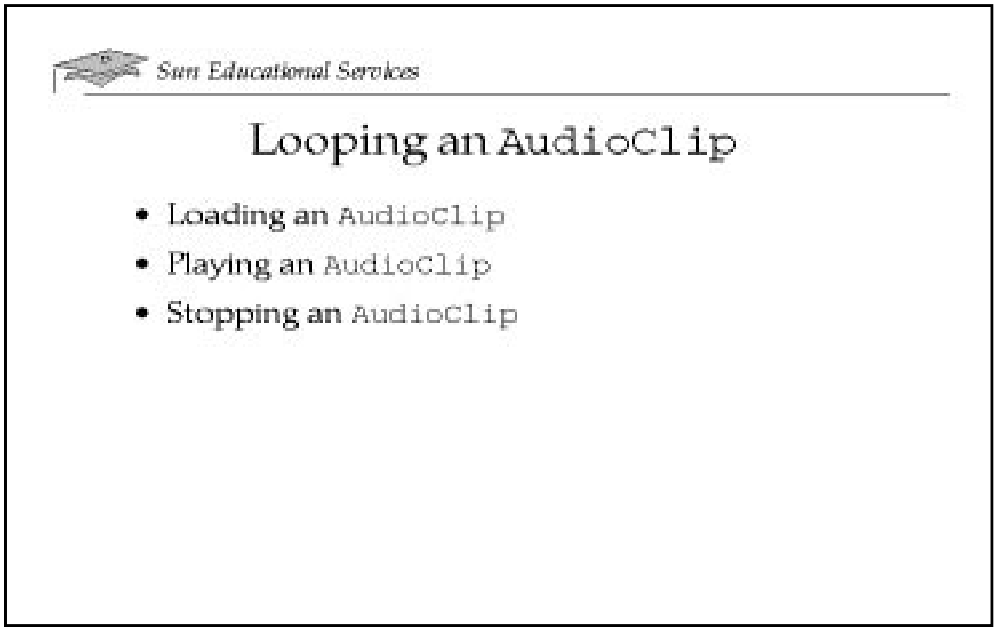
✓ ***In most instances, you cannot use a relative path name that goes up the path hierarchy; for example,*** `../bark.au`***.***

## A Simple Audio Test

The following applet prints the message *Audio Test* in the appletviewer and then plays the audio file, cuckoo.au in the sounds directory:

```
// Applet which plays a sound once

import java.awt.Graphics;
import java.applet.Applet;

public class HwAudio extends Applet {
  public void paint(Graphics g) {
    g.drawString("Audio Test", 25, 25);
    play(getCodeBase(), "sounds/cuckoo.au");
  }
}
```

✓ ***The applet*** `play()` ***method creates an*** `AudioClip` ***(named*** `clip`***) using the URL you give it and then it calls*** `clip.play()`***.***


Sun Educational Services

## Looping an AudioClip

- Loading an AudioClip
- Playing an AudioClip
- Stopping an AudioClip

## *Looping an Audio Clip*

You can load audio clips like images; that is, you can load them and play them later.

### *Loading an Audio Clip*

To load an audio clip, use the `getAudioClip()` method from the `java.applet.Applet` class.

```
AudioClip sound;
sound = getAudioClip(getDocumentBase(), "bark.au");
```

Once a clip is loaded, use one of three methods associated with it: `play()`, `loop()`, or `stop()`.

## *Looping an Audio Clip*

### *Playing an Audio Clip*

Use the `play()` method in the `java.applet.AudioClip` interface to play the loaded audio clip once.

```
sound.play();
```

To start the clip playing and have it loop (automatically repeat), use the `loop()` method in `java.applet.AudioClip`.

```
sound.loop();
```

### *Stopping an Audio Clip*

To stop a running clip, use the `stop` method in `java.applet.AudioClip`.

```
sound.stop();
```

## A Simple Audio Looping Test

The following example automatically loops through a loaded audio clip:

```
1  // Applet which continuously repeats a sound
2
3  import java.awt.Graphics;
4  import java.applet.*;
5
6  public class HwLoop extends Applet {
7    AudioClip sound;
8
9    public void init() {
10     sound = getAudioClip(getCodeBase(), "sounds/cuckoo.au");
11   }
12
13   public void paint(Graphics g) {
14     g.drawString("Audio Test", 25, 25);
15   }
16
17   public void start() {
18     sound.loop();
19   }
20
21   public void stop() {
22     sound.stop();
23   }
24 }
```

**Note –** JDK 1.2 supports a new sound engine that provides playback for Musical Instrument Digital Interface (MIDI) files and the full range of `.wav`, `aiff`, and `.au` files. It introduces a new method `newAudioClip(URL url)`. This method retrieves an audio clip from the given URL. The parameter URL points to the audio clip. The `getAudioClip()` method in line 13 can be replaced with this method. The `newAudioClip` method does not need a `String` as the second parameter. Only the `URL` parameter should be passed.

## Mouse Input

One of the most useful features the Java programming language supports is direct interactivity. A Java applet, like an application, can pay attention to the mouse and react to mouse events. Here is a quick review of mouse support, to aid in understanding the next example.

Recall from Module 9 that the JDK 1.2 event model supports an event type for each type of interactivity. Mouse events are received by classes that implement the `MouseListener` interface, which receives events for:

- `mouseClicked` – The mouse has been clicked (mouse button pressed and then released in one motion).
- `mouseEntered` – The mouse cursor enters a component.
- `mouseExited` – The mouse cursor leaves a component.
- `mousePressed` – The mouse button is pressed down.
- `mouseReleased` – The mouse button is later released.

✓ ***Note that presses and releases are always received, but clicks are the result of a single "click" and might not be received if the user holds the mouse button down.***

## A Simple Mouse Test

The following program displays the location of the mouse click within the applet:

```
// This applet is HelloWorld extended to watch for mouse
// input. "Hello World!" is reprinted at the location of
// the mouse press.

import java.awt.Graphics;
import java.awt.event.*;
import java.applet.Applet;

public class HwMouse
    extends Applet
    implements MouseListener {

  private int mouseX = 25;
  private int mouseY = 25;

  // Register this applet instance to catch
  // MouseListener events.
  public void init() {
    addMouseListener(this);
  }

  public void paint(Graphics g) {
    g.drawString("Hello World!", mouseX, mouseY);
  }

  // Process the mousePressed MouseListener event
  public void mousePressed(MouseEvent evt) {
    mouseX = evt.getX();
    mouseY = evt.getY();
    repaint();
  }

  // We are not using the other mouse events.
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }
  public void mouseReleased(MouseEvent e) { }
}
```

## Reading Parameters

In an HTML file, a <param> tag in an <applet> context can pass configuration information to the applet. For example:

```
<html>
<applet code=DrawAny.class width=200 height=200>
<param name=image value="graphics/duke.gif">
</applet>
</html>
```

Inside the applet, you can use the method getParameter() to read these values.

```
import java.awt.*;
import java.applet.*;

public class DrawAny extends Applet {
  Image im;

  public void init() {
    String imageName = getParameter("image");

    if ( imageName == null ) {
      System.out.println(
          "Error: Cannot find image");
      System.exit(0);
    }

    im = getImage(getDocumentBase(), imageName);
  }

  public void paint(Graphics g) {
    g.drawImage(im, 0, 0, this);
  }
}
```

## Reading Parameters

The method `getParameter()` searches for a match of the name, and returns the associated value as a String.

If the parameter name cannot be found in any `<param>` tag inside the `<applet></applet>` pair, then `getParameter()` returns `null`. A production program should handle this gracefully.

The parameter type is always a `String`. If you need this in other forms you must convert it; for example, read an `int` parameter.

```
int speed = Integer.parseInt (getParameter (″speed″));
```

Parameter names, because of the nature of HTML, are not case sensitive; however, it is good style to make them entirely upper- or lowercase. Enclose parameter value strings in double quotes if they include embedded spaces. Value strings are case sensitive; their capitialization is maintained whether or not double quotes are used.

## Dual Purpose Code

It is possible to create Java software code that can be used as both a Java applet and a Java application from a single class file. There is a little more work involved in understanding what the application requires, but once created, the applet/application code can be used as a template for more complicated programs.

### Applet/Application

```
// Applet/Application which shows an image of
// Duke in surfing mode

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class AppletApp extends Applet {
  Date date;

  public void init() {
    date = new Date();
  }

  public void paint (Graphics g) {
    g.drawString("This Java program started at", 25, 25);
    g.drawString(date.toString(), 25, 60);
  }

  // An application will require a main()
  public static void main (String args[]) {

    // Create a Frame to house the applet
    Frame frame = new Frame("Application");

    // Create an instance of the class (applet)
    AppletApp app = new AppletApp();

    // Add it to the center of the frame
    frame.add(app, BorderLayout.CENTER);
    frame.setSize (250, 150);

```

## Dual Purpose Code

### *Applet/Application(Continued)*

```
  // Register the AppletApp class as the
  // listener for a Window Destroy event
  frame.addWindowListener (new WindowAdapter() {
    public void windowClosing (WindowEvent e) {
      System.exit(0);
    }
  });

  // Call the applet methods
  app.init();
  app.start();
  frame.setVisible(true);  // Invokes paint()
 }
}
```

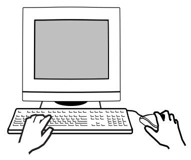
✓ ***Frames do not quit or shut down automatically, you must create your own routine. The*** `WindowClosing` ***event is used here because*** `WindowClosed` ***never seems to arrive.***

---

**Note –** Applications do not have the resources provided by a browser and therefore are unable to use `getImage()` or `getAudioClip()`.

---

## Exercise: Creating Applets



**Exercise objective –** In this lab you will become familiar with applet programming in particular the `paint()` method used for screen update and refresh.

### Preparation

In order to successfully complete this lab, you must be able to display an applet with a browser.

### Tasks

#### *Level 1 Lab: Write an Applet*

Complete the following steps:

1. Open a new Shell or Command Tool window.
2. Using a text editor, type the `HwMouse.java` program or copy it from the `course_examples` directory.
3. Modify the program so that it cycles between three different messages as you click around in the applet.
4. Compile the `HwMouse.java.java` program.

   ```
   javac HwMouse.java
   ```

5. Using a text editor, create a `HwMouse.html` file with an `<applet>` tag to call the `HwMouse.class` program.
6. Test your applet with the `appletviewer` command.

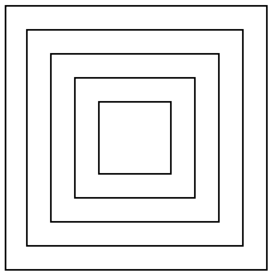
   ```
   appletviewer HwMouse.html
   ```

## Exercise: Creating Applets

### Level 2 Lab: Create Concentric Squares

Perform the following steps:

1. Create an applet, `Squares.java`, that produces a series of concentric squares (or circles) such as this:



2. Try making each square (or circle) a different color. If you import the `java.awt.Color` class, you can add color to Java applets using the `setColor` method.

```
import java.awt.Color;
. . .
public void paint(Graphics g) {
  g.setColor(Color.blue);
  g.drawRect(5, 5, 50, 50);
. . .
}
```

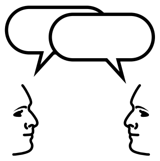
### Level 3 Lab: Create a Java Rollover Applet

Complete the following task:

1. Write an applet that displays an image and plays a sound when the mouse passes over the image.

## *Exercise: Creating Applets*

### *Exercise Summary*



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

❑ Differentiate between a stand-alone application and an applet

❑ Write an HTML tag to call a Java applet

❑ Describe the class hierarchy of the applet and AWT classes

❑ Create the `HelloWorld.Java` applet

❑ List the major methods of an applet

❑ Describe and use the painting model of AWT

❑ Use applet methods to read images and files from URLs

❑ Use `<param>` tags to configure applets

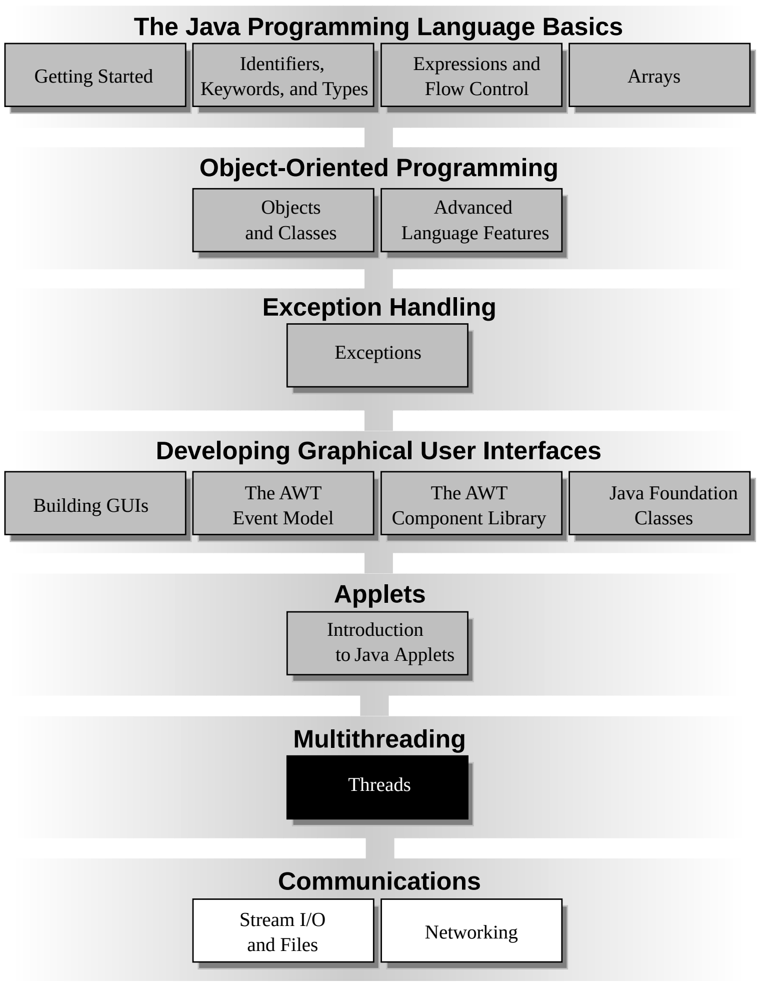
## Think Beyond

How can you use applets on your company's Web page to improve the overall presentation?

# Threads 13

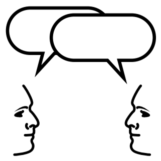
## Course Map

This module covers multithreading, which allows a program to do multiple tasks at the same time.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

- How do I get my programs to perform multiple tasks?

✓ ***There are times when a program needs to do multiple tasks at the same time. For example, a Web browser displaying a sports web page runs a Java applet spinning a football in the corner, but leaves the scrollbar active so the Web page can be traversed at the same time. The Web browser is multithreaded, allowing it this flexibility; without multithreading, the browser would be totally preoccupied with spinning the football and the user would never be able to scroll the Web page. This module discusses how to set up multiple threads in a single program, so that multiple tasks can be done at the same time.***

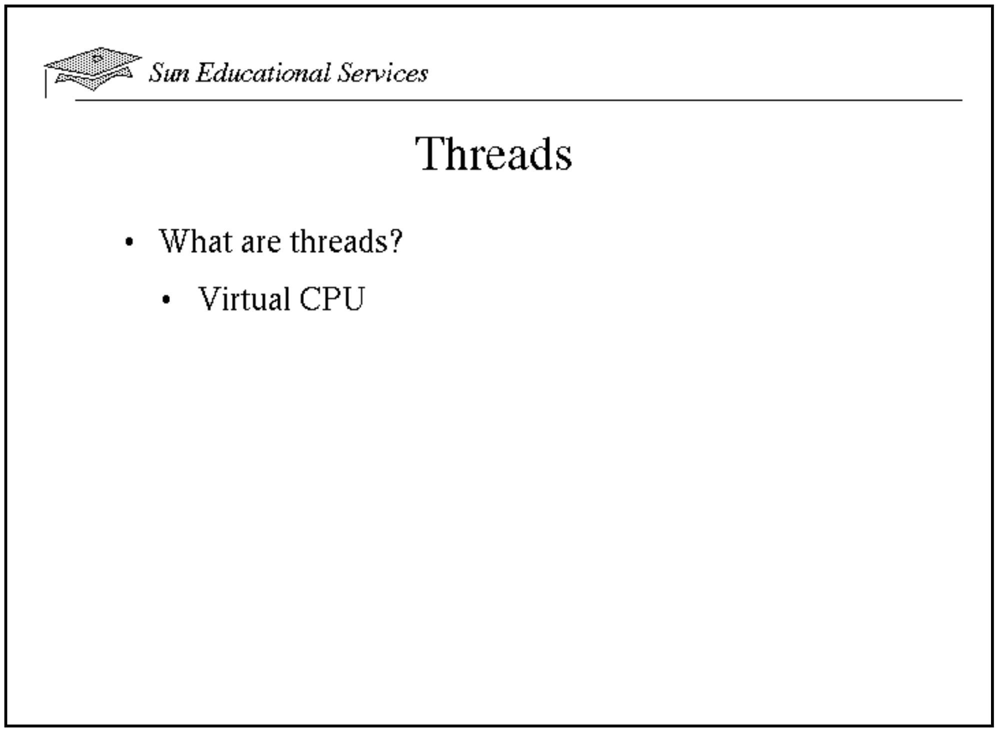
✓ ***Multithreading is useful even in programs running on a single-processor computer. In this scenario, thread two will run while thread one is blocked waiting for I/O. The I/O can take place in parallel with the CPU running the other thread. In a single-threaded case, thread two would have to wait until thread one finished.***

✓ ***Data corruption can result if two independent threads access the same data at the same time. Threads running independently yet sharing common data need to be coordinated. Furthermore, the timing of when one thread runs might depend on the other thread, as in a producer/consumer problem. This module discusses the ramifications of sharing data between threads, and discusses how to coordinate multiple threads sharing the same data.***

## Objectives

Upon completion of this module, you should be able to

- Define a thread
- Create separate threads in a Java software program, controlling the code and data that are used by that thread
- Control the execution of a thread and write platform-independent code with threads
- Describe the difficulties that might arise when multiple threads share data
- Use the keyword `synchronized` to protect data from corruption
- Use `wait()` and `notify()` to communicate beween threads
- Explain why `suspend()`, `resume()`, and `stop()` methods have been deprecated in JDK 1.2



Sun Educational Services

## Threads

- What are threads?
  - Virtual CPU

## Threads

### What Are Threads?

A simplistic view of a computer is that it has a CPU which performs computation, ROM (read-only memory) which contains the program that the CPU executes, and RAM (random-access memory) which holds the data on which the program operates. In this simple view, there is only one job being performed. A more complete view of most modern computer systems allows the possibility of their performing more than one job at the same time.

You do not need to be concerned with how this is achieved, just consider the implications from a programming point of view. If you have more than one job being performed, this is similar to having more than one computer. In this module, a *thread*, or *execution context*, is considered to be the encapsulation of a *virtual CPU* with its own program code and data. The class `java.lang.Thread` allows users to create and control their threads.

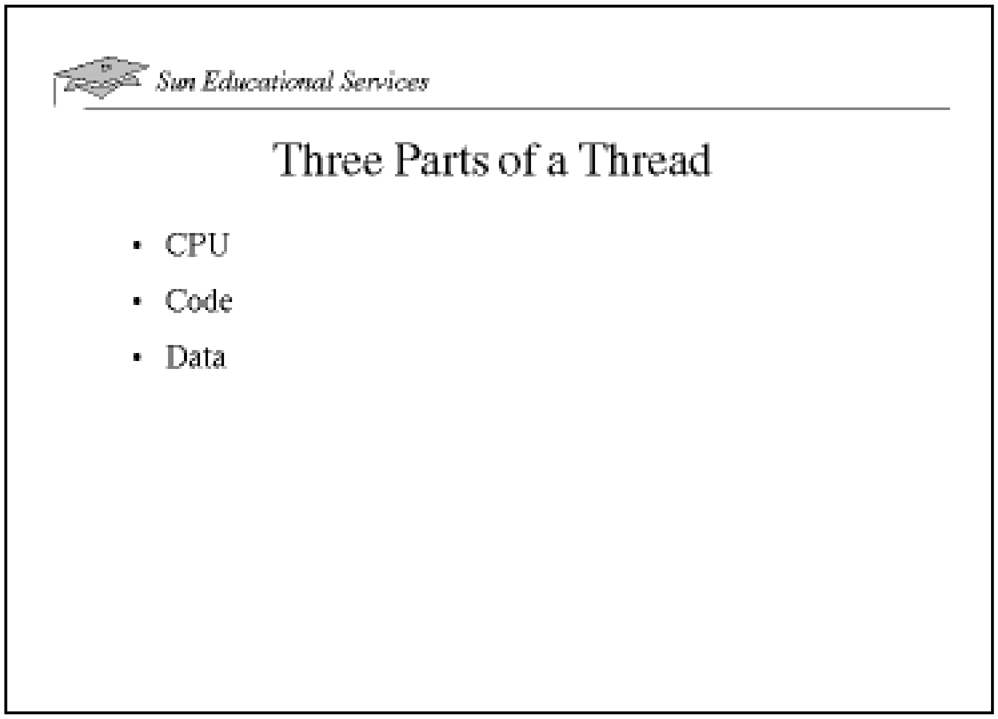
## Threads

### What Are Threads? (Continued)

---

**Note –** This module uses the term *Thread* when referring to the class `java.lang.Thread` and *thread* when referring to an execution context.

---



Sun Educational Services

## Three Parts of a Thread

- CPU
- Code
- Data

## Threads in Java Programming

### Three Parts of a Thread

Process is a program in execution. One or more threads constitute a process.

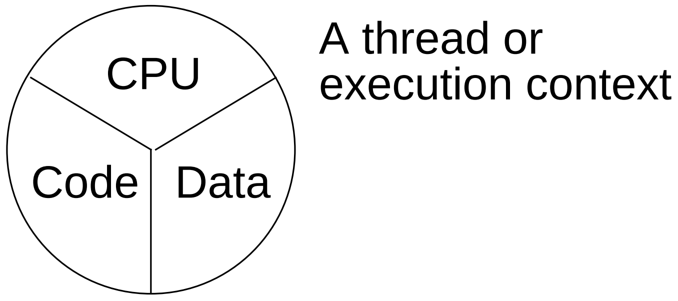


**Figure 13-1** A Thread

## Threads in Java Programming

### Three Parts of a Thread (Continued)

A thread or *execution context* is comprised of three main parts:

- A virtual CPU
- The code the CPU is executing
- The data the code works on

Code may or may not be shared by multiple threads, independent of data. Two threads share the same code when they execute code from instances of the same class.

Likewise, data may or may not be shared by multiple threads, independent of code. Two threads share the same data when they share access to a common object.

In Java programming, the virtual CPU is encapsulated in an instance of the `Thread` class. When a thread is constructed, the code and the data which define its context are specified by the object passed to its constructor.

## Creating the Thread

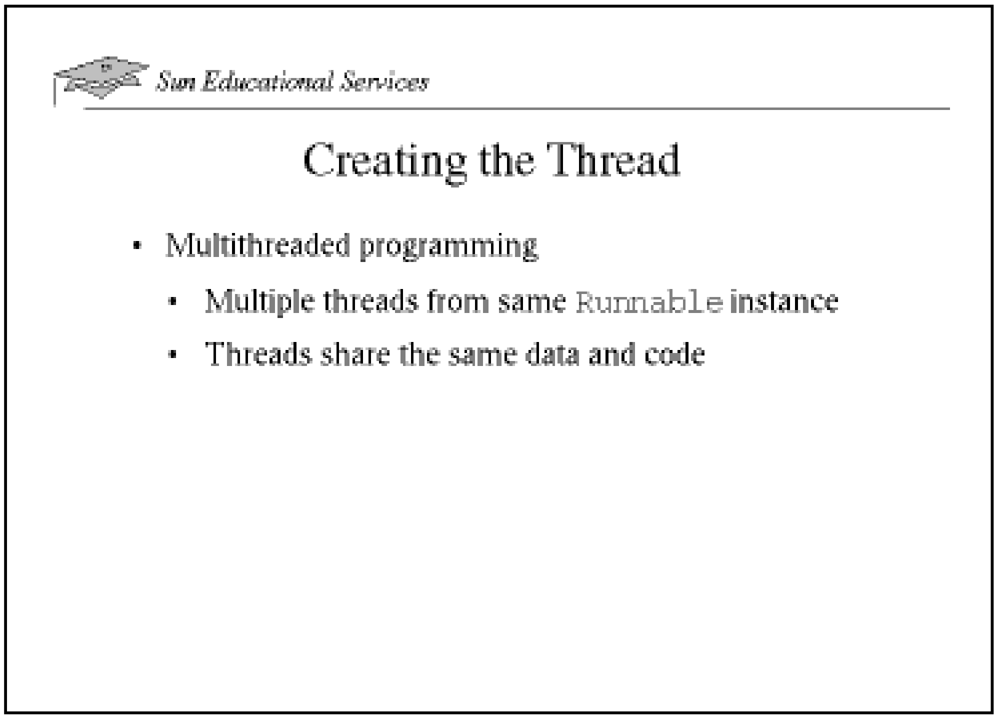
This section examines how a thread is created, and how constructor arguments are used to supply the code and data for a thread when it runs.

A `Thread` constructor takes an argument which is an *instance* of `Runnable`. An instance of `Runnable` is made from a class which implements the `Runnable` interface (that is, provide a `public void run()` method).

For example:

```
public class ThreadTest {
  public static void main(String args[]) {
    Xyz r = new Xyz();
    Thread t = new Thread(r);
    t.start();
  }
}

class Xyz implements Runnable {
  int i;

  public void run() {
    i = 0;

    while (true) {
      System.out.println("Hello " + i++);
      if ( i == 50 ) {
        break;
      }
    }
  }
}
```

First, the `main()` method constructs an instance `r` of *class* `Xyz`. Instance `r` has its own data, in this case the integer `i`. Because the instance, `r`, is passed to the `Thread` class constructor, `r`'s integer `i` is the data with which the thread will work when it runs. The thread will always begin executing at the `run()` method of its loaded `Runnable` instance (`r` in this example.).


Sun Educational Services

## Creating the Thread

- Multithreaded programming
  - Multiple threads from same `Runnable` instance
  - Threads share the same data and code

## *Threads in Java Programming*

### *Creating the Thread*

A multithreaded programming environment allows creation of multiple threads based on the same `Runnable` instance. This could be achieved as follows:

```
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
```

In this case, both threads share the same data and code.

## Threads in Java Programming

### Creating the Thread

To summarize, a thread is referred to through an instance of a `Thread` object. The thread begins execution at the start of a loaded `Runnable` instance's `run()` method. The data that the thread works on is taken from the *specific* instance of `Runnable` which is passed to that `Thread` constructor.

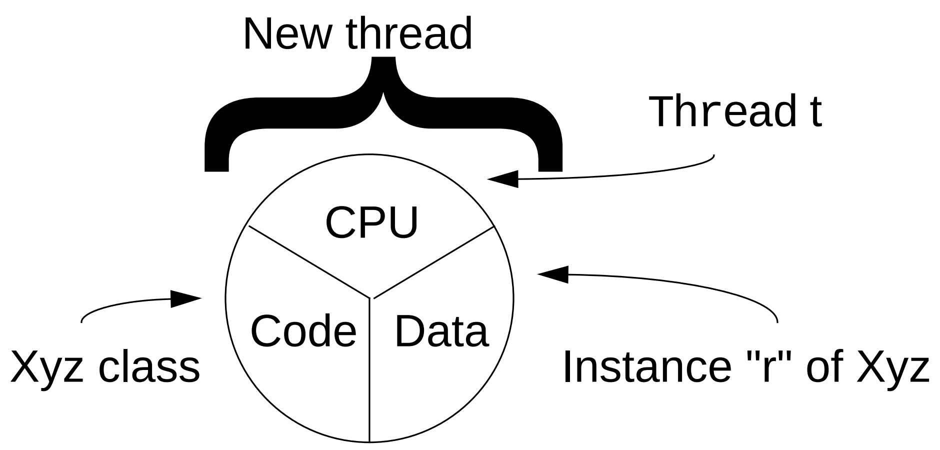
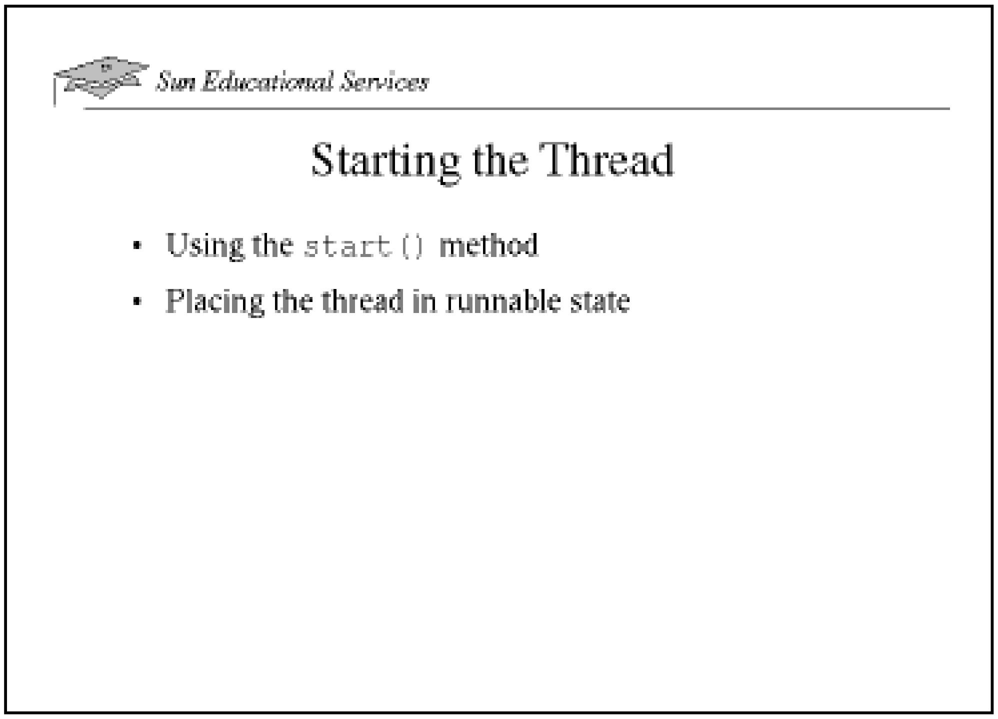


**Figure 13-2** Thread Creation



Sun Educational Services

## Starting the Thread

- Using the `start()` method
- Placing the thread in runnable state

## *Threads in Java Programming*

### *Starting the Thread*

A newly created thread does not start running automatically. You must call its `start()` method. For example, you can issue the following command after line 4 of the previous example:

```
t.start();
```

Calling `start()` places the virtual CPU embodied in the thread into a runnable state, meaning that it becomes viable for scheduling for execution by the JVM. This does not necessarily mean that the thread will run immediately.

## Thread Scheduling

A `Thread` object can exist in many different states throughout its lifetime. The following diagram illustrates this idea:

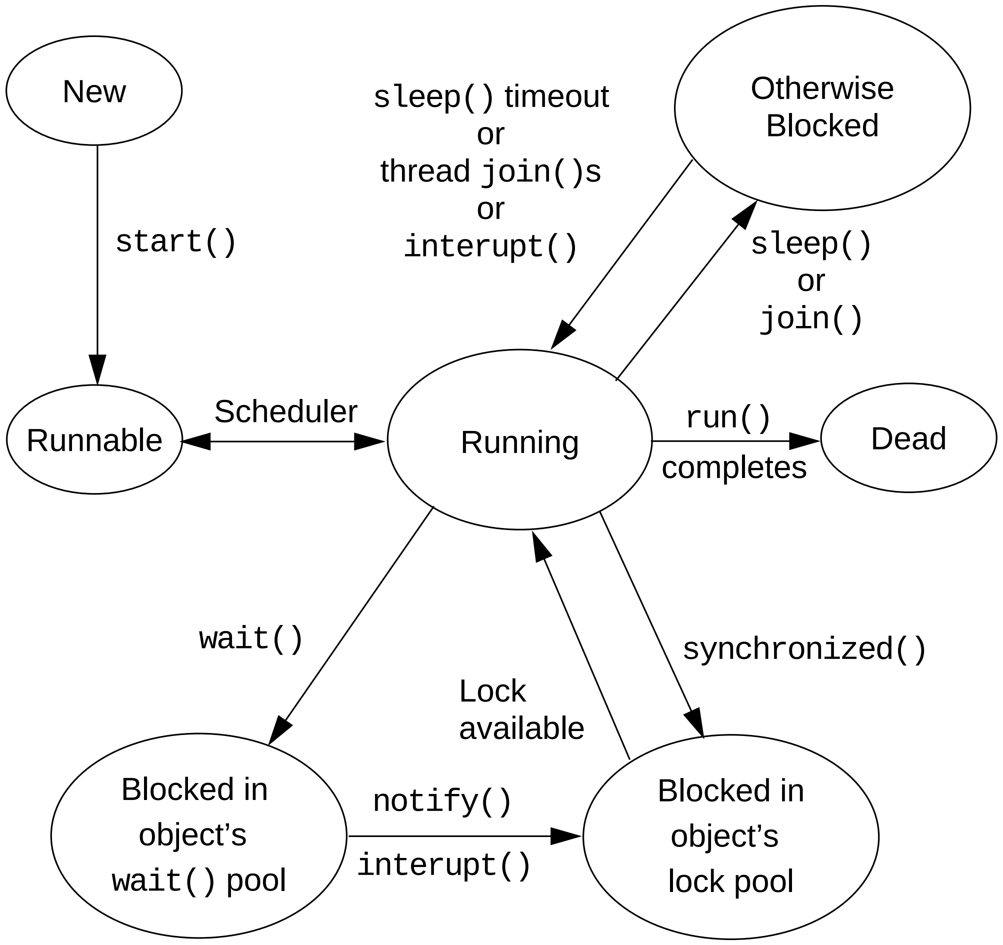


**Figure 13-3** Thread States

✓ ***With JDK1.2, the*** `suspend()`, `resume()`, ***and*** `stop()` ***methods have been deprecated.*** `suspend()` ***is deadlock prone and*** `stop()` ***is unsafe in terms of date protection.***

Although the thread becomes runnable, it does not necessarily start running immediately. Only one action at a time can be done on a machine that has only one CPU. The following section describes how the CPU is allocated when more than one thread is runnable.

## Threads in Java Programming

### Thread Scheduling (Continued)

In Java technology, threads are usually *preemptive*, but not necessarily timesliced. (It is a common mistake to believe that "preemptive" is a fancy word for "does timeslicing.")

✓ ***For the runtime on a Solaris platform, Java technology will not preempt threads of the same priority. However, the runtime on Windows 95 and NT platforms uses timeslicing, so it will preempt threads of the same priority and even threads of higher priority. Preemption is not guaranteed, however, most JVM implementations result in behavior that appears to be strictly preemptive. Across JVM implementations, there is no absolute guarantee of preemption or timeslicing. The only guarantees lie in the coder's use of*** `wait()` ***and*** `sleep()`.

The model of a preemptive scheduler is that many threads might be runnable, but only one thread is actually running. This thread will continue to run until either it ceases to be runnable, or another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *preempted* by the thread of higher priority which gets a chance to run instead.

A thread might cease to be runnable for a variety of reasons. The thread's code can execute a `Thread.sleep()` call, deliberately asking the thread to pause for a fixed period of time. The thread might have to wait to access a resource, and cannot continue until that resource becomes available.

All threads which are runnable are kept in pools according to priority. When a blocked thread becomes runnable, it is placed back into the appropriate runnable pool. Threads from the highest priority non-empty pool will be given CPU time.

✓ ***The last sentence is worded loosely because: (1) in most JVM implementations, priorities seem to work in a preemptive manner, although there is no guarantee that priorities have any meaning at all; and (2) Window's nice values affect thread behavior so that it is possible that a Java priority 4 thread might be running, in spite of the fact that a runnable Java priority 5 thread is waiting for the CPU.***

✓ ***In reality, many JVMs implement pools as queues, but this is not guaranteed behavior.***

## Threads in Java Programming

Given that Java threads are not necessarily timesliced (the process of giving each thread an equal amount of CPU time), you must insure that the code for your threads gives other threads a chance to execute from time to time. This can be achieved by issuing the `sleep()` call at various intervals.

```
public class Xyz implements Runnable {
  public void run() {
    while (true) {
      // do lots of interesting stuff
       :
      // Give other threads a chance
      try {
        Thread.sleep(10);
      } catch (InterruptedException e) {
        // This thread's sleep was interrupted
        // by another thread
      }
    }
  }
}
```

✓ ***Note that in JDK 1.1 it is possible to interrupt a thread with a*** `Thread.interrupt()` ***method.***

Notice the use of the `try` and `catch` block. `Thread.sleep()` and other methods which can pause a thread for periods of time are interruptable. Threads can call another thread's `interrupt()` method, which signals the paused thread with an `InterruptedException`.

Note that `sleep()`is a `static` method in the `Thread` class because it operates on the current thread, and hence is referred to as `Thread.sleep(x)`. The `sleep()` method's argument specifies the minimum number of milliseconds for which the thread must be made inactive. The execution of the thread will not resume until after this period unless it is interrupted, in which case execution is resumed earlier.

## Threads in Java Programming

### Thread Scheduling (Continued)

Another method in the `Thread` class, `yield()`, can be issued to give other threads of the same priority a chance to execute. If other threads at the same priority are runnable, `yield()` places the calling thread into the runnable pool and allows another thread to run. If no other threads are runnable at the same priority, `yield()` does nothing.

Notice that a `sleep()` call gives threads of lower priority a chance to execute. The `yield()` method gives only threads of the same priority a chance to execute.

## Basic Control of Threads

### Terminating a Thread

When a thread completes execution and terminates, it *cannot* run again.

A thread can be stopped by using a flag which indicates that the run() method should exit.

```
public class Xyz implements Runnable {
  private boolean timeToQuit=false;

  public void run() {
    while(! timeToQuit) {
           ...
    }
    // clean up before run() ends
  }

  public void stopRunning() {
    timeToQuit=true;
  }
}

public class ControlThread {
  private Runnable r = new Xyz();
  private Thread t = new Thread(r);

  public void startThread() {
    t.start();
  }

  public void stopThread() {
    // use specific instance of Xyz
    r.stopRunning();
  }
}
```

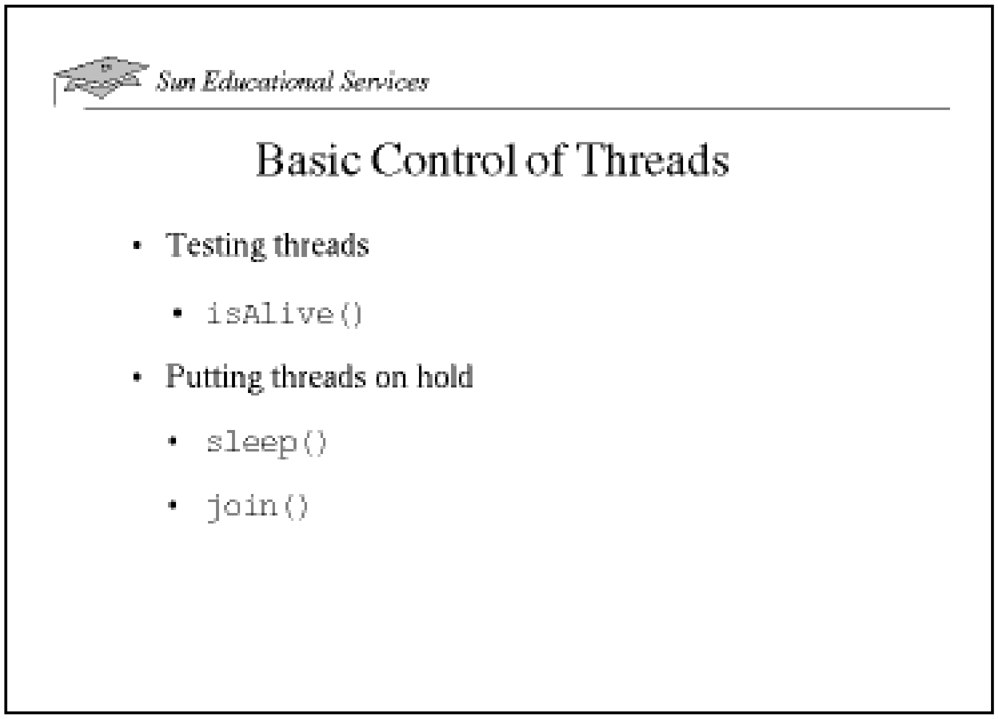
✓ ***The method*** `stop()` ***of the*** `Thread` ***class has been deprecated in JDK 1.2.***

## Basic Control of Threads

### Terminating a Thread (Continued)

Within a particular piece of code, it is possible to obtain a reference to the current thread using the static Thread method currentThread(); for example,

```
public class Xyz implements Runnable {
  public void run() {
    while (true) {
      // lots of interesting stuff
      // Print name of the current thread
      System.out.println(
          "Thread" + Thread.currentThread().getName()+ "completed");
    }
  }
}
```


Sun Educational Services

## Basic Control of Threads

- Testing threads
  - `isAlive()`
- Putting threads on hold
  - `sleep()`
  - `join()`

## *Basic Control of Threads*

### *Testing a Thread*

It is sometimes possible for a thread to be in an unknown state. The method `isAlive()` is used to determine if a thread is still viable. Alive does not imply that the thread is running; it returns `true` for a thread that has been started but has not completed its task.

### *Putting Threads on Hold*

Mechanisms which are able to block the execution of a thread temporarily exist. Execution can be resumed as if nothing had happened, the thread appears to have executed an instruction very slowly.

## Basic Control of Threads

### Putting Threads on Hold (Continued)

#### sleep()

The `sleep()` method is one way to halt a thread for a period of time. Recall that the thread will not necessarily resume its execution at the instant that the sleep period expires. This is because some other thread is likely to be executing at that instant and might not be unscheduled unless (a) the thread "waking up" is of a higher priority, or (b) the running thread blocks for some other reason.

```
public class Xyz implements Runnable {
  ...
  public void run() {
    while (running) {
      // do your task
      try {
        Thread.sleep((int)(Math.random() * 100));
      } catch (InterruptedException e) {
        // somebody woke me up
      }
      ...
    }
  }
}

public class TTest {
  public static void main(String args[]) {
    Runnable r = new Xyz();
    Thread t1 = new Thread(r);
    t1.start();
  }
}
```

✓ ***The methods `suspend()` and `resume()` of the `Thread` class have been deprecated in JDK 1.2. These methods, plus the `stop()` method, have been deprecated for essentially the same reason: they do not work well within the Java technology's model of object synchronization.***

## Basic Control of Threads

### Putting Threads on Hold

#### join()

The `join()` method causes the current thread to wait until the thread on which the `join` method is called terminates. For example:

```
public void doTask() {
  TimerThread tt = new TimerThread (100);
  tt.start ();
  ...
  // Do stuff in parallel with the other thread for
  // a while
  ...
  // Wait here for the timer thread to finish
  try {
    tt.join ();
  } catch (InterruptedException e) {
    // tt came back early
  }
  ...
  // Now continue in this thread
  ...
}
```

The `join` method can also be called with a timeout value in milliseconds. For example:

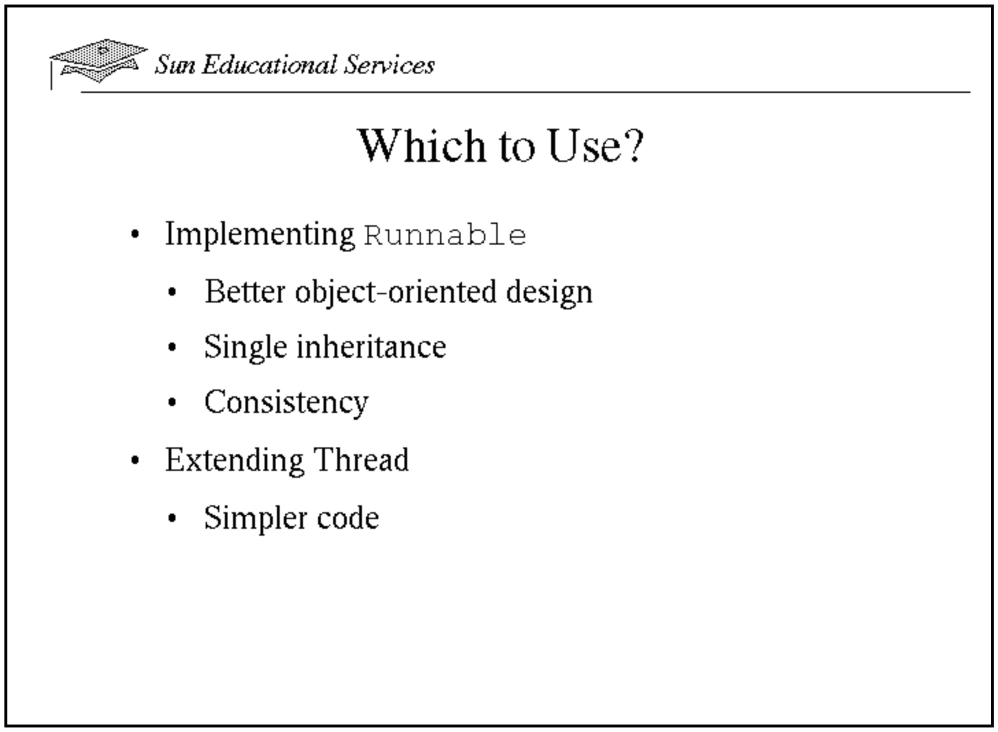
```
void join (long timeout);
```

where the `join` method will either suspend the current thread for *timeout* milliseconds or until the thread it calls on terminates.

## Other Ways to Create Threads

So far, you have seen how thread contexts can be created with a separate class which implements `Runnable`. In fact, this is not the only possible approach. The `Thread` class implements the `Runnable` interface itself, so it is possible to create a thread by creating a class which extends `Thread` rather than implements `Runnable`.

```
public class MyThread extends Thread {
  public void run() {
    while (running) {
      // do lots of interesting stuff
      try {
        sleep(100);
      } catch (InterruptedException e) {
        // sleep interrupted
      }
    }
  }

  public static void main(String args[]) {
    Thread t = new MyThread();
    t.start();
  }
}
```


Sun Educational Services

## Which to Use?

- Implementing `Runnable`
  - Better object-oriented design
  - Single inheritance
  - Consistency
- Extending Thread
  - Simpler code

## *Other Ways to Create Threads*

### *Which to Use?*

Given a choice of approaches, how can you decide between them? There are points in favor of each approach.

## Other Ways to Create Threads

### Which to Use? (Continued)

#### *In Favor of Implementing* `Runnable`

Some points to consider are:

- From an object-oriented design point of view, the `Thread` class is strictly an encapsulation of a virtual CPU, and as such, it should only be extended when the behavior of that CPU model is to be changed or extended. Because of this, and the value of making the distinction between the CPU, code, and data parts of a running thread, this course module has used this approach.
- Since Java technology only allows single inheritance, it is not possible to extend any other class, such as `Applet`, if you have already extended `Thread`. In some situations, this will force you to take the approach of implementing `Runnable`.

## Other Ways to Create Threads

### Which to Use?

#### In Favor of Implementing Runnable (Continued)

- Because there are times when you are obliged to implement `Runnable`, you might prefer to be consistent and always do it this way.

#### In Favor of Extending Thread

Be sure to consider that

- When a `run()` method is embodied in a class which extends the `Thread` class, `this` refers to the actual `Thread` instance which is controlling execution. Therefore, the code no longer needs to use longhand controls like

  ```
  Thread.currentThread().join();
  ```

  but can simply say

  ```
  join();
  ```

Because the resulting code is slightly simpler, many programmers of the Java programming language use the mechanism of extending threads. Be aware, however, that the single inheritance model might cause difficulties later in your code's life cycle if you adopt this approach.

## *Using* `synchronized` *in Java Technology*

✓ ***Use*** `synchronized` ***in your code to prevent data corruption.***

This section covers the use of the `synchronized` keyword. It provides the Java programming language with a mechanism which allows a programmer to control threads which are sharing data.

### *The Problem*

Imagine a class which represents a stack. This class might appear first as:

```
public class MyStack {
  int idx = 0;
  char [] data = new char[6];

  public void push(char c) {
    data[idx] = c;
    idx++;
  }

  public char pop() {
    idx--;
    return data[idx];
  }
}
```

Notice that the class makes no effort to handle overflow or underflow of the stack, and that the stack capacity is quite limited. These aspects are not relevant to this discussion.

The behavior of this model requires that the index value contain the array subscript of the next *empty* cell in the stack. The "predecrement, postincrement” approach is used to generate this information.

## *Using* `synchronized` *in Java Technology*

### *The Problem (Continued)*

Imagine now that *two* threads have a reference to a *single* instance of this class. One thread is pushing data onto the stack and the other, more or less independently, is popping data off of the stack. In principle, it would appear that the data will be added and removed successfully. However, there is a potential problem.

Suppose thread *a* is adding and thread *b* removing characters. Thread *a* has just deposited a character, but has not yet incremented the index counter. For some reason this thread is now preempted. At this point, the data model represented in the object is inconsistent.

```
buffer |p|q|r| | | |
idx = 2     ^
```

Specifically, consistency would require either `idx` = 3 or that the character has not yet been added.

If thread *a* resumes execution, there might be no damage, but suppose thread *b* was waiting to remove a character. While thread *a* is waiting for another chance to run, thread *b* gets its chance to remove a character.

There is an inconsistent data situation on entry to the `pop()` method, yet the `pop` method proceeds to decrement the index value.

```
buffer |p|q|r| | | |
idx = 1   ^
```

This effectively serves to ignore the character *r*. After this, it then returns the character *q*. So far, the behavior has been as if the letter *r* had not been pushed, so it is difficult to say that there is a problem. But look at what happens when the original thread, *a*, continues to run.

Thread *a* picks up where it left off, in the `push()` method, and it proceeds to increment the index value. Now you have

```
buffer |p|q|r| | | |
idx = 2     ^
```

## Using `synchronized` in Java Technology

### The Problem (Continued)

Notice that this configuration implies the *q* is valid and the cell containing *r* is the next empty cell. In other words, *q* will be read as having been placed into the stack twice, and the letter *r* will never appear.

This is a simple example of a general problem that can arise when *multiple* threads are accessing *shared* data. A mechanism is needed to insure that shared data is in a consistent state before any thread starts to use it for a particular task.

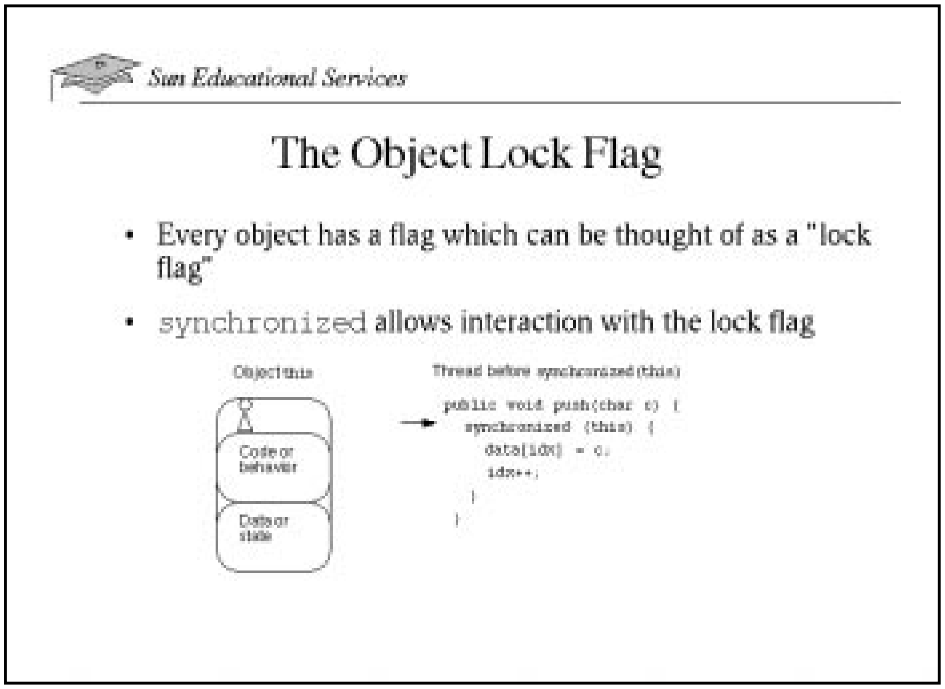
---

**Note –** One approach would be to prevent thread *a* from being switched out until it had completed the critical section of code. This approach is common in low-level machine programming but is generally inappropriate in multi-user systems.

---

**Note –** Another approach, and the one on which Java technology works, is to provide a mechanism to treat the data *delicately*. This approach allows a thread atomic access to data whether or not that thread gets switched out in the middle of performing that access.

---



Sun Educational Services

### The Object Lock Flag

- Every object has a flag which can be thought of as a "lock flag"
- `synchronized` allows interaction with the lock flag

Object this
Code or behavior
Data or state
Thread before synchronized(this)

```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

## *Using* `synchronized` *in Java Technology*

### *The Object Lock Flag*

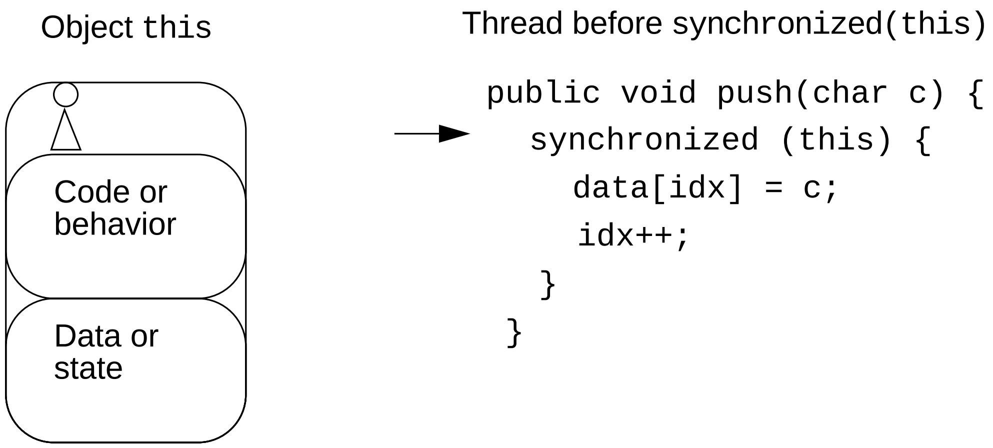
In Java technology, every object has a flag associated with it. This flag can be thought of as a "lock flag." The keyword `synchronized` enables interaction with this flag, and allows exclusive access to code that affects shared data. Look at the modified code fragment:

```
public void push(char c) {
  synchronized(this) {
    data[idx] = c;
    idx++;
  }
}
```

When the thread reaches the `synchronized` statement, it examines the object passed as the argument, and tries to obtain the lock flag from that object before continuing. (See Figure 13-4.)

## *Using* `synchronized` *in Java Technology*

### *The Object Lock Flag*

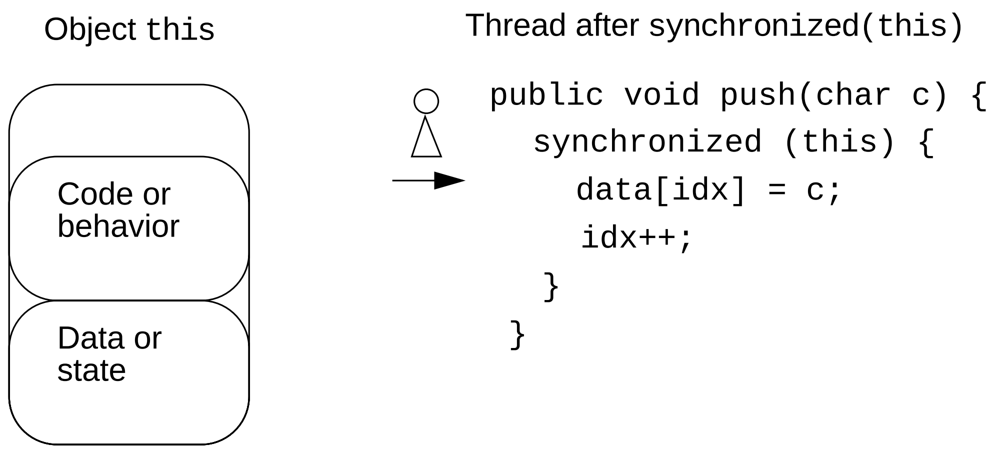


```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

**Figure 13-4** Using the `synchronized` Statement Before a Thread

## *Using* `synchronized` *in Java Technology*

### *The Object Lock Flag (Continued)*



```
public void push(char c) {
  synchronized (this) {
    data[idx] = c;
    idx++;
  }
}
```

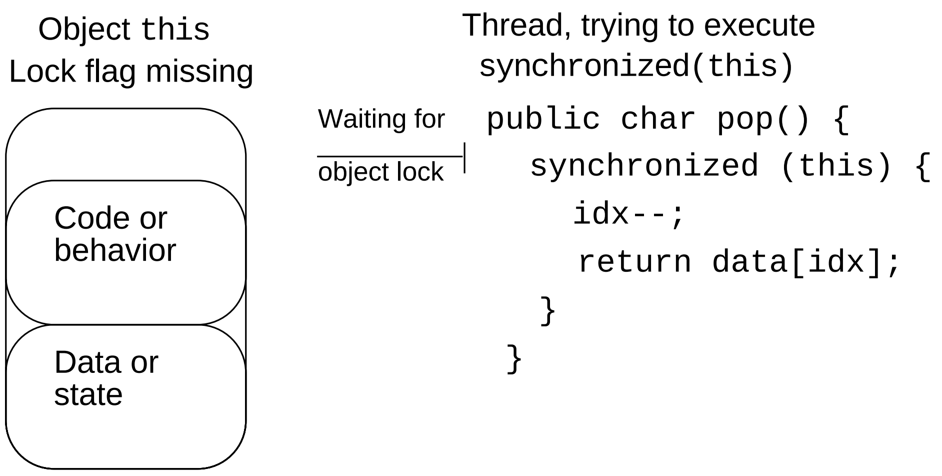
**Figure 13-5** Using the `synchronized` Statement After a Thread

It is important to realize that this has not, of itself, protected the data. If the `pop()` method of the shared data object is not protected by `synchronized`, and `pop()` is invoked by another thread, *there is still a risk of damaging the consistency of the* `data`. All methods accessing shared data must synchronize on the same lock if the lock is to be effective.

## *Using* `synchronized` *in Java Technology*

### *The Object Lock Flag (Continued)*

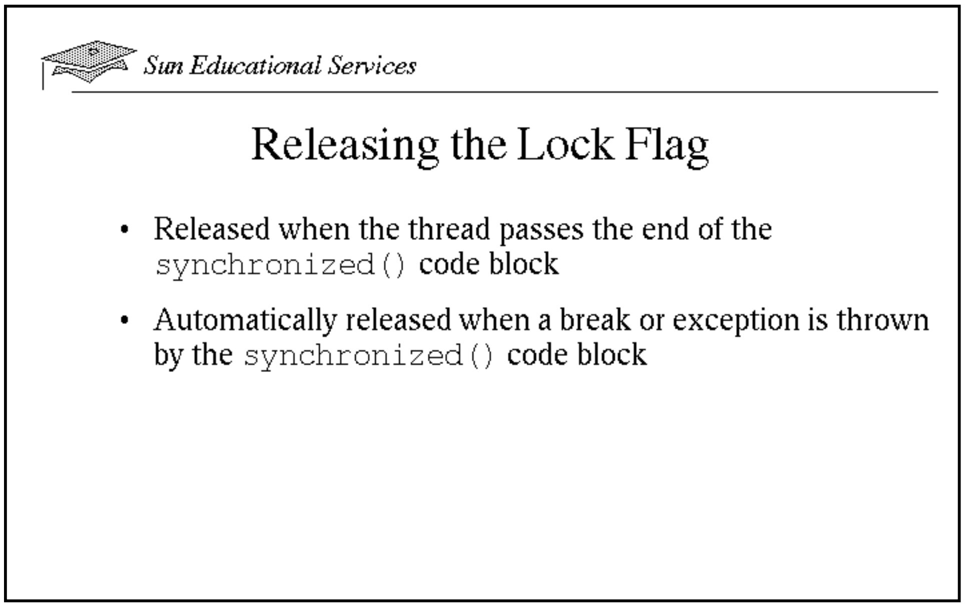
Here is what happens if `pop()` is protected by `synchronized` and another thread tries to execute an object's `pop()` method while the original thread holds the synchronized object's lock flag:



```
public char pop() {
  synchronized (this) {
    idx--;
    return data[idx];
  }
}
```

**Figure 13-6** Thread Trying to Execute `synchronized`

When the thread tries to execute the `synchronized(this)` statement, it tries to take the lock flag from the object `this`. Since the flag is not present, the thread cannot continue execution. The thread then joins a pool of waiting threads which are associated with *that* object's lock flag. When the flag is returned to the object, a thread that was waiting for the flag is given it and the thread continues to run.


Sun Educational Services

### Releasing the Lock Flag

- Released when the thread passes the end of the `synchronized()` code block
- Automatically released when a break or exception is thrown by the `synchronized()` code block

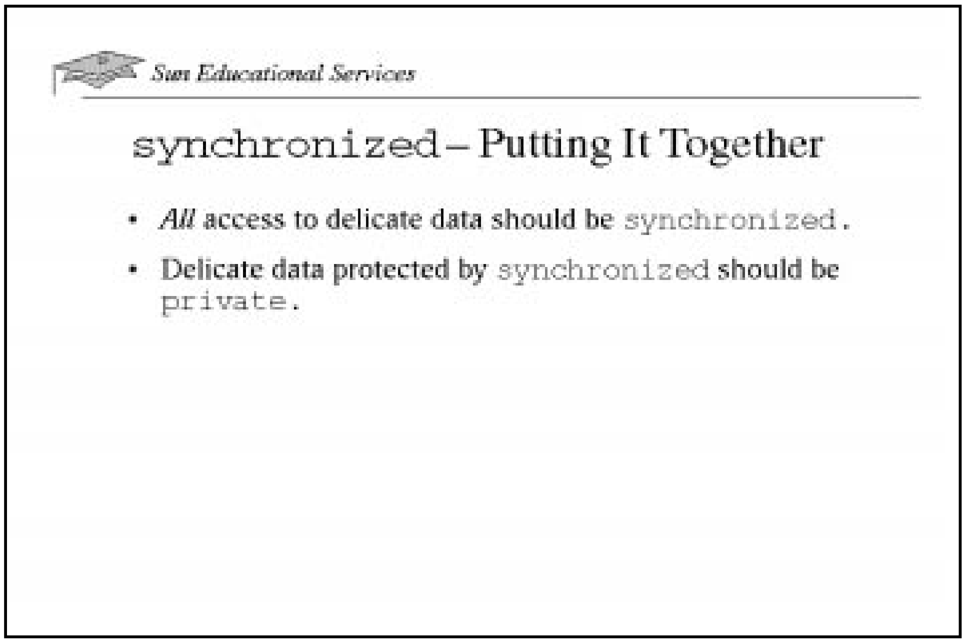
## *Using* `synchronized` *in Java Technology*

### *Releasing the Lock Flag*

Since a thread waiting for the lock flag of an object cannot resume running until the flag is available, it is important for the holding thread to return the flag when it is no longer needed.

The lock flag is given back to its object automatically. When the thread which holds the lock passes the end of the `synchronized()` code block for which the lock was obtained, the lock is released. Java technology takes great care to ensure that the lock is always returned automatically, even if an encountered exception or break statement would transfer code execution out of a synchronized block. Also, if a thread executes nested blocks of code which are synchronized on the same object, that object's flag will be correctly released on exit from the outermost block and the innermost one ignored.

These rules make using synchronized blocks much simpler to manage than equivalent facilities in some other systems.



Sun Educational Services

synchronized – Putting It Together

- *All* access to delicate data should be synchronized.
- Delicate data protected by synchronized should be private.

## *Using* synchronized *in Java Technology*

### synchronized *– Putting It Together*

As has been suggested, the synchronized mechanism works only if *all* access to delicate data are made within the synchronized blocks.

All delicate data protected by synchronized blocks should be marked as *private*. Consider the accessibility of the data items which form the delicate parts of the object. If these are not marked as `private`, they can be accessed from code outside the class definition itself; thus other programmers must not omit the protections that are required.

## *Using* `synchronized` *in Java Technology*

### `synchronized` – *Putting It Together (Continued)*

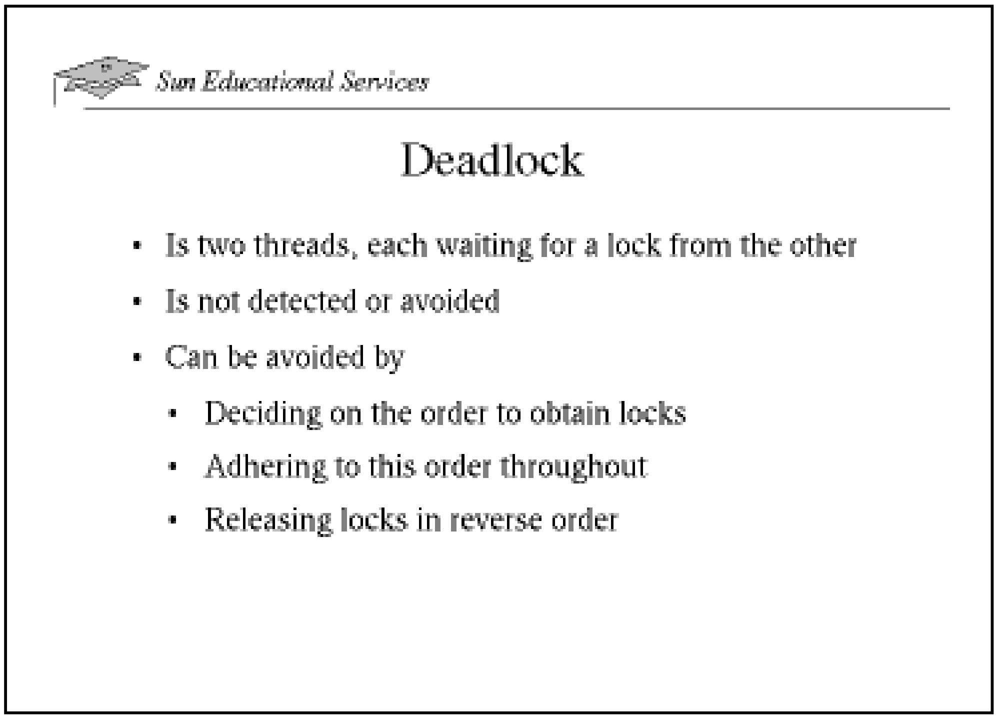
A method consisting entirely of code belonging in a block sychronized to `this` instance might put the `synchronized` keyword in its header. The following two code fragments are equivalent:

```
public void push(char c) {
   synchronized(this) {
   :
   :
   }
}



public synchronized void push(char c) {
   :
   :
}
```

Why use one technique instead of the other?

If you use `synchronized` as a method modifier, the whole method becomes a synchronized block. That can result in the lock flag being held far longer than necessary.

However, marking the method in this way allows users of the method to know, from `javadoc`-generated documentation, that synchronization is taking place. This can be important when designing against deadlock (which is discussed in the next section). Note that the `javadoc` documentation generator propagates the `synchronized` modifier into documentation files, but it cannot do the same for `synchronized(this)` which is found *inside* the method's block.



Sun Educational Services

Deadlock

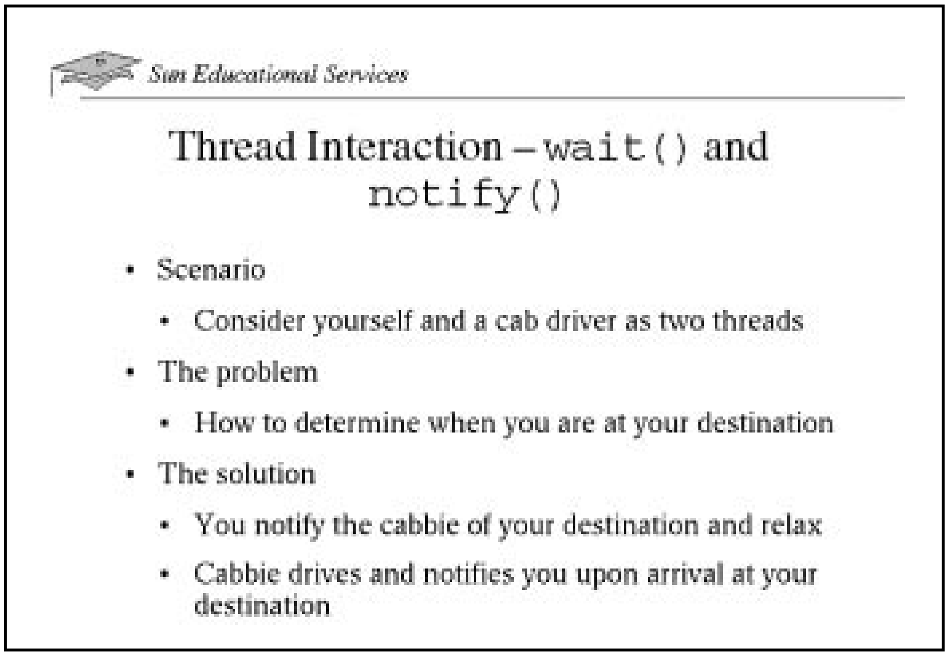
- Is two threads, each waiting for a lock from the other
- Is not detected or avoided
- Can be avoided by
    - Deciding on the order to obtain locks
    - Adhering to this order throughout
    - Releasing locks in reverse order

## *Using* `synchronized` *in Java Technology*

### *Deadlock*

In programs where multiple threads are competing for access to multiple resources, a condition known as *deadlock* can occur. This occurs when one thread is waiting for a lock held by another thread, but the other thread is waiting for a lock already held by the first thread. In this condition, neither can proceed until after the other has passed the end of its `synchronized` block. Since neither is able to proceed, neither can pass the end of its block.

Java technology neither detects nor attempts to avoid this condition. It is therefore the responsibility of the programmer to ensure that a deadlock cannot arise. A general rule of thumb for avoiding a deadlock is: if you have multiple objects that you want to have synchronized access to, make a global decision about the order in which you will obtain those locks, and adhere to that order throughout the program. Release the locks in reverse order that you obtained them.



Sun Educational Services

## Thread Interaction – `wait()` and `notify()`

- Scenario
  - Consider yourself and a cab driver as two threads
- The problem
  - How to determine when you are at your destination
- The solution
  - You notify the cabbie of your destination and relax
  - Cabbie drives and notifies you upon arrival at your destination

## *Thread Interaction –* `wait()` *and* `notify()`

✓ ***Use*** `wait()` ***and*** `notify()` ***to*** *communicate* ***between threads. If multiple threads are waiting on the same renedezvous object, then they must all be waiting for the*** *same* ***condition.***

Often different threads are created specifically to perform unrelated tasks. Sometimes, however, the jobs they perform are actually related in some way, and it might be necessary to program some interactions between them.

### *Scenario*

Consider yourself and a cab driver as two threads. You need a cab to take you to a destination, and the cabbie wants to take on a passenger to make a fare. So, each of you has a task.

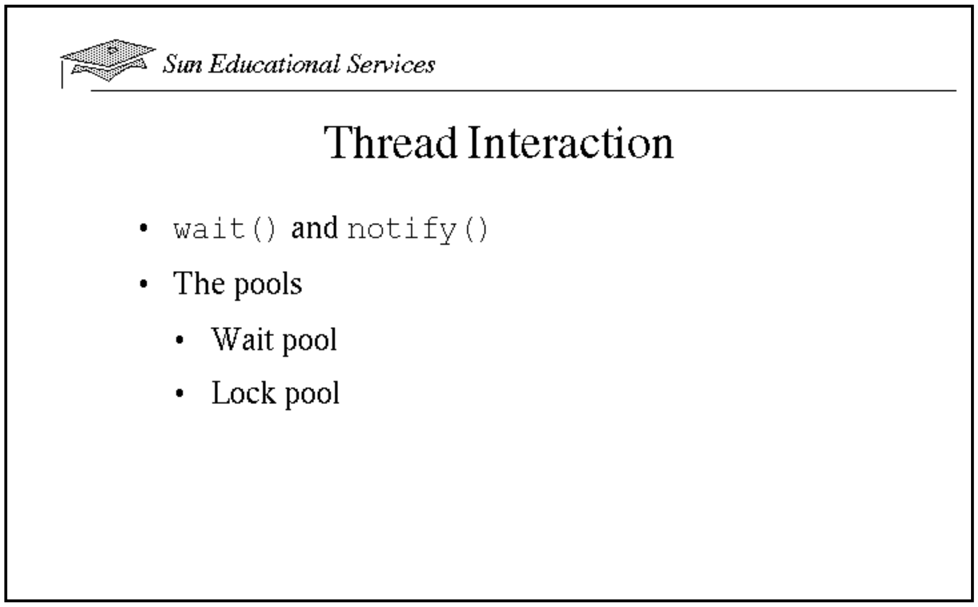
## *Thread Interaction –* `wait()` *and* `notify()`

### *The Problem*

You expect to get into a cab and rest comfortably until the cabbie notifies you that you have arrived at your destination. It would be annoying, for both you and the cabbie, to ask every 2 seconds, "are we there yet?" Between fares, the cabbie wants to sleep in the cab until a passenger needs to be driven somewhere. The cabbie does not want to have to wake up from this nap every 5 mintues to see if a passenger has arrived at the cab stand. So, both threads would prefer to get their jobs done in as relaxed a manner as possible.

### *The Solution*

You and the cabbie require some way of communicating your needs to each other. While you are busy walking down the street toward the cab stand, the cabbie is sleeping peacefully in the cab. When you notify the cabbie that you want a ride, the cabbie wakes up and begins driving, and you get to relax. Once you have arrived at your destination, the cabbie notifies you, to get out of the cab and go to work. The cabbie now gets to wait and nap again until the next fare comes along.



Sun Educational Services

## Thread Interaction

- `wait()` and `notify()`
- The pools
  - Wait pool
  - Lock pool

## *Thread Interaction*

### `wait()` *and* `notify()`

Two methods, `wait()` and `notify()`, are provided in the `java.lang.Object` class for thread communication. If a thread issues a `wait()` call on a rendezvous object `x`, that thread will pause its execution until another thread issues a `notify()` call on the same rendezvous object `x`.

In the previous scenario, the cabbie waiting in the cab translates to the “cabbie” thread executing a `cab.wait()` call, and your need to use the cab translates to the “you” thread executing a `cab.notify()` call.

In order for a thread to call either `wait()` or `notify()` on an object, the thread must have the lock for that particular object. In other words, `wait()` and `notify()` can only be called from within a synchronzied block on the instance on which they are being called. For this example, a block starting with `synchronized(cab)` is required to permit either the `cab.wait()` or the `cab.notify()` call.

## *Thread Interaction*

### `wait()` *and* `notify()` *(Continued)*

#### *The Pool Story*

When a thread executes synchronized code that contains a `wait()` call on a particular object, that thread is placed in the wait pool for that object. Additionally, the thread which calls `wait()` automatically releases that object's lock flag. Different `wait()` methods can be invoked.

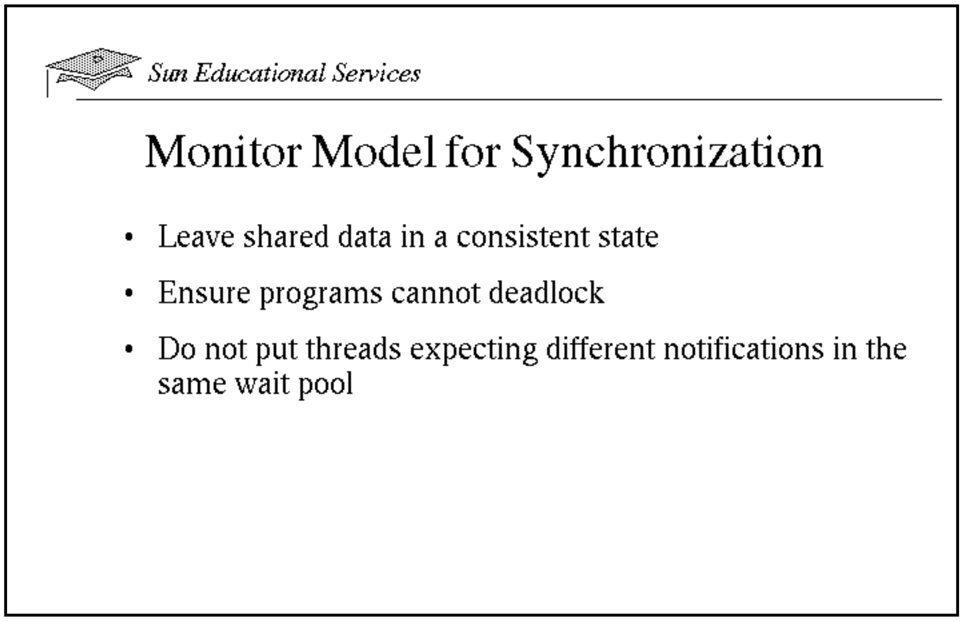
`wait()` or `wait(long timeout)`

✓ ***If a thread that calls*** `wait()` ***happens to be holding more than one object's lock flag,*** *only* ***the lock flag for the object associated with the*** `wait()` ***call is released.***

When a `notify()` call is executed on a particular object, an *arbitrary* thread is moved from that object's wait pool to a lock pool where threads stay until the object's lock flag becomes available. The `notifyAll()` method moves all threads waiting on that object out of the wait pool and into the lock pool. Only from the lock pool can a thread obtain that object's lock flag which allows the thread to continue running where it left off when it called `wait()`.

In many systems that implement the `wait()`/`notify()` mechanism, the thread that wakes up is the one that has been waiting the longest. Java technology, however, does not guarantee this.

Notice that a `notify()` call can be issued without regard to whether any threads are waiting. If the `notify()` method is called on an object when no threads are blocked in the wait pool for that object's lock flag, the call has no effect. Calls to `notify()` are not stored.



Sun Educational Services

## Monitor Model for Synchronization

- Leave shared data in a consistent state
- Ensure programs cannot deadlock
- Do not put threads expecting different notifications in the same wait pool

## *Thread Interaction*

### *Monitor Model For Synchronization*

Coordination between two threads needing access to a *common* data can get very complex. You must take great care that no thread leaves shared data in an inconsistent state when there is the possibility that any other thread can access that data. You also must ensure that your program cannot deadlock because threads are not able to release the appropriate lock when other threads are waiting for that lock.

In the cab example, the code relied on one rendezvous object, the cab, on which `wait()` and `notify()` were executed. If anyone were expecting a bus, you would need a separate bus object on which to apply `notify()`. Remember that all threads in the same wait pool must be satisified by notification from *that* wait pool's controlling object. Never design code that puts threads expecting to be notified for *different* conditions in the *same* wait pool.

## Putting It Together

What follows is an example of thread interaction which demonstrates the use of `wait()` and `notify()` methods to solve a classic producer-consumer problem.

Start by looking at the outline of the stack object and the details of the threads that will access it. Then look at the details of the stack, and the mechanisms used to protect the stack's data and to implement the thread communication based on the stack's state.

The example stack class, called `SyncStack` to distinguish it from the core class `java.util.Stack`, offers the following public API:

```
public synchronized void push(char c);
public synchronized char pop();
```

## Putting It Together

### Producer

The producer thread runs the following method:

```
public void run() {
  char c;

  for (int i = 0; i < 200; i++) {
    c = (char)(Math.random() * 26 +'A');
    theStack.push(c);
    System.out.println("Producer" + num + ": " + c);
    try {
      Thread.sleep((int)(Math.random() * 300));
    } catch (InterruptedException e) {
      // ignore it
    }
  }
}
```

This generates 200 random uppercase characters and pushes them onto the stack with a random delay of 0 to 300 milliseconds between each push. Each pushed character is reported on the console, along with an identifier for which producer thread is executing.

## Putting It Together

### Consumer

The consumer thread runs the following method:

```
public void run() {
  char c;
  for (int i = 0; i < 200; i++) {
    c = theStack.pop();
    System.out.println("Consumer" + num + ": " + c);

    try {
      Thread.sleep((int)(Math.random() * 300));
    } catch (InterruptedException e) { }

  }
}
```

This collects 200 characters from the stack, with a random delay of 0 to 300 milliseconds between each attempt. Each popped character is reported on the console, along with an identifier for which consumer thread is executing.

Now consider construction of the stack class. You are going to create a stack that has a seemingly limitless size, using the Vector class. With this design, your threads will only have to communicate based on whether the stack is empty.

✓ ***Designing code that also communicates about the stack being full is more complex. It requires multiple rendezvous objects and the use of semaphores. Time constraints prevent discussing such a complex example here.***

✓ ***You might remember the producer/consumer example from the older revisions of this course. This code worked because the test harness used only one producer thread and one consumer thread. The*** `notify()` ***calls were made on the single stack object, for*** *different* ***reasons and whether the stack was full or empty. This setup is inherently flawed. As soon as multiple producer and consumer threads are introduced, deadlock occur.***

## Putting It Together

### SyncStack *Class*

A newly constructed SyncStack object's buffer should be empty. The following code can be used to build your class:

```
public class SyncStack {

  private Vector buffer = new Vector(400,200);

  public synchronized char pop() {
  }

  public synchronized void push(char c) {
  }
}
```

Notice the absence of any constructor. It is considered good style to include a constructor, but it has been omitted here for brevity.

## Putting It Together

### SyncStack *Class (Continued)*

Now consider the push() and pop() methods. They must be synchronzied in order to proctect the shared buffer. In addition, if the stack is empty in the pop() method, the executing thread must wait. When the stack in the push() method is no longer empty, waiting threads are notified.

The pop() method is as follows:

```
public synchronized char pop() {
  char c;
  while (buffer.size() == 0) {
    try {
      this.wait();
    } catch (InterruptedException e) {
      // ignore it...
    }
  }
  c = ((Character)buffer.remove(buffer.size()-1)).
      charValue();
  return c;
}
```

The wait() call is made with respect to the stack object which shows how the rendezvous is being made with a *particular object*. Nothing can be popped from the stack when it is empty, so a thread trying to pop data from the stack must wait until the stack is no longer empty.

The wait() call is placed in a try/catch block because an interrupt() call can terminate the thread's waiting period. The wait() must also be within a loop for this example. It must wait if its wait() is interrupted and the stack is still empty.

The pop() method for the stack is synchronized for two reasons. First, popping a character off of the stack affects the shared data buffer. Second, the call to this.wait() must be within a block that is synchronized on the stack object which is represented by this.

## Putting It Together

### SyncStack *Class (Continued)*

You will now see how the push() method uses this.notify() to release a thread from the stack object's wait pool. Once a thread is released, can obtain the lock on the stack, and continue executing the pop() method which removes a character from the stack's buffer.

---

**Note –** In pop(), the wait() method is called *before* any modifications are made to the stack's shared data. This is a crucial point, since the data *must* be in a consistent state before the object's lock is released and a thread's execution changes the stack's data.

---

Another point to be considered is of error checking. You might notice that there is no explicit code to prevent a stack underflow. This is not necessary because the only way to remove characters from the stack is through the pop() method, and this method causes the executing thread to enter the wait() state if no character is available. Hence, error checking is unnecessary.

The push() method is similar; it affects the shared buffer and must also be synchronized. In addition, since the push() method adds a character to the buffer, it is responsible for notifying threads that are waiting for a non-empty stack. This notification is done with respect to the stack object.

The push() method is as follows:

```
public synchronized void push(char c) {
  this.notify();
  Character charObj = new Character(c);
  buffer.addElement(charObj);
}
```

## *Putting It Together*

### `SyncStack` *Class (Continued)*

The call to `this.notify()` serves to release a *single* thread which called `wait()` because the stack is empty. Calling `notify()` before the shared data actually gets changed is of no consequence. The stack object's lock is released only upon exit from the synchronized block, so threads waiting for that lock can obtain it while the stack data are being changed by the `pop()` method.

## SyncStack Example

### Complete Code

The producer, consumer, and stack code now must be assembled into complete classes. A test harness is required to bring these pieces together. Pay particular attention to how SyncTest creates only one stack object that is *shared by all threads*.

#### SyncTest.java

```
package mod13;

public class SyncTest {

  public static void main(String[] args) {

    SyncStack stack = new SyncStack();

    Producer p1 = new Producer(stack);
    Thread prodT1 = new Thread (p1);
    prodT1.start();

    Producer p2 = new Producer(stack);
    Thread prodT2 = new Thread (p2);
    prodT2.start();

    Consumer c1 = new Consumer(stack);
    Thread consT1 = new Thread (c1);
    consT1.start();

    Consumer c2 = new Consumer(stack);
    Thread consT2 = new Thread (c2);
    consT2.start();
  }
}
```

## SyncStack Example

### Complete Code

#### Producer.java

```
package mod13;

public class Producer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Producer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = (char)(Math.random() * 26 +'A');
      theStack.push(c);
      System.out.println("Producer" +num+ ": " +c);
      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) {
        // ignore it
      }
    }
  }
}
```

## SyncStack Example

### Complete Code

#### Consumer.java

```
package mod13;

public class Consumer implements Runnable {
  private SyncStack theStack;
  private int num;
  private static int counter = 1;

  public Consumer (SyncStack s) {
    theStack = s;
    num = counter++;
  }

  public void run() {
    char c;
    for (int i = 0; i < 200; i++) {
      c = theStack.pop();
      System.out.println("Consumer"+num+": " +c);

      try {
        Thread.sleep((int)(Math.random() * 300));
      } catch (InterruptedException e) { }

    }
  }
}
```

## SyncStack Example

### Complete Code

#### SyncStack.java

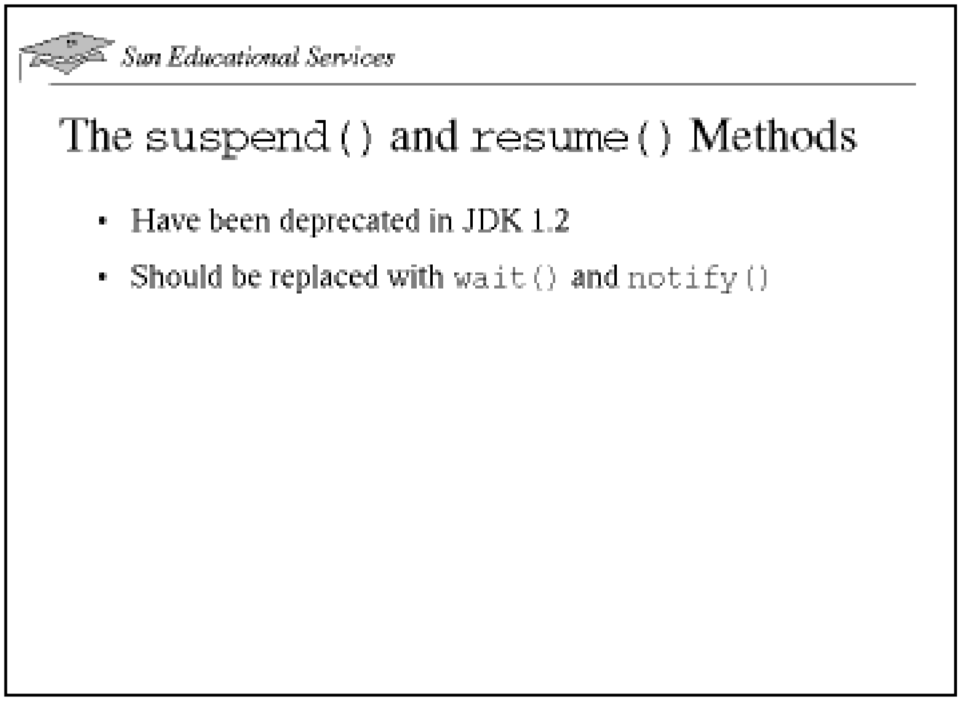
```
package mod13;

import java.util.Vector;

public class SyncStack {
  private Vector buffer = new Vector(400, 200);

  public synchronized char pop() {
    char c;
    while (buffer.size() == 0) {
      try {
        this.wait();
      } catch (InterruptedException e) {
        // ignore it...
      }
    }
    c = ((Character)buffer.remove(buffer.size()-1)).
        charValue();
    return c;
  }

  public synchronized void push(char c) {
    this.notify();
    Character charObj = new Character(c);
    buffer.addElement(charObj);
  }
}
```

## SyncStack Example

### Complete Code

An example of the output from `java mod13.SyncTest` follows. Every time this thread code is run, the results will vary.

```
Producer2: F
Consumer1: F
Producer2: K
Consumer2: K
Producer2: T
Producer1: N
Producer1: V
Consumer2: V
Consumer1: N
Producer2: V
Producer2: U
Consumer2: U
Consumer2: V
Producer1: F
Consumer1: F
Producer2: M
Consumer2: M
Consumer2: T
```



Sun Educational Services

## The suspend() and resume() Methods

- Have been deprecated in JDK 1.2
- Should be replaced with wait() and notify()

## *Thread Control in JDK 1.2*

### *The `suspend()` and `resume()` Methods*

The methods `suspend()` and `resume()` have been deprecated as of JDK 1.2. The `resume()` method's sole purpose is to unsuspend threads, so without `suspend()`, there is no longer a need for `resume()`. `suspend()` is inherently dangerous from a design perspective for two reasons: it is deadlock-prone, and it allows one thread direct control over another thread's code execution. Each danger will be explained in turn.

Assume that you have two threads, *threadA* and *threadB*. While executing its code, *threadB* obtains the lock on an object and then continues with its task. *threadA*'s executing code calls `threadB.suspend()`, which causes *threadB* to stop executing its code.

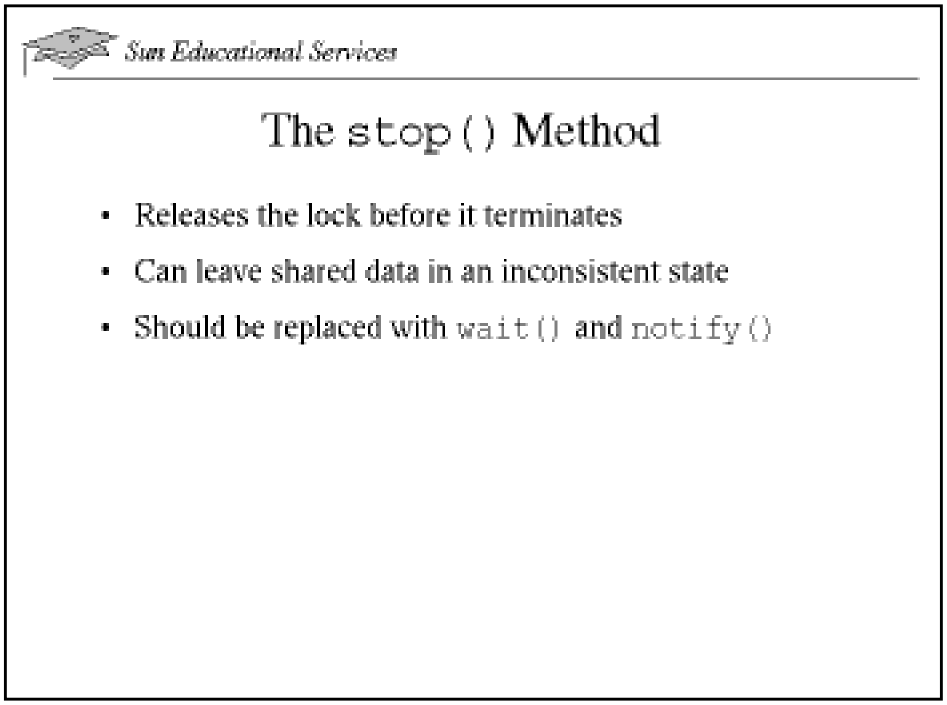
## Thread Control in JDK 1.2

### The `suspend()` and `resume()` Methods

Deadlock can occur because `threadB.suspend()` does not release the lock that *threadB* is holding. If the thread that called `threadB.suspend()` requires the lock which *threadB* is holding, the two threads are deadlocked.

Assume *threadA* calls `threadB.suspend()`. If *threadA* is in control when *threadB* is suspended, *threadB* never gets the opportunity to manipulate the shared data into a steady state. Only *threadB* should determine when to stop executing its own code.

Rather than use `suspend()` and `resume()`, control your threads using `wait()` and `notify()` mechanisms on a rendezvous object. This forces the thread to determine when to "suspend" itself by executing a `wait()` call. This causes the rendezvous object's lock to be released automatically, and gives the thread an opportunity to stabilize any data before calling `wait()`.


Sun Educational Services

The stop() Method

- Releases the lock before it terminates
- Can leave shared data in an inconsistent state
- Should be replaced with wait() and notify()

## *Thread Control in JDK 1.2*

### *The* `stop()` *Method*

The situation with the `stop()` method is similar, but with different consequences. When a thread that holds an object lock is stopped, it releases the lock that it holds before it terminates. This avoids the deadlock problem discussed previously, but it introduces another problem.

In the previous example, if the thread is stopped after the character has been added to the stack but before the index value is incremented, you have an inconsistent stack structure when the lock is released.

There are always certain critical operations that must be executed atomically, and stopping a thread that is executing one of those operations defeats the atomicity of the operation.

## Thread Control in JDK 1.2

### The `stop()` Method (Continued)

A separate, but equally important, issue about stopping threads involves general threads design strategies. Create a thread to do one particular job and live for the life of the entire program. In other words, you do not design your program such that it creates and disposes of threads arbitrarily or creates endless numbers of dialogs or socket endpoints; each takes system resources that are not infinitely available. This does not imply that a thread must execute continuously, it simply means that the proper, safe mechanisms of `wait()` and `notify()` should be used for thread control.

### Proper Thread Control

Now that you know how to design your threads to behave well and communicate using `wait()` and `notify()` calls, and not to rely on `suspend()` or `stop()`, examine the following code. It is important to note that the `run()` method shown ensures that shared data are in a consistent state before execution is paused or terminated.

## Thread Control in JDK 1.2

### Proper Thread Control ( Continued)

```java
public class ControlledThread extends Thread {
  static final int SUSP = 1;
  static final int STOP = 2;
  static final int RUN = 0;
  private int state = RUN;

  public synchronized void setState(int s) {
    state = s;
    if ( s == RUN ) {
      notify();
    }
  }

  public synchronized boolean checkState() {
    while ( state == SUSP ) {
      try {
        wait();
      } catch (InterruptedException e) {
        // ignore
      }
    }
    if ( state == STOP ) {
      return false;
    }
    return true;
  }

  public void run() {
    while ( true ) {
      doSomething();

      // Be sure shared data is in consistent state in
      // case the thread is waited or marked for exiting
      // from run()
      if ( !checkState() ) {
        break;
      }
    }
  }
}
```

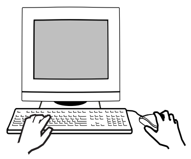
## Thread Control in JDK 1.2

### Proper Thread Control (Continued)

A thread that wants to suspend, resume, or stop a controlled thread calls that thread's `setState()` method with the appropriate value, and when the thread determines that it is safe to do so, it will suspend (by using the `wait()` method) or stop (by exiting from the `run()` method) itself .

A more detailed discussion of this problem is beyond the scope of this module.

## Exercise: Using Multithreaded Programming



**Exercise objective –** In this lab you will become familiar with the concepts of multithreading by writing some multithreaded programs. Create a multithreaded applet.

### Preparation

In order to successfully complete this lab, you must understand the concepts of multithreading as presented in this module.

### Tasks

#### *Level 1 Lab: Create Three Threads*

Perform the following steps:

1. Create a simple program called `ThreeThreads.java` that creates three threads which display the time that it ran. (Consider using the `Date()` class.)

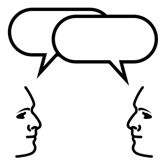
#### *Level 2 Lab: Use Animation*

Complete the following steps:

1. Create an applet called `ThreadedAnimation.java` that takes 10 images of Duke™ waving (in the `graphics/Duke` directory) and displays them in a sequence that makes it look like Duke is waving.
2. Use the `MediaTracker` class to smooth the loading of these images.
3. Allow the user to stop and start the animation using successive mouse clicks.

## Exercise: Using Multithreaded Programming

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explores with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Define a thread
- ❑ Create separate threads in a Java software program, controlling the code and data that are used by that thread
- ❑ Control the execution of a thread and write platform-independent code with threads
- ❑ Describe the difficulties that might arise when multiple threads share data
- ❑ Use the keyword `synchronized` to protect data from corruption
- ❑ Use `wait()` and `notify()` to communicate beween threads
- ❑ Use `synchronized` to protect data from corruption
- ❑ Explain why `suspend()`, `resume()` and `stop()` methods have been deprecated in JDK 1.2

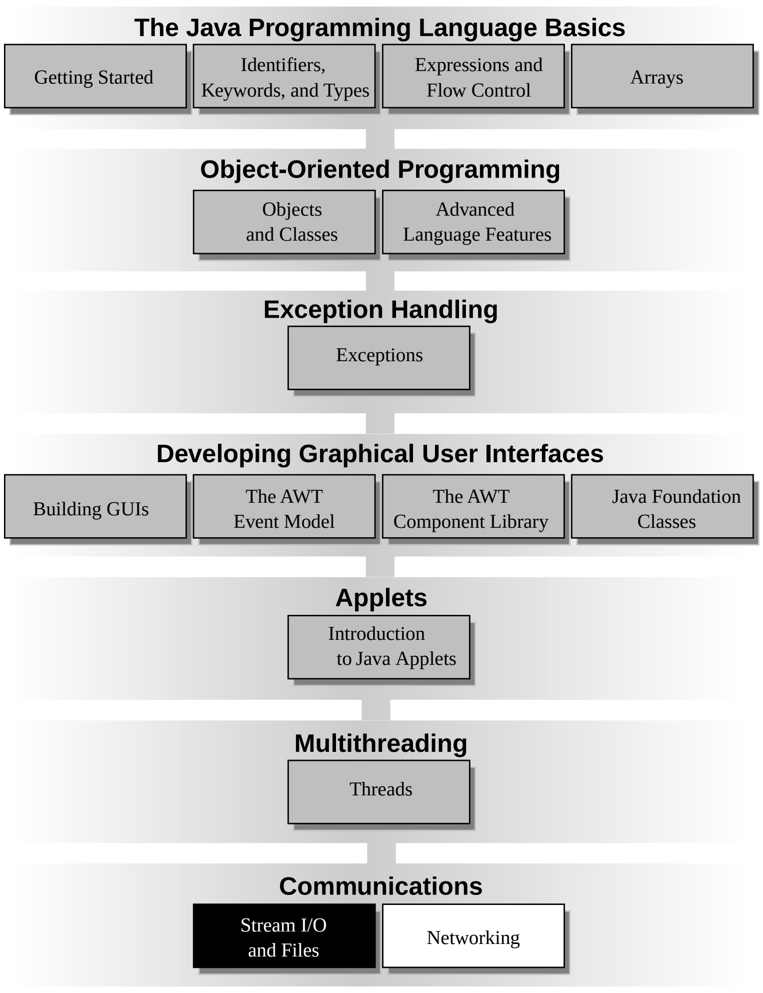
## Think Beyond

Do you have applications that could benefit from being multithreaded?

# Stream I/O and Files 14

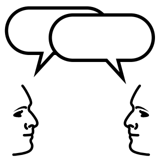
## Course Map

This module discusses the stream I/O mechanism used for files, sockets, and other sources of data.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

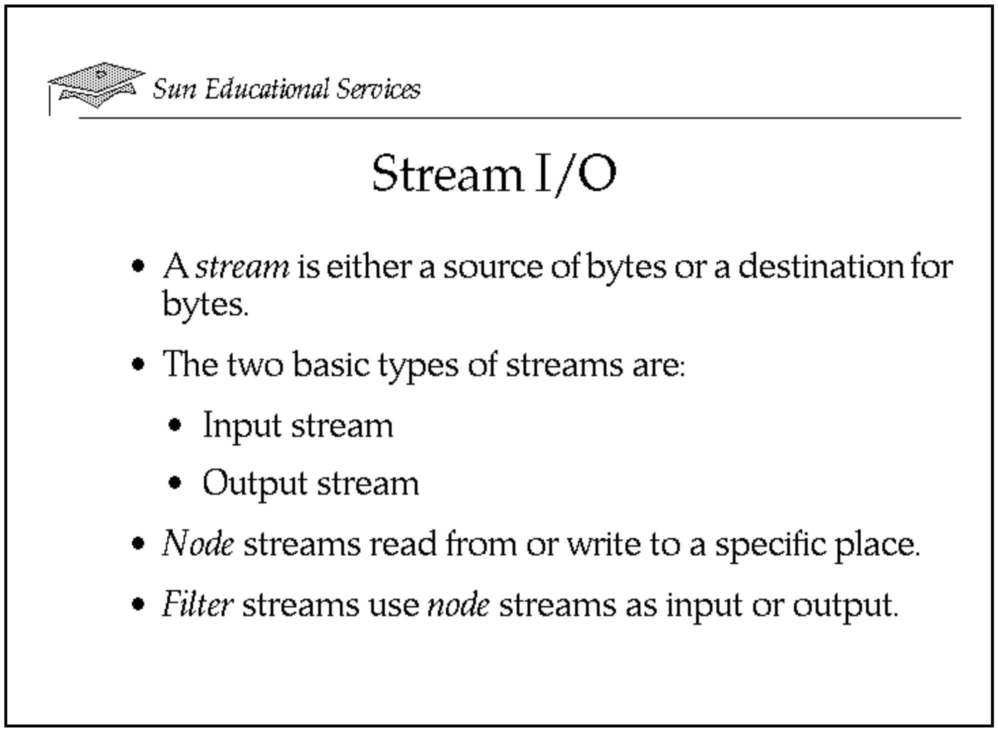
- What mechanisms are in place within the Java programming language to read and write from files?

✓ ***Input can come from more places than just a GUI, and output can go to more places than just the screen or console. Oftentimes a program needs to save its state or some data to a file, or communicate over the network to a server. This module describes the framework of the Java platform's stream I/O mechanism. The framework's flexibility is demonstrated by showing how it is easily adapted to various I/O scenarios. This module shows how whole objects can be saved and restored, allowing them to be reinstantiated without being reinitialized. Other handy file-related classes are also covered.***

## Objectives

At the end of this module, you should be able to

- Describe the main features of the `java.io` package
- Construct file and filter streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them
- Examine and manipulate files and directories
- Read, write, and update text and data files
- Use the Serialization interface to persist the state of objects


Sun Educational Services

## Stream I/O

- A *stream* is either a source of bytes or a destination for bytes.
- The two basic types of streams are:
  - Input stream
  - Output stream
- *Node* streams read from or write to a specific place.
- *Filter* streams use *node* streams as input or output.

## *Stream I/O*

This module examines how the Java programming language uses streams to handle byte and character, and object I/O. Later sections examine some specific details of handling files and working with the data they contain.

### *Stream Fundamentals*

A *stream* is either a source of or a destination for bytes. The order is significant. For example, a program that wants to collect input from a keyboard can use a stream to do so.

The two basic categories of streams are: *input streams* and *output streams*. You can read from an input stream, but you cannot write to it. Conversely, you can write to an output stream, but you cannot read from it. To read bytes from an input stream, there must be a source of characters associated with the stream.

## Stream I/O

### Stream Fundamentals (Continued)

In the `java.io` package some streams are *node* streams, that is they read from or write to a specific place such as a disk file or an area of memory. Other streams are called *filters*. A filter input stream is created with a connection to an existing input stream. This is done so that when you try to read from the filter input stream object it supplies you with characters that originally came from the other input stream object.

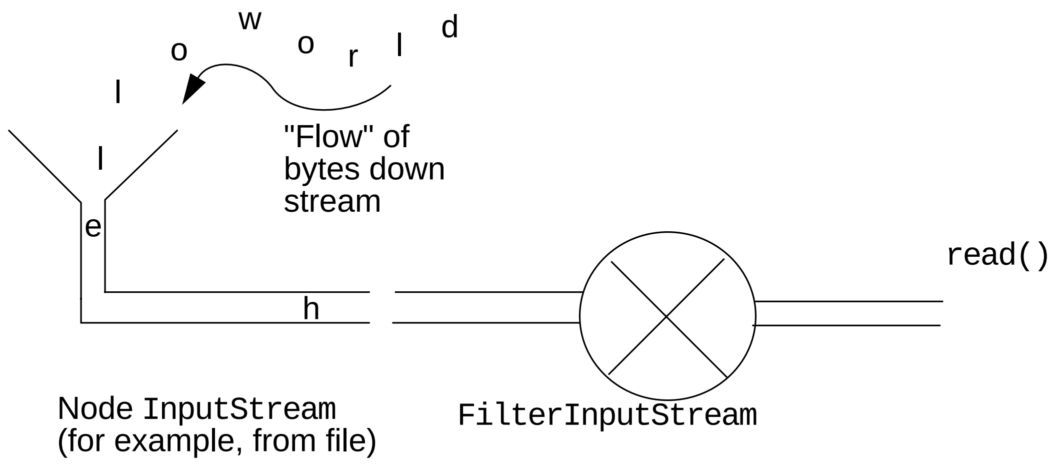
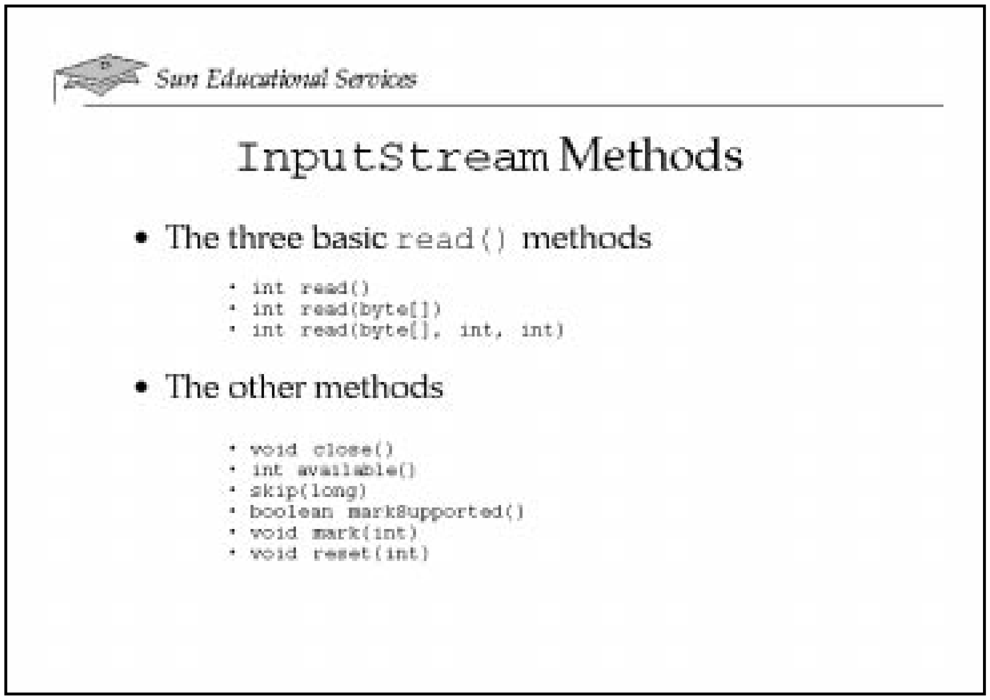

w
o
o
r
l
d
l
l
e
h
"Flow" of bytes down stream
read()
Node InputStream (for example, from file)
FilterInputStream

**Figure 14-1** Streams


Sun Educational Services

InputStream Methods

- The three basic read() methods
  - int read()
  - int read(byte[])
  - int read(byte[], int, int)
- The other methods
  - void close()
  - int available()
  - skip(long)
  - boolean markSupported()
  - void mark(int)
  - void reset(int)

## *Stream I/O*

### `InputStream` *Methods*

```
int read()
int read(byte[])
int read(byte[], int, int)
```

These three methods provide access to the data from the input pipe. The simple `read()` method returns an `int` which contains either a byte read from the stream or -1, which indicates the end of file condition. The other two methods are read into a byte array and return the number of bytes read. The two `int` arguments in the third method indicate a subrange in the target array which needs to be filled.

---

**Note –** For efficiency, always read data in the largest practical block.

---

## Stream I/O

### `InputStream` *Methods (Continued)*

```
void close()
```

When you have finished with a stream, close it. If you have a “stack” of streams, using filter streams, close the stream at the top of the stack. This operation also closes the lower streams.

```
int available()
```

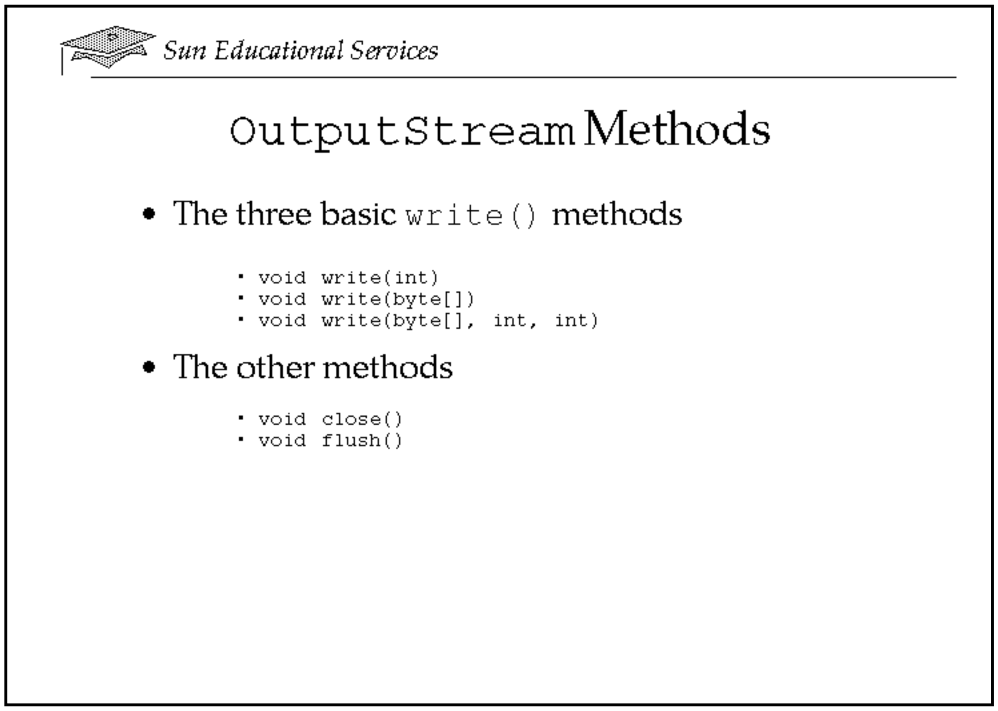
This method reports the number of bytes which are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

```
skip(long)
```

This method discards the specified number of bytes from the stream.

```
boolean markSupported()
void mark(int)
void reset()
```

These methods can be used to perform "push back" operations on a stream if supported by that stream. The `markSupported()` method will return `true` if the `mark()` and `reset()` methods are operational for that particular stream. The `mark(int)` method is used to indicate that the current point in the stream should be noted and a buffer big enough for at least the specified argument number of bytes should be allocated. After subsequent `read()` operations, calling the `reset()` method returns the input stream to the point you marked. If you read past the marked buffer, `reset()` has no meaning.


Sun Educational Services

OutputStream Methods

- The three basic write() methods
  - void write(int)
  - void write(byte[])
  - void write(byte[], int, int)
- The other methods
  - void close()
  - void flush()

## Java Stream I/O

### OutputStream *Methods*

```
void write(int)
void write(byte[])
void write(byte[], int, int)
```

These methods write to the output stream. As with input, always try to write data in the largest practical block.

```
void close()
```

Output streams should be closed when you have finished with them. Again, if you have a stack and close the top one, this closes the rest of the streams.

```
void flush()
```

Sometimes an output stream accumulates writes before committing them. The `flush()` method allows you to force writes.

## Basic Stream Classes

Several stream classes are defined in the `java.io` package. The following diagram illustrates the hierarchy of some of the classes in that package. Some of the more common ones are described in the following sections.

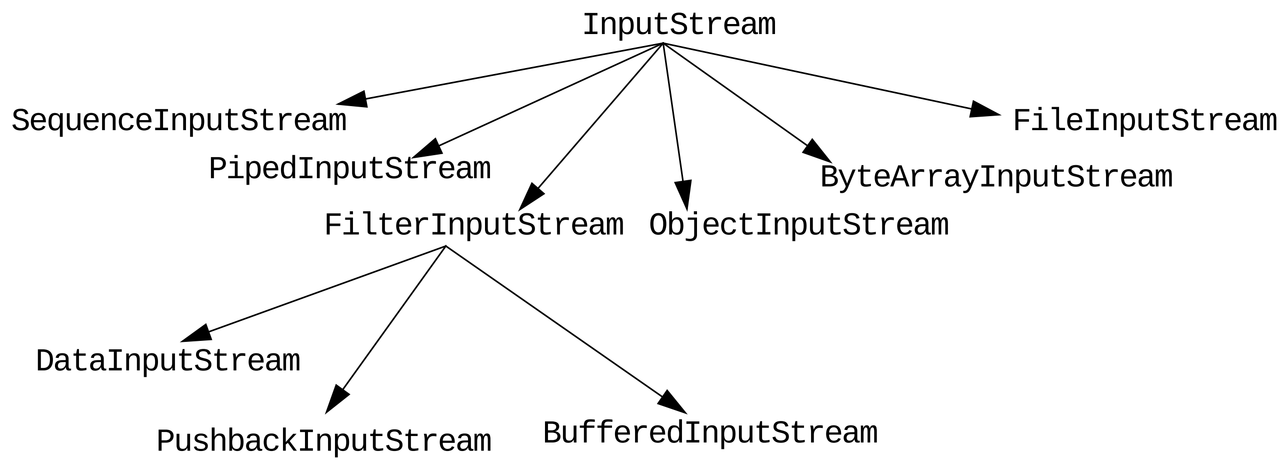
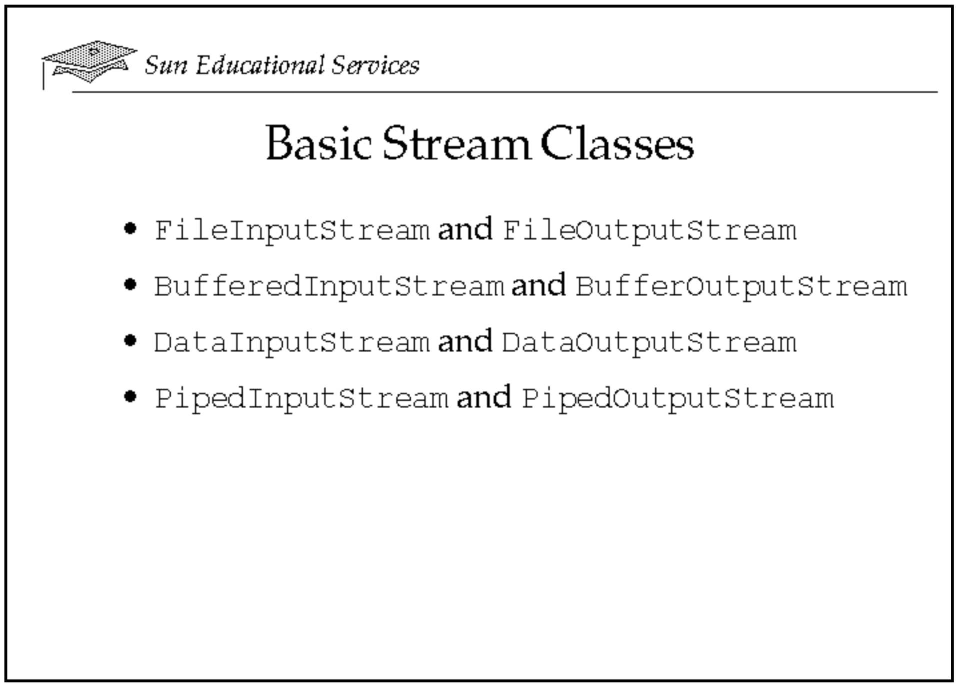


**Figure 14-2** Stream Class Hirearchy



Sun Educational Services

## Basic Stream Classes

- FileInputStream and FileOutputStream
- BufferedInputStream and BufferOutputStream
- DataInputStream and DataOutputStream
- PipedInputStream and PipedOutputStream

## *Basic Stream Classes*

### `FileInputStream` *and* `FileOutputStream`

These classes are node streams and, as the name suggests, they use disk files. The constructors for these classes allow you to specify the path of the file to which they are connected. To construct a `FileInputStream`, the associated file must exist and be readable. If you construct a `FileOutputStream`, the output file will be overwritten if it already exists.

```
FileInputStream infile =
       new FileInputStream("myfile.dat");

FileOutputStream outfile =
       new FileOutputStream("results.dat");
```

### `BufferedInputStream` *and* `BufferedOutputStream`

These are filter streams which should be used to increase the efficiency of I/O operations.

## *Basic Stream Classes*

### `DataInputStream` and `DataOutputStream`

These filter streams allow reading and writing of Java primitive types and some special formats via streams. A number of methods are provided for the different primitives. For example:

#### `DataInputStream` *Methods*

```
byte readByte()
long readLong()
double readDouble()
```

#### `DataOutputStream` *Methods*

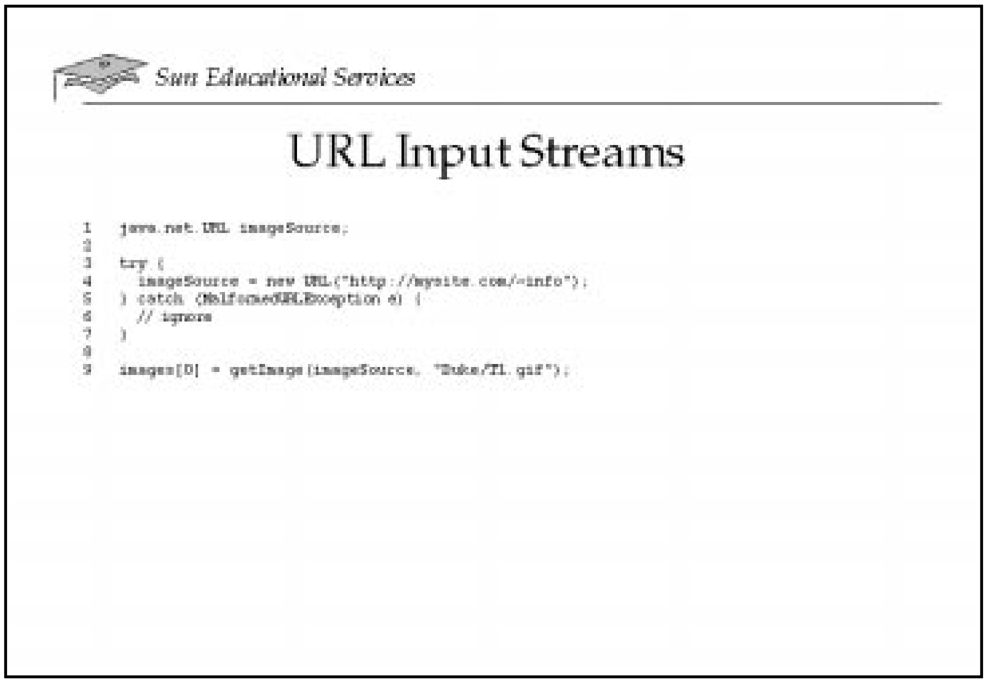
```
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

Notice that the methods of `DataInputStream` are paired with the methods of `DataOutputStream`.

These streams have methods for reading and writing strings, but these methods should not be used. They have been deprecated and replaced by readers and writers that will be discussed later.

### `PipedInputStream` and `PipedOutputStream`

Piped streams can be used for communicating between threads. A `PipedInputStream` object in one thread receives its input from a complementary `PipedOutputStream` object in another thread. The piped streams must have both an input side and an output side to be useful.



Sun Educational Services

## URL Input Streams

```
java.net.URL imageSource;

try {
  imageSource = new URL("http://mysite.com/~info");
} catch (MalformedURLException e) {
  // ignore
}

images[0] = getImage(imageSource, "Duke/T1.gif");
```

## *URL Input Streams*

In addition to basic file access, Java technology provides you with the ability to use uniform resource locators (URLs) as a means of accessing objects across a network. You implicitly use a `URL` object when accessing sounds and images via the `getDocumentBase()` method for applets.

```
String imageFile = new String ("images/Duke/T1.gif");
images[0] = getImage(getDocumentBase(), imageFile);
```

However, you can provide a direct URL as follows:

```
java.net.URL imageSource;

try {
    imageSource = new URL("http://mysite.com/~info");
} catch (MalformedURLException e) {}
  images[0] = getImage(imageSource, "Duke/T1.gif");
```

✓ ***This example assumes the host*** `mysite.com` ***is running an*** `httpd` ***daemon which can handle the request.***

## URL Input Streams

### Opening an Input Stream

You can open an input stream off of an appropriate `URL` object by storing a data file in or below the document base directory.

```
InputStream is = null;
String datafile = new String("Data/data.1-96");
byte buffer[] = new byte[24];
try {
  // new URL throws a MalformedURLException
  // URL.openStream() throws an IOException
  is = (new URL(getDocumentBase(),datafile)).openStream();
} catch (Exception e) {
  // ignore
}
```

Now you can use it to read information, just as with a `FileInputStream` object:

```
try {
  is.read(buffer, 0, buffer.length);
} catch (IOException e1) {
  // ignore
}
```

**Caution –** Remember that most users have their browser security set to prevent applets from accessing files.

## Readers and Writers

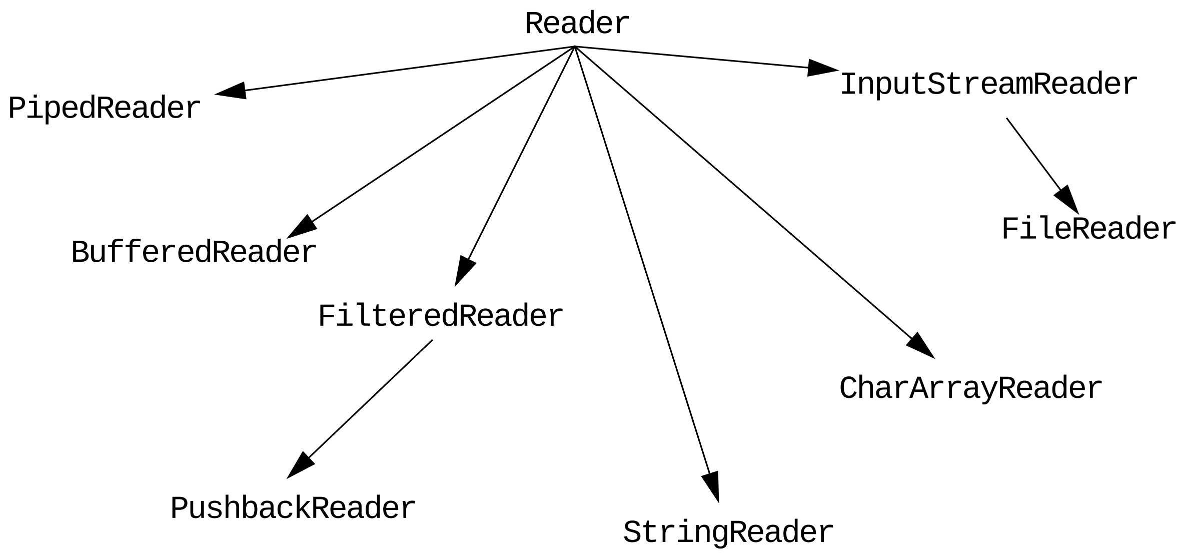


**Figure 14-3** Reader Hirearchy

### Unicode

Java technology uses Unicode to represent strings and characters, and it provides 16-bit versions of streams to allow characters to be treated similarly. These 16-bit versions are called *readers* and *writers*. As with streams, a variety of them are available in the `java.io` package.

## Readers and Writers

### Unicode (Continued)

The most important versions of readers and writers are `InputStreamReader` and `OutputStreamWriter`. These classes are used to interface between byte streams and character readers and writers.

When you construct an `InputStreamReader` or `OutputStreamWriter`, conversion rules are defined to change between 16-bit Unicode and other platform specific representations.

### Byte and Character Conversions

By default, if you construct a reader or writer connected to a stream, the conversion rules will change between bytes using the default platform character encoding and Unicode. In English-speaking countries, the byte encoding used will be International Organization for *Standardization (ISO) 8859-1*.

You can specify an alternative byte encoding by using one of the supported encoding forms. You can find a list of the supported encoding forms in the documentation found at `jdk1.2/docs/guide/internet/encoding.doc.html`.

Using this conversion scheme, Java technology is able to use the full flexibility of the local platform character set while still retaining platform independence through the internal use of Unicode.

### The Buffered Reader and Writer

Because converting between formats is like other I/O operations, it is most efficient when performed in large data blocks. It is a good idea to chain a `BufferedReader` or `BufferedWriter` (as appropriate) onto the end of an `InputStreamReader` or `InputStreamWriter`. Remember to use the `flush()` method on a `BufferedWriter`.

## Readers and Writers

### Reading String Input

The following example shows a technique that should be used to read String information from the console standard input:

```
import java.io.*;

public class CharInput {

  public static void main (String args[])
      throws java.io.IOException {
    String s;
    InputStreamReader ir;
    BufferedReader in;

    ir = new InputStreamReader(System.in);
    in = new BufferedReader(ir);

    while ((s = in.readLine()) != null) {
      System.out.println("Read: " + s);
    }
  }
}
```

### Using Other Character Conversions

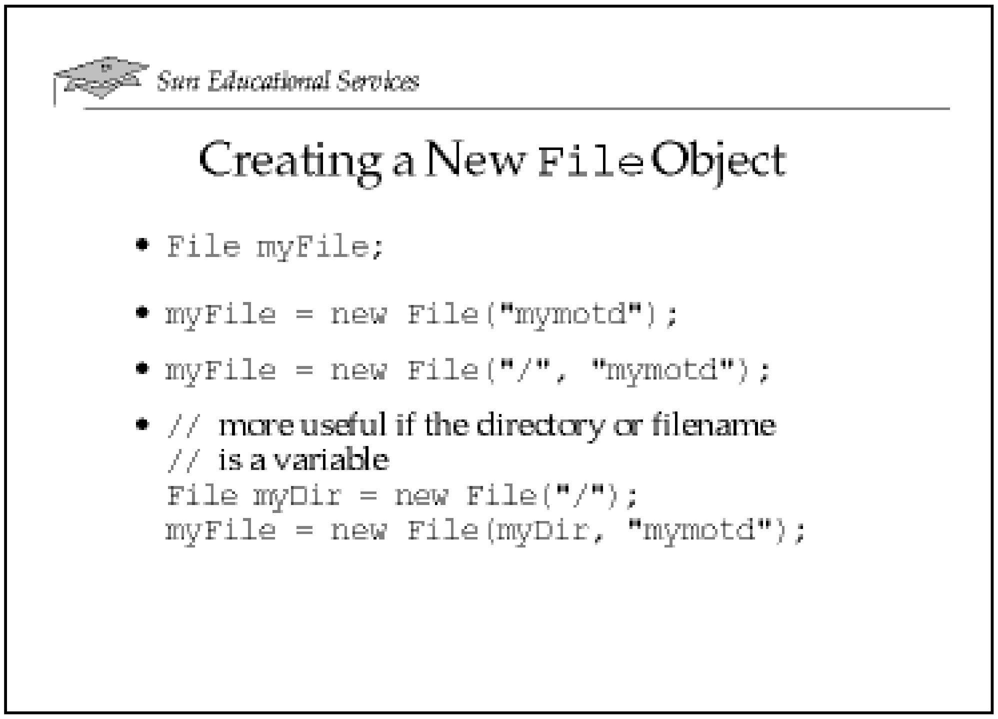
If you need to read input from a character encoding that is not your local one (for example, reading from a network connection with a different type of machine), you can construct the `InputStreamReader` with an explicit character encoding like this:

```
ir = new InputStreamReader(System.in, "ISO8859_1")
```

---

**Note –** If you are reading characters from a network connection, use this form. If you do not, your program will always attempt to convert the characters it reads as if they were in the local representation, which will probably not be correct. ISO 8859_1 is the Latin-1 encoding scheme that maps on to ASCII.

---


Sun Educational Services

## Creating a New File Object

- `File myFile;`
- `myFile = new File("mymotd");`
- `myFile = new File("/", "mymotd");`

-
  ```
  // more useful if the directory or filename
  // is a variable
  File myDir = new File("/");
  myFile = new File(myDir, "mymotd");
  ```

## *Files*

### *Creating a New File Object*

The File class provides several utilities for dealing with files and obtaining basic information about them.

-
  ```
  File myFile;
  myFile = new File("mymotd");
  ```

-
  ```
  myFile = new File("/", "mymotd");
  ```

-
  ```
  // more useful if the directory or filename is
  // a variable
  File myDir = new File("/");
  myFile = new File(myDir, "mymotd");
  ```

## Creating a New File Object (Continued)

The constructor you use often depends on the other file objects you can access. For example, if you use only one file in your application, use the first constructor. If, however, you use several files from a common directory, using the second or third constructors might be easier.

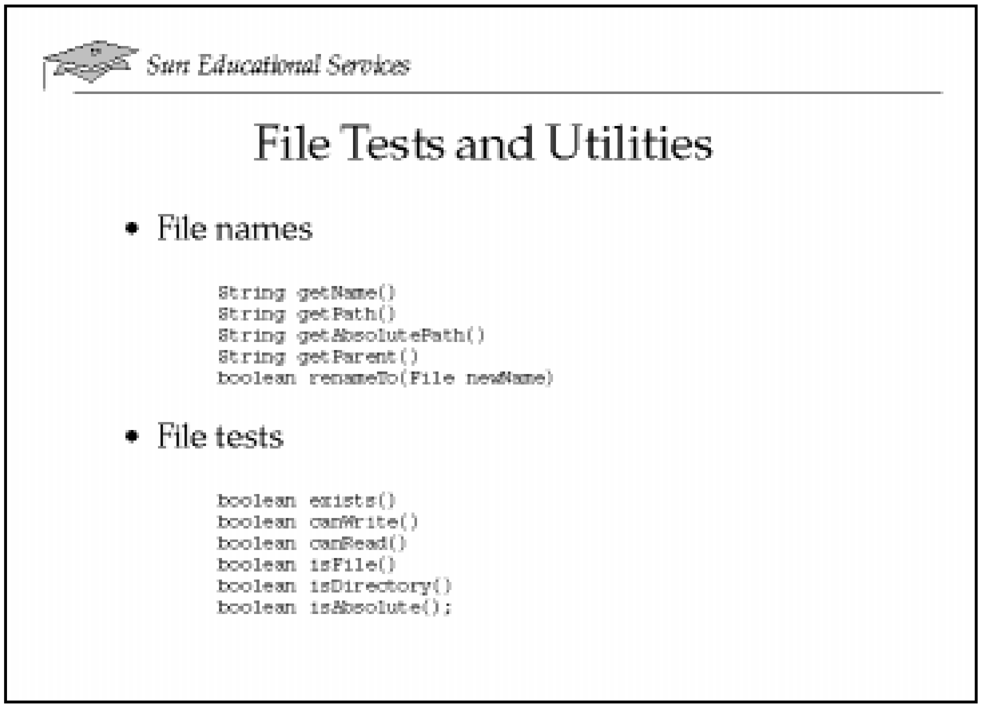
The class `File` defines platform-independent methods for manipulating a file maintained by a native file system. However it does not allow you to access the contents of the file.

---

**Note –** You can use a `File` object as the constructor argument for `FileInputStream` and `FileOutputStream` objects in place of a String. This gives you independence from the local file system conventions and is generally recommended.

---

✓ ***Although Windows 95 and NT use*** `\` ***as directory delimiters,*** `/` ***works within Java*** `File` ***and other streaming classes. If no device is specified (***`/thisDevice/outputFile`***), the paths are considered relative to the current device. If you need to specify a different device, preface the path with the device name (***`D:/otherDevice/inputFile`***). Notice the use of*** `/`***; it is used even though you are referencing files on a Windows 95 or NT system.***



Sun Educational Services

## File Tests and Utilities

- File names

```
String getName()
String getPath()
String getAbsolutePath()
String getParent()
boolean renameTo(File newName)
```

- File tests

```
boolean exists()
boolean canWrite()
boolean canRead()
boolean isFile()
boolean isDirectory()
boolean isAbsolute();
```

## File Tests and Utilities

Once you have created a File object, you can use any of the following methods to gather information about the file:

### File Names

- `String getName()`
- `String getPath()`
- `String getAbsolutePath()`
- `String getParent()`
- `boolean renameTo(File newName)`

### File Tests

- `boolean exists()`
- `boolean canWrite()`

## File Tests and Utilities

### *File Tests (Continued)*

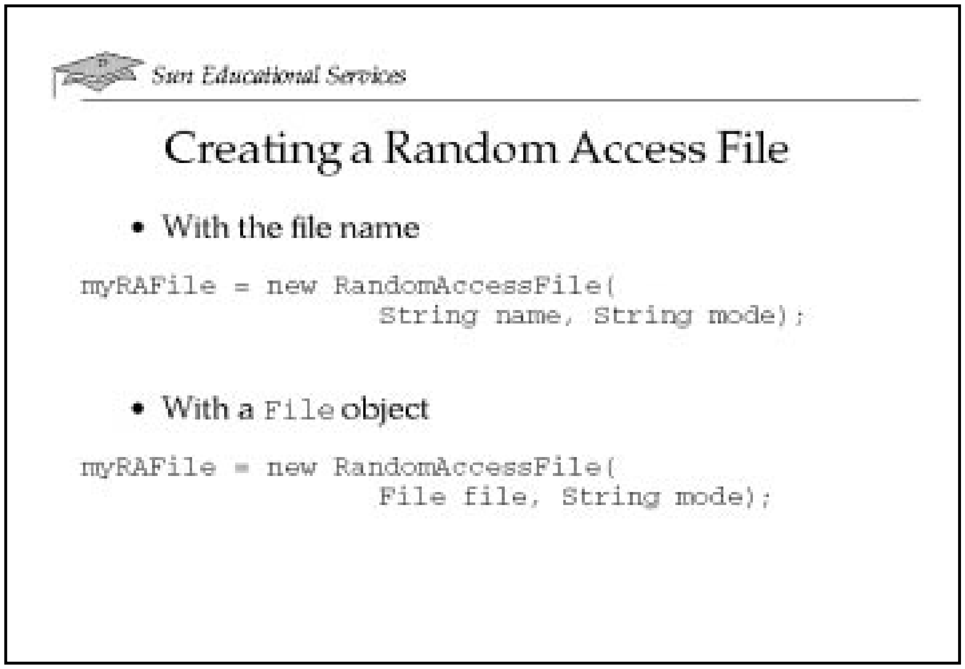
- `boolean canRead()`
- `boolean isFile()`
- `boolean isDirectory()`
- `boolean isAbsolute()`

### *General File Information and Utilities*

- `long lastModified()`
- `long length()`
- `boolean delete()`

### *Directory Utilities*

- `boolean mkdir()`
- `String[] list()`



Sun Educational Services

### Creating a Random Access File

- With the file name

```
myRAFile = new RandomAccessFile(
                 String name, String mode);
```

- With a File object

```
myRAFile = new RandomAccessFile(
                 File file, String mode);
```

## Random Access Files

Often you will find that you want to read data within a file without reading the file from beginning to end. You might want to access a text file as a database in which case you move around reading one record, then another, and then another—each in different parts of the file. The Java programming language provides a RandomAccessFile class for handling this type of input or output.

### Creating a Random Access File

You have two options for opening a random access file:

- With the file name

```
myRAFile = new
     RandomAccessFile(String name, String mode);
```

## *Random Access Files*

### *Creating a Random Access File (Continued)*

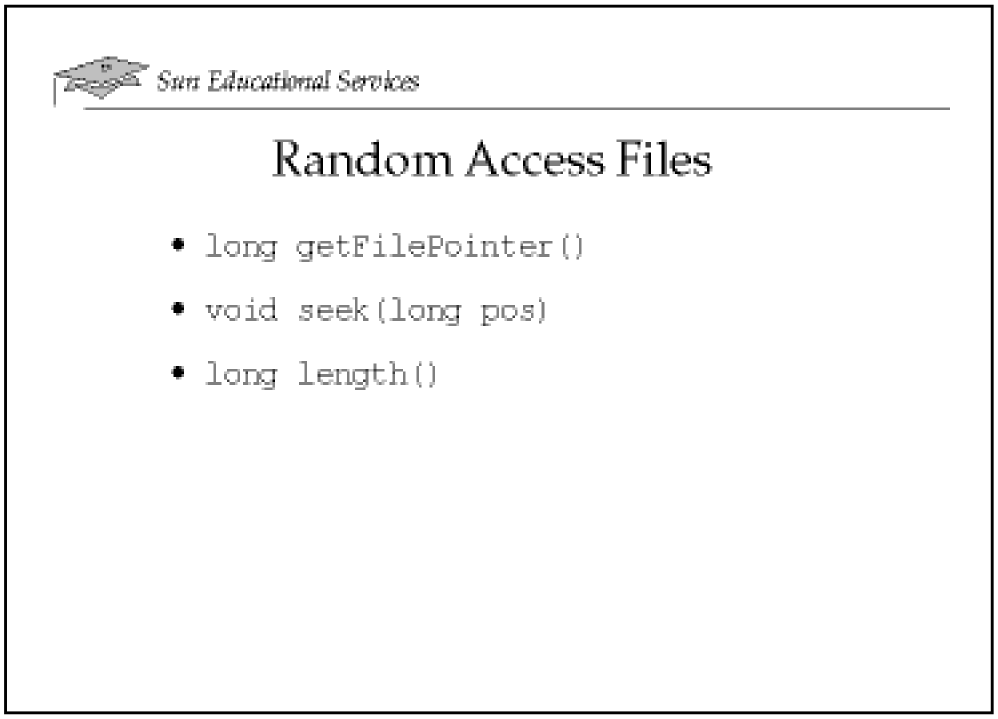
- With a `File` object

```
myRAFile = new RandomAccessFile(
                   File file, String mode);
```

The `mode` argument determines whether you have read-only (*r*) or read / write (*rw*) access to this file.

For example, you can open a database file for updating:

```
RandomAccessFile myRAFile;
myRAFile = new RandomAccessFile("db/stock.dbf","rw");
```


Sun Educational Services

Random Access Files

- long getFilePointer()
- void seek(long pos)
- long length()

## Random Access Files

### Accessing Information

`RandomAccessFile` objects expect to read and write information in the same manner as data input and data output objects. You have access to all of the `read()` and `write()` operations found in the `DataInputStream` and `DataOutputStream` classes.

The Java programming language provides several methods to help you move around inside the file. Obtain the current location of the file pointer:

```
long getFilePointer();
```

## Random Access Files

### Accessing Information (Continued)

```
void seek(long pos);
```

Sets the file pointer to the specified absolute position. The position is given as a byte-offset from the beginning of the file. Position 0 marks the beginning of the file.

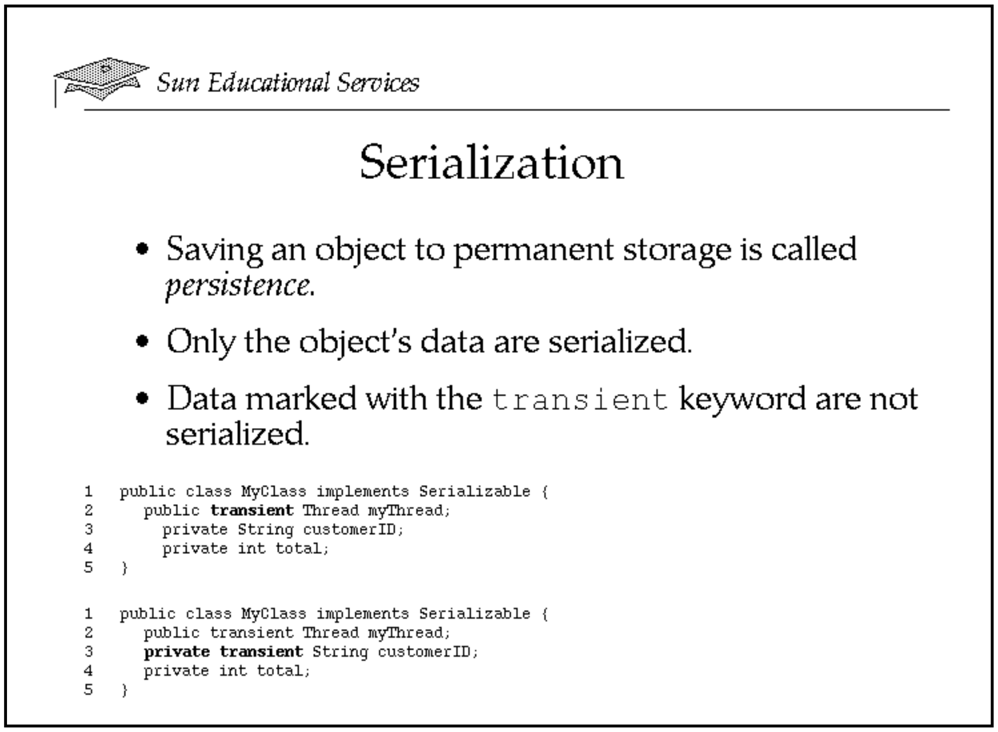
```
long length();
```

Obtain the length of the file. Position `length()` marks the end of the file.

### Appending Information

You can use random access files to accomplish an appending mode for file output.

```
myRAFile = new RandomAccessFile("java.log","rw");
myRAFile.seek(myRAFile.length());
// Any subsequent write()s will be appended to the
// file
```


Sun Educational Services

## Serialization

- Saving an object to permanent storage is called *persistence*.
- Only the object's data are serialized.
- Data marked with the `transient` keyword are not serialized.

```
public class MyClass implements Serializable {
  public **transient** Thread myThread;
    private String customerID;
    private int total;
}
```

```
public class MyClass implements Serializable {
  public transient Thread myThread;
  **private transient** String customerID;
  private int total;
}
```

## *Serialization*

New since JDK 1.1 is the addition of the `java.io.Serializable` interface and changes to the JVM to support the ability to save a Java technology object to a stream.

Saving an object to some type of permanent storage is called *persistence*. An object is said to be *persistent capable* when it is possible to store that object on a disk or tape or send it to another machine to be stored in memory or on disk.

The `java.io.Serializable` interface has no methods and only serves as a "marker" that indicates that the class that implements the interface can be considered for serialization. Objects from classes that do not implement `Serializable` cannot save or restore their state.

✓ ***The implementation of a class which implements `Externalizable` is beyond the scope of this discussion. Also excluded is how readObject() and `writeObject()` can be defined in a `Serializable` class's implementation.***

## Object Graphs

When an object is serialized only the data of the object are preserved; methods and constructors are not part of the serialized stream. When a data variable is an object, the data members of that object are also serialized if that object's class is also serializable. The tree, or structure of an object's data, including these sub-objects, constitutes the object *graph*.

Some object classes are not serializable because the data they represent are constantly changing; for example, `java.io.FileInputStream` and `java.lang.Thread`. If a serializable object contains a reference to a non-serializable element, the entire serialization operation fails and a `NotSerializableException` is thrown.

If the object graph contains a non-serializable object reference, the object can still be serialized if the reference is marked with the `transient` keyword.

```
public class MyClass implements Serializable {
  public transient Thread myThread;
  private String customerID;
  private int total;
}
```

The field access modifier (`public`, `protected`, *default*, and `private`) has no effect on the data field being serialized. Data is written to the stream in byte format and with strings represented as UTF (file system safe universal character set transformation format) characters. The `transient` keyword prevents the data from being serialized.

```
public class MyClass implements Serializable {
  public transient Thread myThread;
  private transient String customerID;
  private int total;
}
```

## Writing and Reading an Object Stream

### Writing

Writing and reading an object to a file stream is a simple process. Consider the following code fragment which sends an instance of a `java.util.Date` object to a file:

```
import java.io.*;
import java.util.Date;

public class SerializeDate {

  SerializeDate() {
    Date d = new Date ();

    try {
      FileOutputStream f =
          new FileOutputStream ("date.ser");
      ObjectOutputStream s =
          new ObjectOutputStream (f);
      s.writeObject (d);
      s.close ();
    } catch (IOException e) {
      e.printStackTrace ();
    }
  }

  public static void main (String args[]) {
    new SerializeDate();
  }
}
```

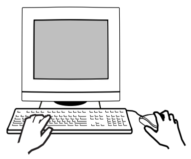
## Writing and Reading an Object Stream

### Reading

Reading the object is as simple as writing it, with one caveat—the `readObject()` method returns the stream as an `Object` type, and it must be cast to the appropriate class name before methods on that class can be executed.

```
import java.io.*;
import java.util.Date;

public class UnSerializeDate {

  UnSerializeDate () {
    Date d = null;

    try {
      FileInputStream f =
          new FileInputStream ("date.ser");
      ObjectInputStream s =
          new ObjectInputStream (f);
      d = (Date) s.readObject ();
      s.close ();
    } catch (Exception e) {
      e.printStackTrace ();
    }

    System.out.println(
      "Unserialized Date object from date.ser");
    System.out.println("Date: "+d);
  }

  public static void main (String args[]) {
    new UnSerializeDate();
  }
}
```

## *Exercise: Getting Acquainted With I/O*



**Exercise objective –** In this lab you will become familiar with stream I/O by writing programs which perform I/O to files.

### *Preparation*

You should understand the basic concepts of a database and the basic concepts of writing data to a stream.

### *Tasks*

#### *Level 1 Lab: Open a File*

Complete these steps:

1. Create a Java application called `DisplayFile.java` that will open, read, and display the contents of any readable file.

2. Have your application display an appropriate error message when it cannot display a file by including the appropriate exceptions.

#### *Level 2 Lab: Create a Simple Database Program*

Perform these steps:

1. Create an application called `DBTest.java` that emulates a small database program that stores and retrieves records relating to products. Use the `RandomAccessFile` class and a flat file.

   ▼ Records in the database should consist of String names and integer quantities.

   ▼ Your program should allow the user to display a record, update a record, and add a new record.

# Exercise: Getting Acquainted With I/O

## Tasks

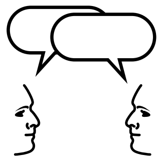
### Level 3 Lab :Use Persistence

Complete the following step:

1. Use the `Paint` program from Module 9 and save the state of the canvas to a file.

## Exercise: Getting Acquainted With I/O

### Exercise Summary



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Check Your Progress

Before continuing on to the next module, check to be sure that you can

- ❑ Describe and use streams philosophy of the `java.io` package
- ❑ Construct file and filter streams, and use them appropriately
- ❑ Distinguish readers and writers from streams, and select appropriately between them
- ❑ Examine and manipulate files and directories
- ❑ Read, write, and update text and data files
- ❑ Use the Serialization interface to persist the state of objects

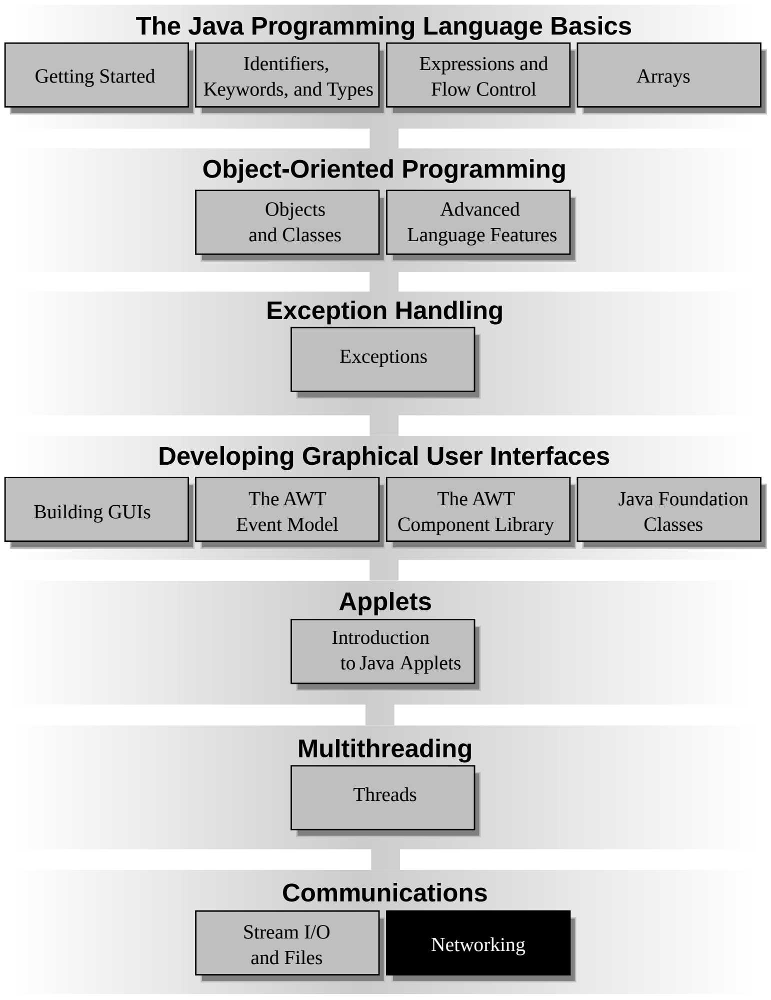
## Think Beyond

Do you have applications that require file I/O?

# *Networking* 15

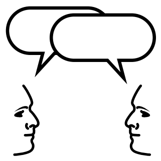
## *Course Map*

This module discusses JDK support for sockets and socket programming. Socket programming is used to communicate with other programs running on computers on the same network.


The Java Programming Language Basics
Getting Started
Identifiers, Keywords, and Types
Expressions and Flow Control
Arrays
Object-Oriented Programming
Objects and Classes
Advanced Language Features
Exception Handling
Exceptions
Developing Graphical User Interfaces
Building GUIs
The AWT Event Model
The AWT Component Library
Java Foundation Classes
Applets
Introduction to Java Applets
Multithreading
Threads
Communications
Stream I/O and Files
Networking

## Relevance

✓ ***Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.***



**Discussion –** The following question is relevant to the material presented in this module:

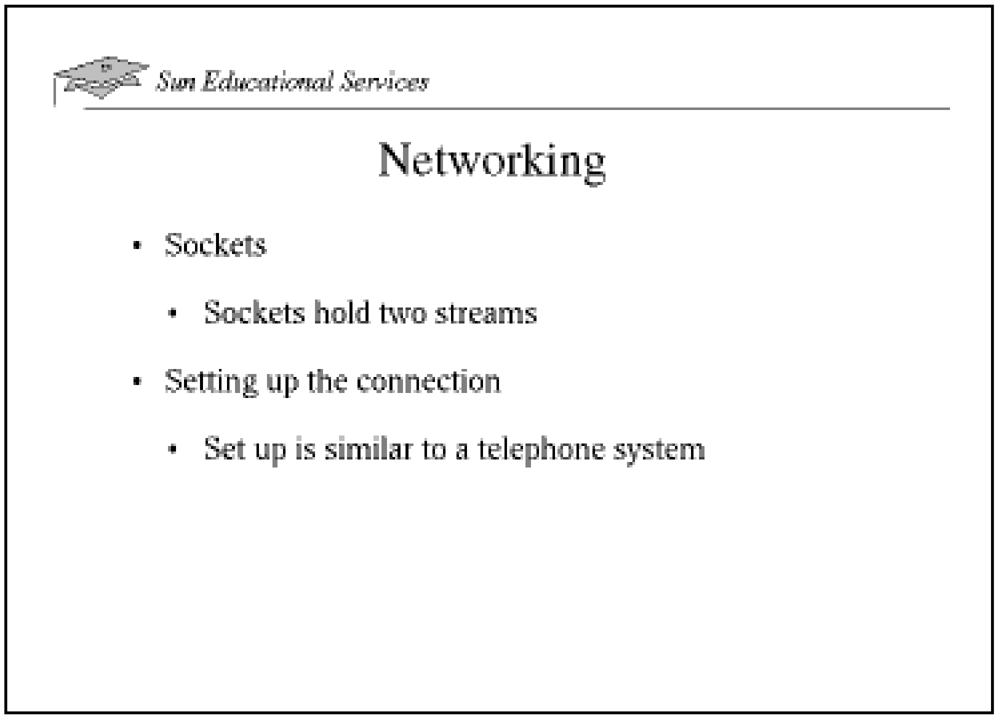
- How can a communication link between a client machine and a server on the network be established?

✓ ***Networked servers are a convenient way of providing services to any client on the same network who needs them. Sockets provide a means of network communication to make this possible. The JDK provides an interface to sockets through classes in the*** `java.net` ***package. This module examines two kinds of socket programming offered by the JDK, from both a server and client perspective.***

## Objectives

At the end of this module, you should be able to

- Develop code to set up network connection
- Understand TCP/IP and UDP protocol
- Use `ServerSocket` and `Socket` classes for implementing TCP/IP client and servers
- Use `DatagramPacket` and `DatagramSocket` classes for creating effecting a UDP-based network communication



Sun Educational Services

## Networking

- Sockets
    - Sockets hold two streams
- Setting up the connection
    - Set up is similar to a telephone system

## *Networking*

### *Sockets*

*Socket* is the name given, in one particular programming model, to the endpoints of a communication link between processes. Because of the popularity of that particular programming model, the name socket has been reused in other areas, including Java technology.

When processes communicate over a network, Java technology uses its streams model. A socket can hold two streams: one input stream and one output stream. A process sends data to another process through the network by writing to the output stream associated with the socket. A process reads data written by another process by reading from the input stream associated with the socket.

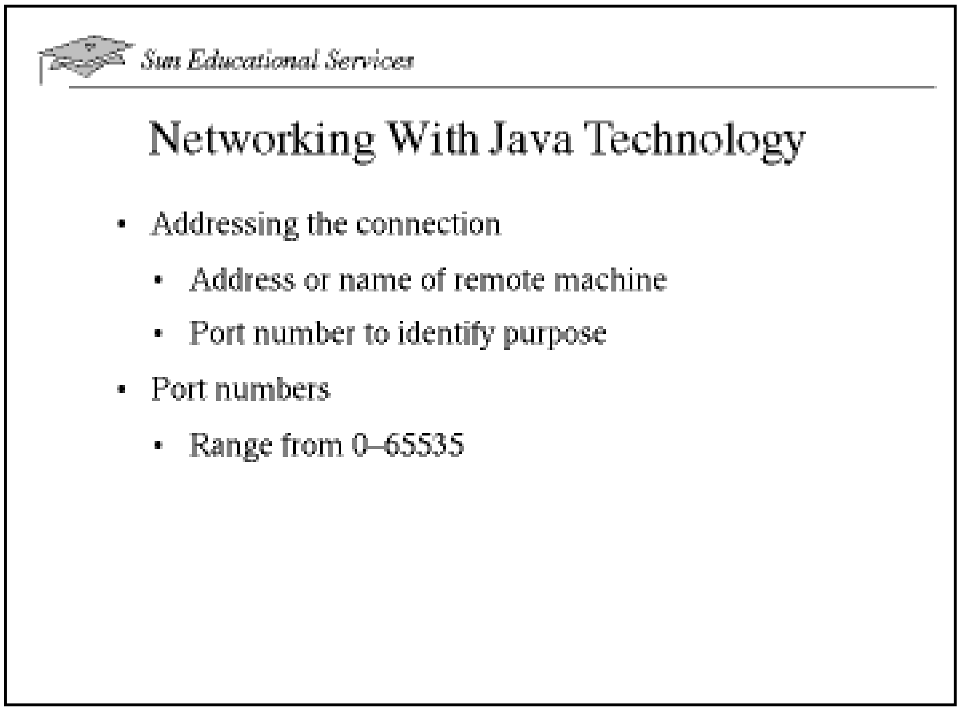
Once the network connection has been set up, using the streams associated with that connection is very similar to using any other stream.

# Networking

## Setting up the Connection

To set up the connection, one machine must be running a program that is waiting for a connection, and the other machine must try to reach the first. This is similar to a telephone system; one party must make the call, while the other party must be waiting by the telephone when that call is made.

Transmission Control Protocol/Internet Protocol, or *TCP/IP*, is the first type of connection protocol that is presented in this module.



Sun Educational Services

## Networking With Java Technology

- Addressing the connection
  - Address or name of remote machine
  - Port number to identify purpose
- Port numbers
  - Range from 0–65535

## *Networking With Java Technology*

### *Addressing the Connection*

When making a telephone call, you need to know the telephone number to dial. When making a network connection, you need to know the address or the name of the remote machine. In addition, a network connection requires a port number which you can think of as a telephone extension number. Once you have connected to the proper computer, you need to identify a particular purpose for the connection. So, in the same way that you can use a particular telephone extension number to talk to the accounts department, you can use a particular port number to communicate with the accounting program.

## Networking With Java Technology

### Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and range from 0–65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless you want to communicate with one of those services (such as `telnet`, Simple Mail Transport Protocol (SMTP) mail, `ftp`, and so forth).

Both client and server must agree in advance on which port to use. If the port numbers used by the two parts of the system do not agree, no communication will occur.

# Networking With Java Technology

## Java Networking Model

In the Java programming language, TCP/IP socket connections are implemented with classes in the `java.net` package. Below is a diagram of what occurs on the server side and the client side.

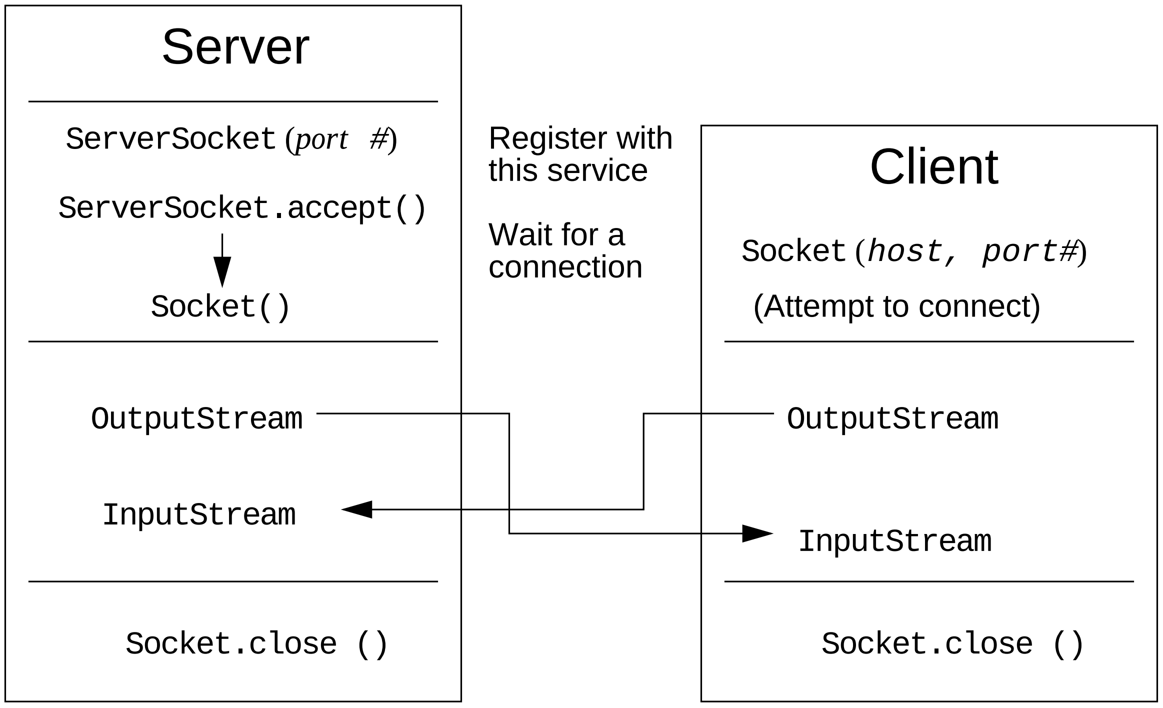


**Figure 15-1** TCP/IP Socket Connections

In this illustration,

- The server assigns a port number. When the client requests a connection, the server opens the socket connection with the `accept()` method.
- The client establishes a connection with *`host`* on port *`port#`*.
- Both the client and server communicate by using an `InputStream` and an `OutputStream`.

## Minimal TCP/IP Server

TCP/IP server applications rely on the ServerSocket and Socket networking classes provided by Java programming language. The ServerSocket class takes most of the work out of establishing a server.

```
import java.net.*;
import java.io.*;

public class SimpleServer {
  public static void main(String args[]) {
    ServerSocket s = null;
    Socket s1;
    String sendString = "Hello Net World!";
    int slength = sendString.length();
    OutputStream s1out;
    DataOutputStream dos;

    // Register your service on port 5432
    try {
      s = new ServerSocket(5432);
    } catch (IOException e) {
      // ignore
    }

    // Run the listen/accept loop forever
    while (true) {
      try {
        // Wait here and listen for a connection
        s1=s.accept();

        // Get a communication stream associated with
        // the socket
        s1out = s1.getOutputStream();
        dos = new DataOutputStream (s1out);

        // Send your string!
        // (UTF provides machine independence)
        dos.writeUTF(sendString);

```

## Minimal TCP/IP Server

```
          // Close the connection, but not the server socket
          dos.close();
          s1out.close();
          s1.close();
        } catch (IOException e) {
          // ignore
        }
      }
    }
  }
```

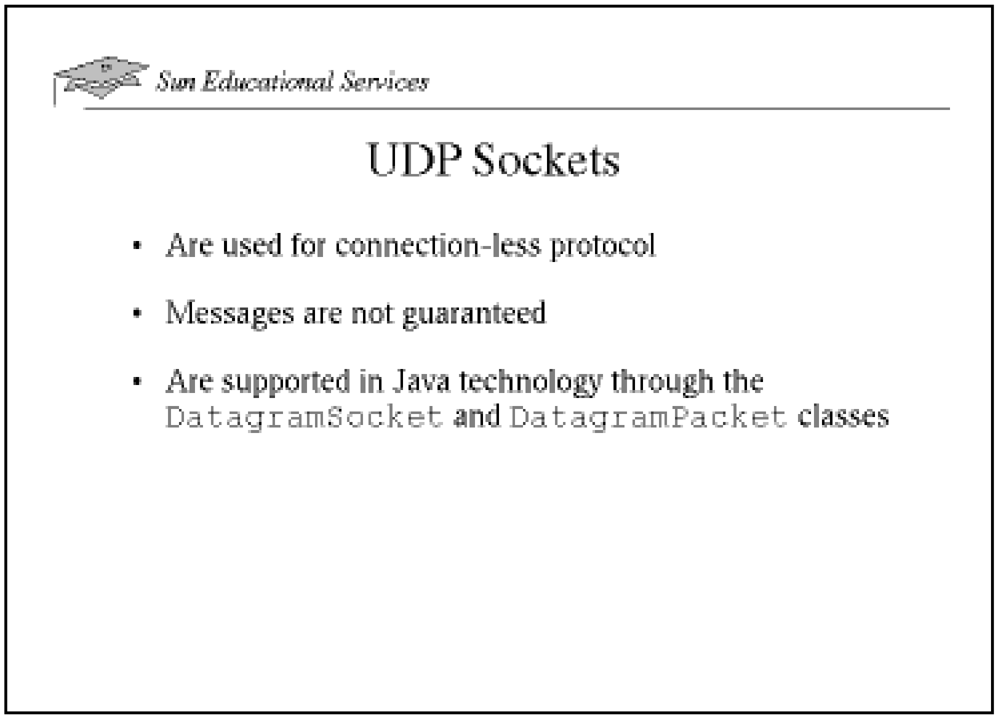
✓ ***If you use*** `writeUTF`***, you*** *must* ***use*** `readUTF` ***on the other end of the connection.***

## Minimal TCP/IP Client

The client side of a TCP/IP application relies on the `Socket` class. Again, much of the work involved in establishing connections has been done by the `Socket` class. The client attaches to the server presented on the previous page and prints everything sent by the server to the console.

```
import java.net.*;
import java.io.*;

public class SimpleClient {

  public static void main(String args[])
      throws IOException {
    int c;
    Socket s1;
    InputStream s1In;
    DataInputStream dis;

    // Open your connection to a server, at port 5432
    // localhost used here
    s1 = new Socket("127.0.0.1",5432);

    // Get an input file handle from the socket and
    // read the input
    s1In = s1.getInputStream();
    dis = new DataInputStream(s1In);

    String st = new String (dis.readUTF());
    System.out.println(st);

    // When done, just close the connection and exit
    dis.close();
    s1In.close();
    s1.close();
  }
}
```

✓ ***Currently the Windows 95/NT*** **`Socket close()`** ***method does not correctly send an EOF (end-of-file) message to the connected socket. A temporary fix is to send an ending close message such as "bye."***



Sun Educational Services

## UDP Sockets

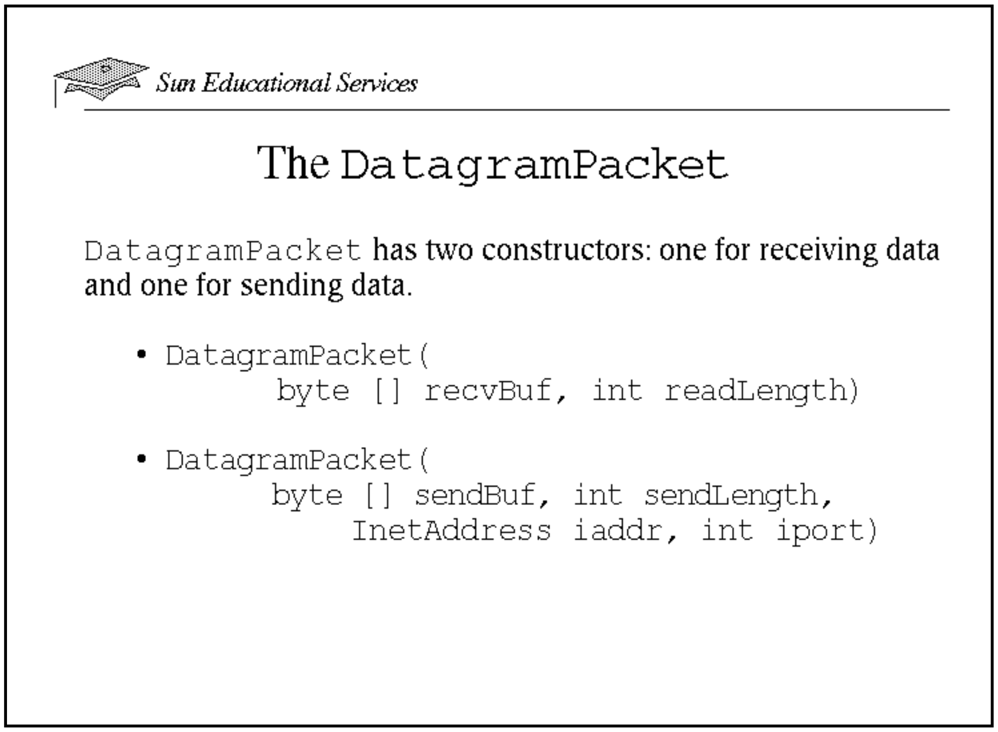
- Are used for connection-less protocol
- Messages are not guaranteed
- Are supported in Java technology through the `DatagramSocket` and `DatagramPacket` classes

## *UDP Sockets*

TCP/IP is a connection-oriented protocol. The User Datagram Protocol (UDP), on the other hand is a "connection-less" protocol. A simple yet elegant way to think of the differences between these two protocols is the difference between a telephone call and postal mail.

Telephone calls guarantee that there is synchronous communication; messages are sent and received in their intended order. A conversation using postcards sent via postal mail is not guaranteed—the messages might be received out of order, if received at all.

The User Datagram Protocol is supported through two classes: `DatagramSocket` and `DatagramPacket`. The packet is a self-contained message that includes information about the sender, the length of the message, and the message itself.



Sun Educational Services

## The DatagramPacket

DatagramPacket has two constructors: one for receiving data and one for sending data.

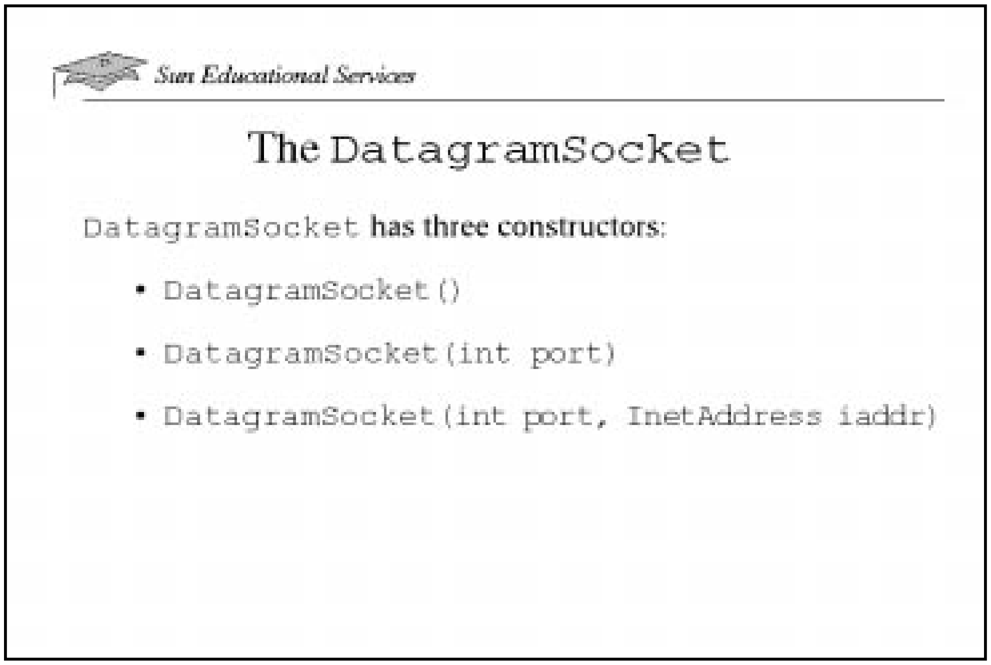
```
• DatagramPacket(
        byte [] recvBuf, int readLength)

• DatagramPacket(
        byte [] sendBuf, int sendLength,
            InetAddress iaddr, int iport)
```

## *UDP Sockets*

### DatagramPacket

`DatagramPacket` has two constructors: one for receiving data and one for sending data:

- `DatagramPacket (byte[] recvBuf, int readLength)` – Used to set up a byte array to receive a UDP packet. The `byte` array is empty when it is passed to the constructor and the `int` variable is set to the number of bytes to read (which is not greater than the size of the byte array).
- `DatagramPacket (byte[] sendBuf, int sendLength, InetAddress iaddr, int iport)` – Used to set up a UDP packet for transmission. The `sendLength` is not larger than the `sendBuf` byte array.



Sun Educational Services

## The DatagramSocket

DatagramSocket has three constructors:

- DatagramSocket()
- DatagramSocket(int port)
- DatagramSocket(int port, InetAddress iaddr)

## *UDP Sockets*

### *The* `DatagramSocket`

`DatagramSocket` is used to read and write UDP packets. This class has three constructors that allow you to specify which port and internet address to bind to:

- `DatagramSocket()` – Bind to any available port on the local host
- `DatagramSocket(int port)` – Bind to the specified port on the local host
- `DatagramSocket(int port, InetAddress iaddr)` – Bind to the specified port on the specified address

## Minimal UDP Server

This minimal UDP server listens to requests from the client on port 8000. When it receives a DatagramPacket from the client, it sends the current time on the server.

```
import java.io.*;
import java.net.*;
import java.util.*;

public class UdpServer{

  //This method retrieves the current time on the server
  public byte[] getTime(){
    Date d= new Date();
    return d.toString().getBytes();
  }

  // Main server loop.
  public void go() throws IOException {

    DatagramSocket datagramSocket;
    // Datagram packet from the client
    DatagramPacket inDataPacket;
    // Datagram packet to the client
    DatagramPacket outDataPacket;
    // Client return address
    InetAddress clientAddress;
    // Client return port
    int clientPort;
    // Incoming data buffer.  Ignored.
    byte[] msg= new byte[10];
    // Stores retrieved time
    byte[] time;

    // Allocate a socket to man port 8000 for requests.
    datagramSocket = new DatagramSocket(8000);
    System.out.println("UDP server active on port 8000");

```

## Minimal UDP Server

```
    // Loop forever
    while(true) {

      // Set up receiver packet.  Data will be ignored.
      inDataPacket = new DatagramPacket(msg, msg.length);

      // Get the message.
      datagramSocket.receive(inDataPacket);

      // Retrieve return address information, including
      // InetAddress and port from the datagram packet
      // just recieved.

      clientAddress = inDataPacket.getAddress();
      clientPort = inDataPacket.getPort();

      // Get the current time.
      time = getTime();

      //set up a datagram to be sent to the client using the
      //current time, the client address and port
      outDataPacket =
          new DatagramPacket(
              time, time.length, clientAddress, clientPort);

      //finally send the packet
      datagramSocket.send(outDataPacket);
    }
  }

  public static void main(String args[]) {
    UdpServer udpServer = new UdpServer();

    try {
      udpServer.go();
    } catch (IOException e) {
      System.out.println(
          "IOException occured with socket.");
      System.out.println (e);
      System.exit(1);
    }
  }
}
```

## Minimal UDP Client

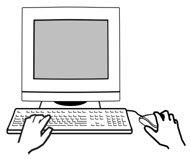
This minimal UDP client sends an empty packet to the server created previously and receives a packet containing the actual time on the server.

```
import java.io.*;
import java.net.*;

public class UdpClient {

  public void go()
      throws IOException,
      UnknownHostException {

    DatagramSocket datagramSocket;
    // Datagram packet to the server
    DatagramPacket outDataPacket;
    // Datagram packet from the server
    DatagramPacket inDataPacket;
    // Server host address
    InetAddress serverAddress;
    // Buffer space.
    byte[] msg = new byte[100];
    // Received message in String form.
    String receivedMsg;

    // Allocate a socket by which messages are sent
    // and received.
    datagramSocket = new DatagramSocket();

    // Server is running on this same machine for this
    // example.
    // This method can throw an UnknownHostException.
    serverAddress = InetAddress.getLocalHost();

    // Set up a datagram request to be sent to the server.
    // Send to port 8000.
    outDataPacket =
        new DatagramPacket(msg, 1, serverAddress, 8000);

    // Make the request to the server.
    datagramSocket.send(outDataPacket);

```

## Minimal UDP Client

```
    // Set up a datagram packet to receive
    // server's response.
    inDataPacket = new DatagramPacket(msg, msg.length);

    // Receive the time data from the server
    datagramSocket.receive(inDataPacket);

    // Print the data received from the server
    receivedMsg = new String(
      inDataPacket.getData(), 0, inDataPacket.getLength());
    System.out.println(receivedMsg);

    //close the socket
    datagramSocket.close();
  }

  public static void main(String args[]) {
    UdpClient udpClient = new UdpClient();

    try {
      udpClient.go();
    } catch (Exception e) {
      System.out.println ("Exception occured with socket.");
      System.out.println (e);
      System.exit(1);
    }
  }
}
```

## Exercise: Using Socket Programming



**Exercise objective –** Gain experience using sockets by implementing a client and server which communicate using sockets.

### Preparation

In order to successfully complete this lab, you must have a clear understanding of HTML and the network.

### Tasks

#### Level 1 Lab: Create Sockets

Work in pairs on this lab so you can use the host name of a machine other than your own. You are going to create a server and client pair and a program that requests a file from one of them.

1. The server template (`FileServer.java`) is in the `templates` directory. Develop the methods that will enable the server to receive a string file name from the client and attempt to open it, and pass it back to the client on the socket.

2. The client template (`ReadFile.java`) is also in the `templates` directory. The client program takes a string file name argument and sends it to the server, and then waits for the server to send either an error response or the file itself.

## Exercise: Using Socket Programming

### Tasks (Continued)

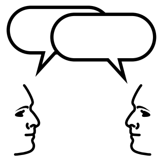
#### *Level 3 Lab: Create a Multithreaded Server (`MultiFileServer.java`)*

Complete the following steps:

1. Extend the client code to enable the client to request multiple files.
2. Extend the client to save the file to disk if an error is not returned.
3. Extend the server to use threads so that multiple clients can connect to the server at one time.

## *Exercise: Using Socket Programming*

### *Exercise Summary*



**Discussion –** Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ ***If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

✓ ***Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.***

- Interpretations

✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

✓ ***Have students articulate any conclusions they have reached as a result of this exercise experience.***

- Applications

✓ ***Explore with the students how they might apply what they learned in this exercise to situations at their workplace.***

## Notes

## Check Your Progress

Before continuing, check to be sure that you can

- ❑ Develop code to set up network connection
- ❑ Understand TCP/IP and UDP protocol
- ❑ Use `ServerSocket` and `Socket` classes for implementing TCP/IP client and servers
- ❑ Use `DatagramPacket` and `DatagramSocket` for effecting a UDP-based network communication

## Think Beyond

There are several advanced Java platform topics, many of which are addressed in other Sun Education courses. See Appendix A for a brief discourse on some of them. Be sure and check out the JavaSoft™ Web site (http://`www.javasoft.com`.) as well.

# *Elements of Advanced Java Programming* — *A*

## *Objectives*

At the end of this appendix, you should be able to

- Understand two-tier and three-tier architectures for distributed computing
- Understand the role of the Java programming language as a front end for database applications
- Use the JDBC API
- Understand data interchange methodologies using object brokers
- Explain the JavaBeans Component Model

## Introduction to Two-Tier and Three -Tier Architectures

*Client/server* computing involves two or more computers sharing tasks related to a complete application. Ideally, each computer is performing logic appropriate to its design and stated function.

The most widely used form of client/server implementation is a two-tier client/server. This involves a front-end client application communicating with a back-end database engine running on a separate computer. Client programs send SQL statements to the database server. The server returns the appropriate results, and the client is responsible for handling the data.

The basic two-tier client server model is used for applications which can run with many popular databases including Oracle, Sybase, and Informix.

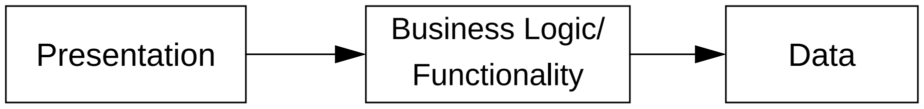
A major performance penalty is paid in two-tier client/server. The client software ends up larger and more complex as most of the logic is handled there. The use of server side logic is limited to database operations. The client here is referred to as *thick client*.

Thick clients tend to produce frequent network traffic for remote database access. This augurs well for Intranet and local srea networks (LAN)-based network topologies but produces a large footprint on the desktop in terms of disk and memory requirements. Also, not all back-end database servers are the same in terms of server logic offered, and all of them have their own API sets to optimize and scale performance that programmers must use. Three-tier client/server, which is described next, takes care of scalability, performance, and logic partitioning in a more efficient manner.

## The Three-Tier Architecture

Three-tier is the most advanced type of client/server software architecture. A three-tier client/server demands a much steeper development curve initially, especially when you have to support a number of different platforms and network environments. The payback comes in the form of reduced network traffic, excellent Internet and intranet performance, and more control over system expansion and growth.

### Three -Tier Client/Server Definition



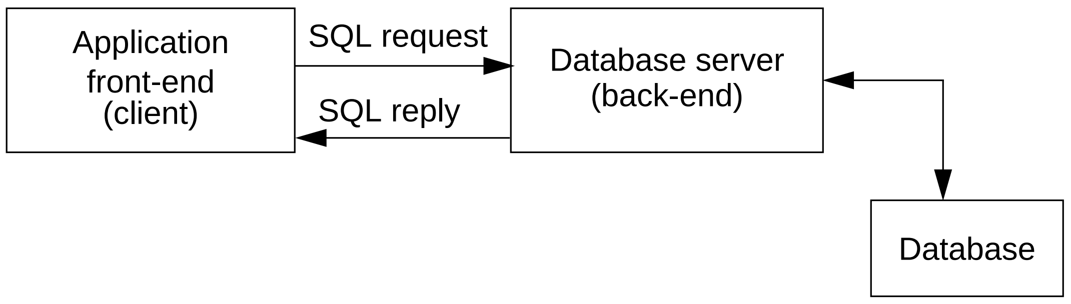
The three components or tiers of a three-tier client/server environment are *presentation*, *business logic* or *functionality*, and *data*. They are separated such that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers. For example, if you wanted to replace a character-oriented screen (or screens) with a GUI (the presentation tier), you would write the GUI using an established API or interface to access the same functionality program(s) in the character-oriented screen(s). The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having any impact on the actual databases. The third tier or data tier, includes existing systems, applications, and data that has been encapsulated to take advantage of this architecture with minimal transitional programming effort.

## A Database Frontend

The Java programming language offers wide benefits to software engineers creating front-end applications for database-oriented systems. With its "write once, run anywhere"SM language feature, the Java programming language offers immediate advantages in terms of its deployment on a wide range of hardware and operating systems. Programmers do not have to write platform-specific code for front-end applications, even in a multi-platform environment.

With Javca technology's rich set of supported front-end development classes, it is possible to interact with databases through the JDBC API. The JDBC API provides a connectivity to back-end databases which can be queried with results be handled by the frontend.

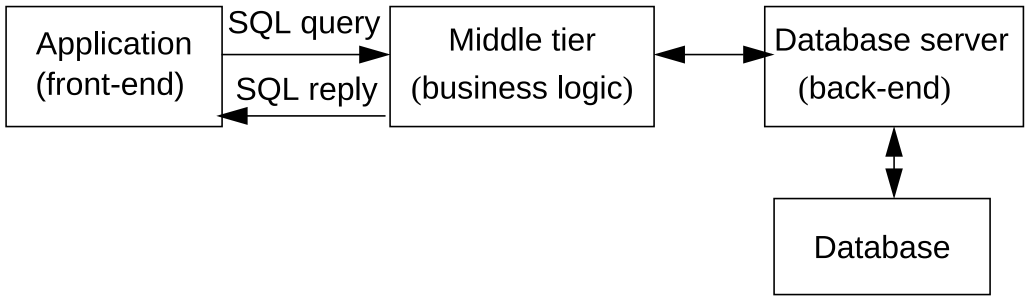
In a two-tier model, the database resides on a database server. The client executes a front-end application which opens a socket for communication over the network. The socket provides a communication path between the client application and the backend server. In the following illustration, client programs send SQL database query requests to the database server. The server returns the results to the client, which formats the results for presentation.



Frequently used mechanisms for data manipulations are often embedded as "stored procedures." Triggers automatically execute stored procedures when certain conditions are activated during the course of manipulations on the database. The primary drawback of this model is that all business rules are implemented in the client application, creating large client-side runtimes and increased rewriting of the client's code.

## A Database Frontend

In a three-tier model, all of the business rules are embedded in the client (front-end) tier. It communicates with an intermediate server which provides a layer of abstraction from the back-end applications. This middle tier manages the business rules which manipulate the data per the governing conditions of the applications. It can also accept connections from several clients to one or more database servers on a variety of communications protocols. The middle tier provides a database-independent interface for applications and makes the frontend robust. .



# Introduction to the JDBC API

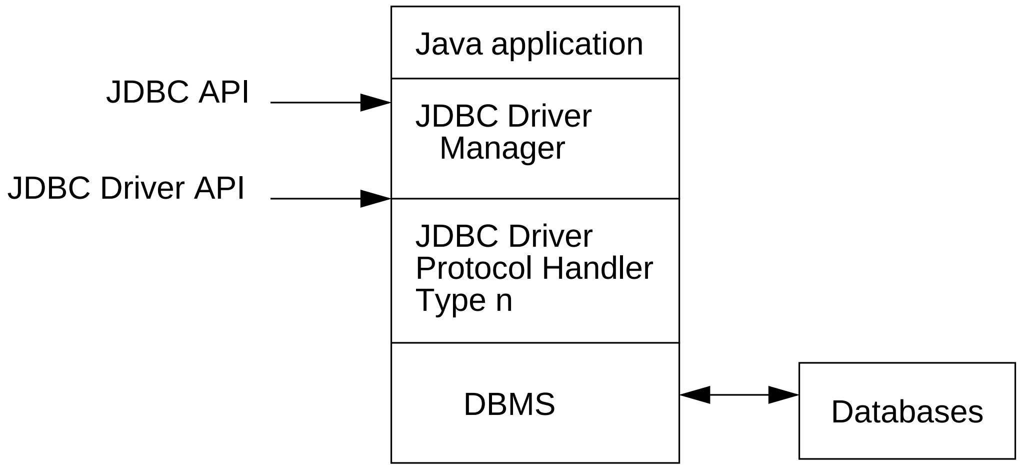
The ability to create robust, platform-independent applications and Web-based applets prompted developers to create front-end connectivity solutions. JavaSoft worked with database and database-tool vendors to create a database management system (DBMS)-independent mechanism that would enable developers to write client-side applications which worked with all databases. This effort resulted in the *Java Database Connectivity Application Programming Interface* (JDBC API).

## JDBC, An Overview

TheJDBC provides a standard interface for accessing a relational database. Modeled after the *open database connectivity* (ODBC) specification, the JDBC package contains a set of classes and methods for issuing SQL statements, table updates, and calls to stored procedures.

✓ ***Choosing ODBC was a pragmatic choice since it is a widely accepted and implemented standard for SQL database access. Virtually all databases support ODBC.***

As shown in the following figure, a Java programming language front-end application uses JDBC API to interact with JDBC Driver Manager. The JDBC Driver Manager uses the JDBC Driver API to load the appropriate JDBC driver. JDBC drivers, which are available from different database vendors, communicate with the underlying DBMS.



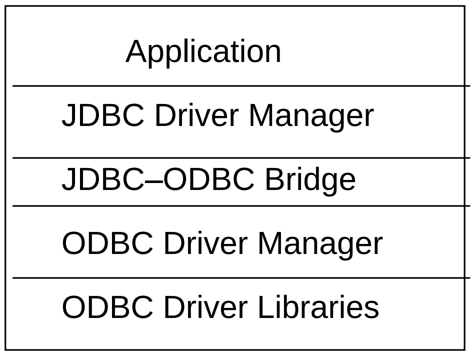
# Introduction to JDBC API

## JDBC Drivers

Java applications use the JDBC API to connect with a database through a database driver. Most database engines have different types of JDBC drivers associated with them. JavaSoft has defined four types of drivers. For more details refer to
`http://java.sun.com/products/jdbc/jdbc.drivers.html`

## The JDBC ODBC Bridge

The JDBC–ODBC bridge is a JDBC driver which translates JDBC calls to ODBC operations. This bridge enables all DBMS which support ODBC to interact with Java applications.



The JDBC–ODBC bridge interface is provided as a set of the C shared dynamic libraries. ODBC provides a client side set of libraries and a driver specific to the client's operating system. These ODBC calls are made as C calls and the client must have a local copy of the ODBC driver and associated client-side libraries. This places a restriction on its usage in web-based applications.

## Distributed Computing

There are Java technologies available for creating distributed computing environments. Two popular technologies are the *remote method invocation* (RMI) and the *common object request broker architecture* (CORBA). RMI is analogous to the *remote procedure call* (RPC) and preferred by programmers of the Java programming language. CORBA provides flexibility in heterogeneous development environments.

✓ ***For advanced coverage in distributed programming, recommend the SL-301 course.***

### RMI

The RMI feature enables a program running on a client computer to make method calls on an object located on a remote server machine. It gives a programmer the ability to distribute computing across a networked environment. Object-oriented design requires that every task be executed by the object most appropriate to that task. RMI takes this concept one step further by allowing a task to be performed on the machine most appropriate to the task. RMI defines a set of remote interfaces that can be used to create remote objects. A client can invoke methods of a remote object with the same syntax that it uses to invoke methods on a local object. The RMI API provides classes that handle all of the underlying communication and parameter referencing requirements of accessing remote methods.

With all of the distributed computing architectures, an application process or *object server (daemon)* advertises itself to the world by registering with a naming service on the local machine (node). In the case of RMI, a naming service daemon called the RMI registry runs over an RMI port which by default listens over IP port 1099 on that host. The RMI registry contains an internal table of remote object references. For each remote object, the table contains a registry name and a reference to that object. Multiple instances can be stored for the same object by instantiating and binding it multiple times to the registry, using different names.

## RMI

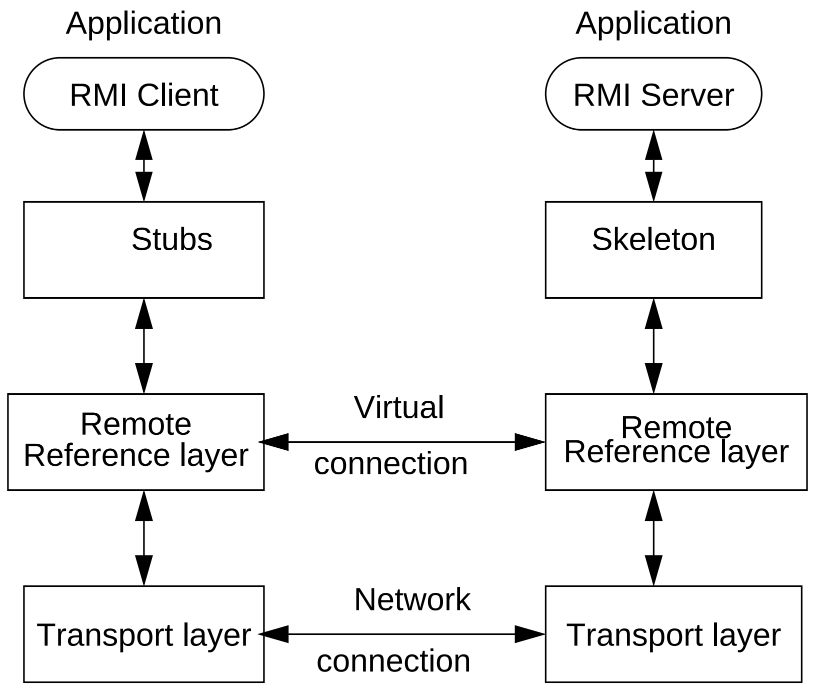
When an RMI client binds a remote object through the registry, it receives a local reference to the remote instantiated object through its interface and communicates with the object through that reference. Local references to the same remote object can exist on multiple clients; any variables and methods contained within the remote object are thus shared.

The applet begins by importing the appropriate RMI packages and creating a reference to the remote object. Once the applet establishes this link, it can call the remote object's methods as if they were locally available to the applet.

## RMI

### RMI Architecture

The RMI architecture provides three layers: Stubs/ Skeleton, Remote Reference and Transport layers.



The Transport layer creates and maintains physical connections between the client and server. It handles the data stream passing through the Remote/Reference layers (RRLs) on the client and server side.

The Remote Reference layer provides an independent reference protocol for establishing a virtual network between the client and server. It establishes interfaces to the lower Transport layer and the upper Stub/Skeleton layer.

A Stub is a client-side proxy representing the remote object. The client interacts with the Stub through interfaces. The Stub appears as a local object to the client. The Skeleton on the server side acts as an interface between the RRL and the object implemented on the server side.

## RMI

### Creating an RMI Application

This section guides you through the steps for creating , compiling and running an RMI application. The following steps illustrate the process:

- Define interfaces for remote classes.
- Create and compile implementation classes for the remote classes.
- Create stub and skeleton classes using the `rmic` command.
- Create and compile the server application.
- Start the RMI Registry and the server application
- Create and compile a client program to access the remote objects.
- Test the client.

## CORBA

CORBA is a *specification* that defines how distributed objects can interoperate. The CORBA specification is controlled by the Object Management Group (OMG), an open consortium of more than 700 companies that work together to define open standards for distributed computing. For more details refer to the following URL:

`http://www.omg.org`

CORBA objects can be written in any programming language, including C and C++. These objects can also exist on any platform, including Solaris, Windows 95/NT, openVMS, Digital UNIX, HP-UX and many others. This means, a Java application running on a Windows 95 platform can dynamically load and use C++ objects stored on the Internet by using a UNIX Web server.

Language independence is made possible via the construction of interfaces to objects using the *Interface Definition Language* (IDL). IDL allows all CORBA objects to be described in the same manner; the only requirement is a "bridge" between the native language (C/C++, COBOL, Java) and IDL.

At the core of CORBA is the *object resource broker* (ORB). The ORB is the principal component for the transmission of information between the client and the server of the CORBA application. The ORB manages marshalling requests, establishes a connection to the server, sends the data, and executes the requests on the server side. The same process occurs when the server wants to return the results of the operation. ORBs from different vendors communicate over TCP/IP using the *Internet Inter ORB Protocol (IIOP)*, which is a part of the CORBA 2.0 standard.

## The Java IDL

Java IDL adds CORBA capability to the Java programming language, providing standards-based interoperability and connectivity. Java IDL enables distributed Java web applications to transparently invoke operations on remote network services, using the industry standard IDL and IIOP.

Java IDL is not an *implementation* of OMG's IDL. It is, in fact, a CORBA ORB that uses IDL to define interfaces. The `idltojava` compiler generates portable client stubs and server skeletons. The CORBA client interacts with another object running on a remote server by accessing a reference object through its *naming service*. Like the RMI Registry, the naming service is an application that runs as a background process on a remote server. It holds a table of named services and remote object references used to resolve client requests.

The steps involved in setting up a CORBA object can be summarized as follows:

- Create the object's interface using the Interface Definition Language (IDL).
- Convert the interface into stub and skeleton objects using the `javatoidl` compiler.
- Implement the skeleton object, creating the CORBA server object.
- Compile and execute the server object, binding it to the naming service.
- Create and compile a client object, which invokes the methods within the server object.
- Execute the client object, accessing the server object through the CORBA naming service.

## RMI Versus CORBA

RMI's biggest advantage stems from the fact that it was designed to be a secure solution. This means that building RMI applications is simple, and all remote objects have the same features as local objects. It is also possible to combine the best of JDBC and RMI for a multi-tier solution.

CORBA, meanwhile, benefits from the fact that it is a language-independent solution, which adds significant complexity to the development cycle and precludes garbage collection features. For a database application developer, CORBA provides the ultimate flexibility in a heterogeneous environment. The server could be developed in C or C++, and the client could be a Java applet.

## The Java Beans Component Model

JavaBeans™ is an integration technology, a component framework that allows resuable component objects (called *Beans*) to communicate with one another and with the framework.

A Java Bean is an independent and reusable software component that can be manipulated visually in a builder tool. Beans can be visible objects, like AWT components, or invisible objects, like queues and stacks. A builder/integration tool manipulates Beans to create applets and applications. The component model specified by the JavaBeans 1.00-A specification defines five major services:

- Introspection

  This process exposes the properties, methods, and events that a JavaBean component supports. It is used at runtime and while the Bean is being created with a visual development tool.

- Communication

  This event-handling mechanism creates an event which serves as a message to other components.

- Persistence

  Persistence is a means of storing the state of a component. The simplest way to support persistence is to take advantage of Java object serialization, but it is up to the individual browsers or the applications that use the bean to actually save the state of the Bean.

- Properties

  Properties are attributes of a Bean that are referenced by name. These properties are usually read and written by calling methods on the Bean created specifically for that purpose. Some property types affect neighboring Beans as well as the one in which the property originates.

- Customization

  One of the primary characteristics of a Bean is its reusability. The Beans framework provides several ways of customizing existing Beans into new ones.

## Bean Architecture

A Bean is represented by an interface which is seen by the users. The environment must connect to this interface, if it wants to interact with this Bean. Beans consist of three general-purpose interfaces: Events, Properties, and Methods. Since Beans rely on their state, they must persistent over time.

### Events

Bean events are the mechanism for sending asynchronous messages between Beans, and between Beans and containers. A Bean uses an event to notify another Bean to take an action or to inform the Bean that a state change has occurred. An event allows your Beans to communicate when something interesting happens; to do this, they make use of the event model of JDK 1.1. There are three parts to this communication: EventObject, EventListener, and an Event Source.

## The Java Beans Component Model

### Bean Architecture (Continued)

JavaBeans communicate primarily using event listener interfaces that extend EventListener.

Bean developers can design their own event types and event listener interfaces and make their Beans act as a source by implementing the addXXXListener(EventObject e) and removeXXXListener(EventObject e) methods, where XXX is the name of the event type. Then, the developers can make other Beans act as event targets by implementing the XXXListener interface. The sourceBean and the targetBean are brought together by calling sourceBean.addXXXListener(targetBean).

#### *Properties*

Properties define the characteristics of the Bean. They can be changed at runtime through their `get` and `set` methods.

Properties can be used to send two-way synchronous communications between Beans. Beans also support asynchronous property changes between Beans via special event communication.

#### *Methods*

Methods are operations through which interaction with a Bean can be done. Beans receive notification of events by having the appropriate event source method call them. Some methods are special and deal with properties and events. These methods must follow special naming conventions outlined in the Beans specification. Other methods might be unrelated to an event or property. All public methods of a Bean are accessible to the Beans framework and can be used to connect a Bean to other Beans.

# The Java Beans Component Model

## Bean Introspection

The JavaBean introspection process exposes the properties, methods, and events of a Bean. Bean classes are assumed to have properties if there are methods that either set or get a property type.

The `BeanInfo` interface, provided by the Java Beans API, enables Bean designers to expose properties, events, methods, and any global information about a Bean. The `BeanInfo` interface provides a series of methods to access Bean information, but a Bean developer can also include private description files that the `BeanInfo` class uses to define Bean information. By default, a BeanInfo object is created when introspection is run on the Bean (Figure A-1).

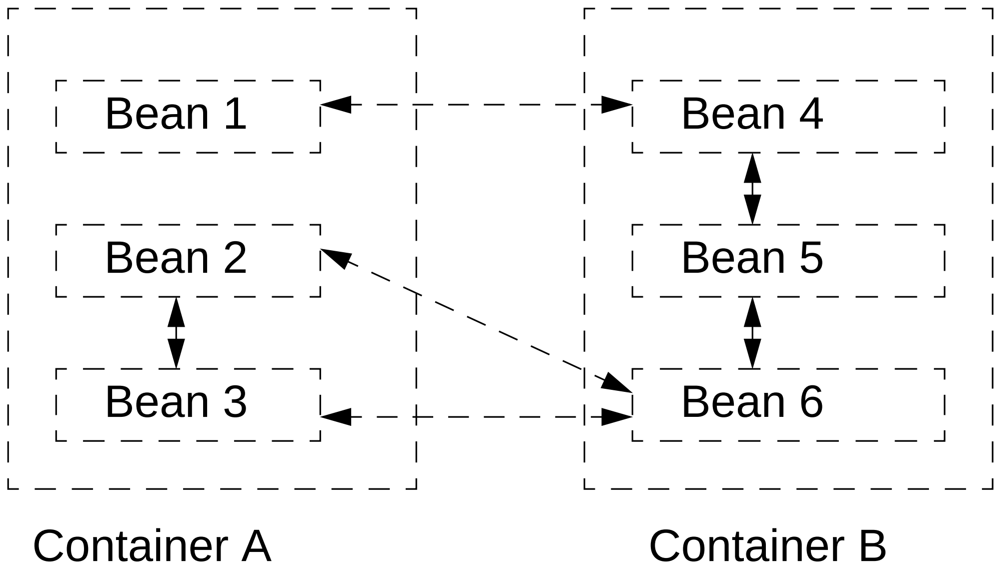


**Figure A-1** A Sample Bean Interaction

## The Java Beans Component Model

### A Sample Bean Interaction (Continued)

In Figure A-1, Container A and Container B contain six Beans. A Bean can interact with other Beans that are present in the same container and with Beans that are in a separate container. In this example, Bean 1 interacts with Bean 4. It does not communicate with Bean 2 and Bean 3 which reside in the same container. This illustrates the point that a Bean can communicate with any other Bean and is not restricted to communicating with a Bean in the same container. However, Bean 4 communicates with Bean 5 which resides in the same container. Source Bean 1 sends an event to the target Bean 4 which causes it to listen for messages on its event listener. All other inter- and intra-container Bean interactions can be explained in a similar manner.

### The Beans Development Kit (BDK)

The BDK is a Java application developed by JavaSoft that allows Java developers to create reusable components that use the Bean model. It is a complete system which contains source code for all examples and documentation. The BDK comes with a sample Bean builder and customizer application called BeanBox. The BeanBox is a test container that can be used to do the following:

- Resize and move Beans
- Alter Beans with property sheets
- Customize Beans with a customizer application
- Connect Beans together
- Drop Beans onto a composition window
- Save Beans through serialization
- Restore Beans

It comes with a set of 12 sample Beans, which cover all of the aspects of the Java Beans API.

✓ ***Recommend the SL-291 course.***

## JAR Files

JAR (Java Archive) is a platform-independent file format that aggregates many files into one. Multiple Java applets and their requisite components (.class files, images, and sounds) can be bundled in a JAR file and subsequently downloaded to a browser in a single Hypertext Transfer Protocol (HTTP) transaction, greatly improving the download speed. The JAR format also supports compression, which reduces the file size, further improving the download time. In addition, the applet author can digitally sign individual entries in a JAR file to authenticate their origin. It is fully backward-compatible with existing applet code and can be extended.

Changing the `applet` tag in your HTML page to accommodate a JAR file is easy. The JAR file on the server is identified by the ARCHIVE parameter, which contains the directory location of the jar file is relative to the location of the HTML page. For example:

```
<applet code=Animator.class
  archive="jars/animator.jar"
  width=460 height=160>
  <param name=foo value="bar">
</applet>
```

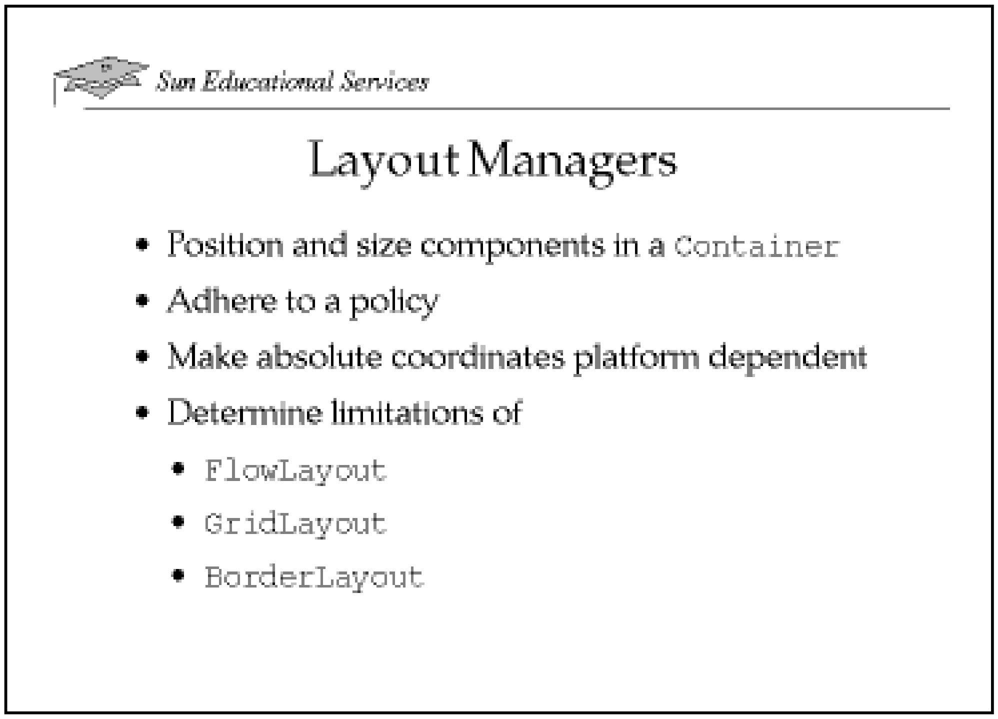
## Check Your Progress

At the end of this appendix, check to be sure that you can

- ❑ Identify advantages of multi-tier over two-tier architectures
- ❑ Understand the role of Java technology as a database frontend
- ❑ Use the JDBC API
- ❑ Understand the role of Java technology in distributed computing
- ❑ Explain the Java Bean Component Model

# *Using the* `GridBagLayout` *B*

This appendix discusses the use of the `GridBagLayout` in the production of complex user interfaces.



Sun Educational Services

## Layout Managers

- Position and size components in a `Container`
- Adhere to a policy
- Make absolute coordinates platform dependent
- Determine limitations of
  - `FlowLayout`
  - `GridLayout`
  - `BorderLayout`

## *Layout Managers*

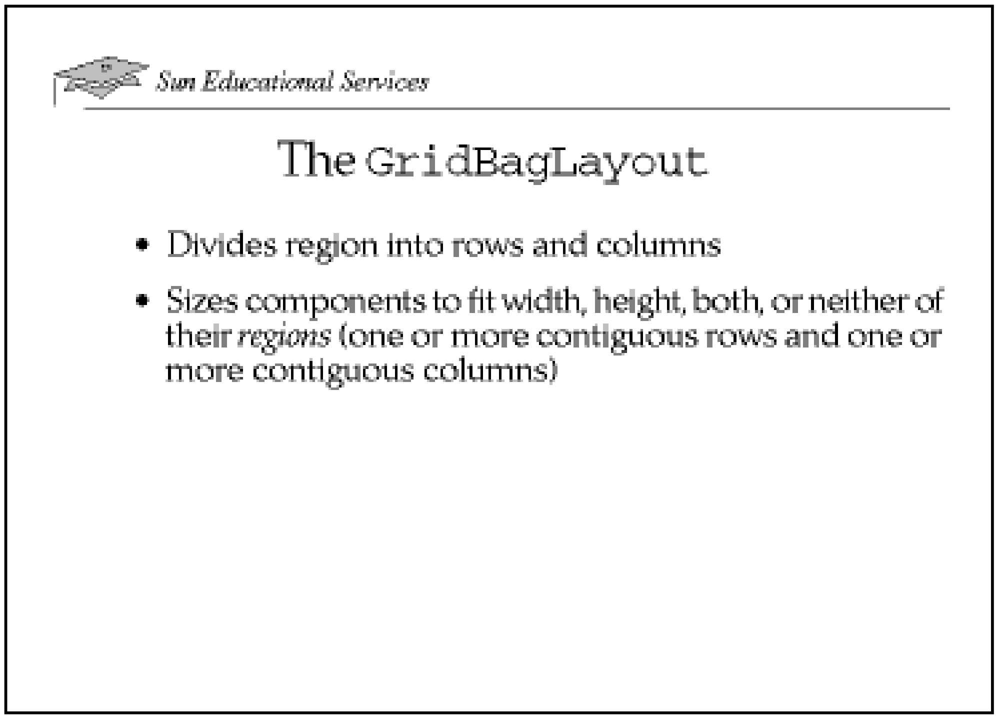
GUIs should make extensive use of layout managers, since the alternative, absolute positioning by pixel coordinates, is not platform portable. Issues such as the sizes of fonts and screens ensure that a layout that is correct and based on coordinates will be unusable on any other platform.

Layout managers avoid these difficulties by laying out the GUI according to a policy. For example the policy of the `GridLayout` is to position child components in equal-sized cells, starting at the top left and working left to right, top to bottom until the grid is full.

This course assumes you know about the basic three layout managers, `FlowLayout`, `GridLayout`, and `BorderLayout`. If you are unsure about any of these, ask your instructor if you can discuss them during a break.

## Layout Managers

If you do know the basic three layout managers, you will also know that they are somewhat limited in their capabilities, and that it can be hard, often involving many nested panels, to produce a layout that is useful in a production program. This appendix looks at the `GridBagLayout,` which is more powerful.



Sun Educational Services

### The GridBagLayout

- Divides region into rows and columns
- Sizes components to fit width, height, both, or neither of their *regions* (one or more contiguous rows and one or more contiguous columns)

## *The* GridBagLayout

The `GridBagLayout` lays out components using a grid. However, unlike the `GridLayout`, child components are not necessarily constrained to occupy exactly one entire grid cell, neither are all rows and columns equal in size. Rather, a component can be assigned multiple cells, horizontally, vertically, or both, and can exist within that region.

## *The* `GridBagLayout`

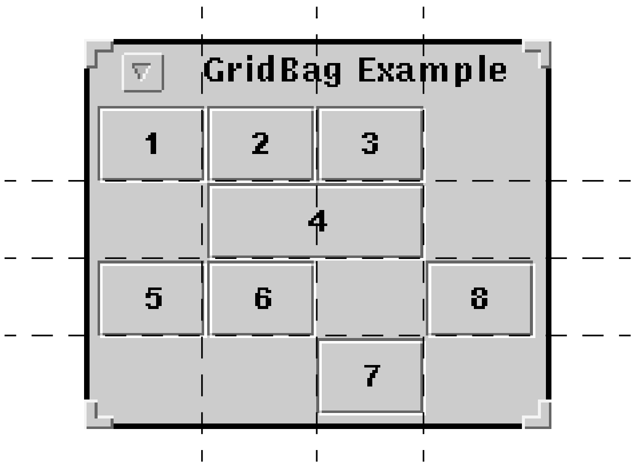


**Figure B-1** Sample `GridBagLayout` With Four Rows and Four Columns

The number of rows and columns in a `GridBagLayout` is determined by the number of cells that are in use. This contrasts with the `GridLayout` where (generally) you specify the row and column count at the time the layout is constructed.

The basic height of a row is determined by the largest component in that row. Similarly, the basic width of a column depends on the largest component in it. In Figure B-1, notice that each grid cell is the basic size of a `JButton` with a single-digit label.

## The `GridBagLayout`

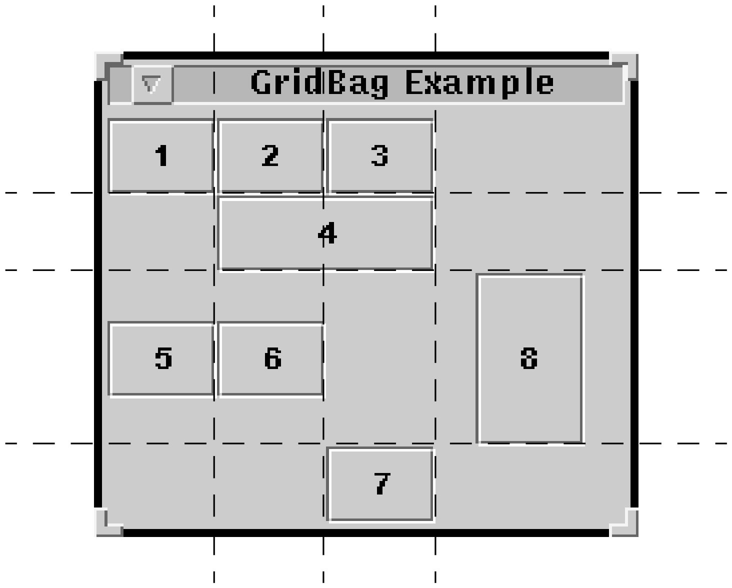


**Figure B-2** Sample `GridBagLayout` Showing Cells Expanded by Weight

Where the total space available to the `GridBagLayout` exceeds that needed for all the basic dimensions, the extra space is shared using a concept called *weight*. In Figure B-2, the weight has been applied to the last column and to the third row (that is, the row and column that includes the button labeled “8”).

A component in a `GridBagLayout` can occupy multiple consecutive rows, and multiple consecutive columns if desired. The total space alloted to one component is referred to as the component’s *region*. In Figure B-2, the button labelled “4” extends across two columns horizontally.

The size of a component in a `GridBagLayout` is not necessarily constrained to occupy the entire assigned region. Instead, the component can have its natural size, its natural height with the full width of its region, or its natural width with the full height of its region. Of course, it can also be constrained to fill the region. This property is known as the *fill* of a component. In Figure B-2, the buttons labelled “5” and “6” do not fill the vertical space available to them; similarly, the button labelled “8” fills vertically, but not horizontally.

## *The* GridBagLayout

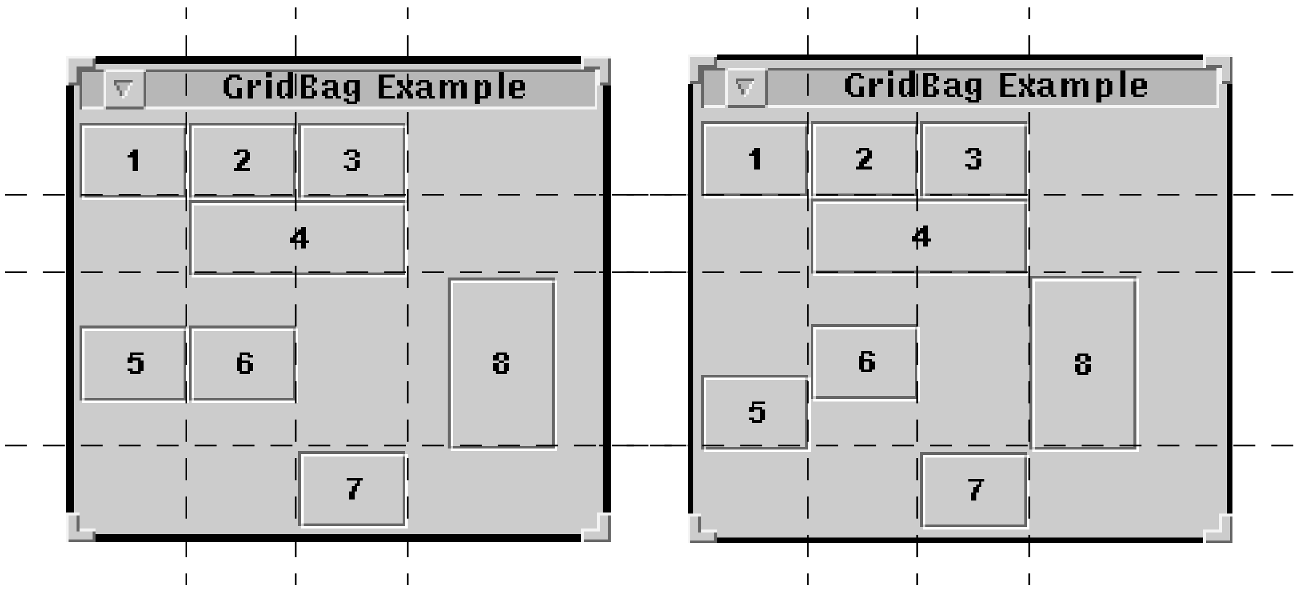


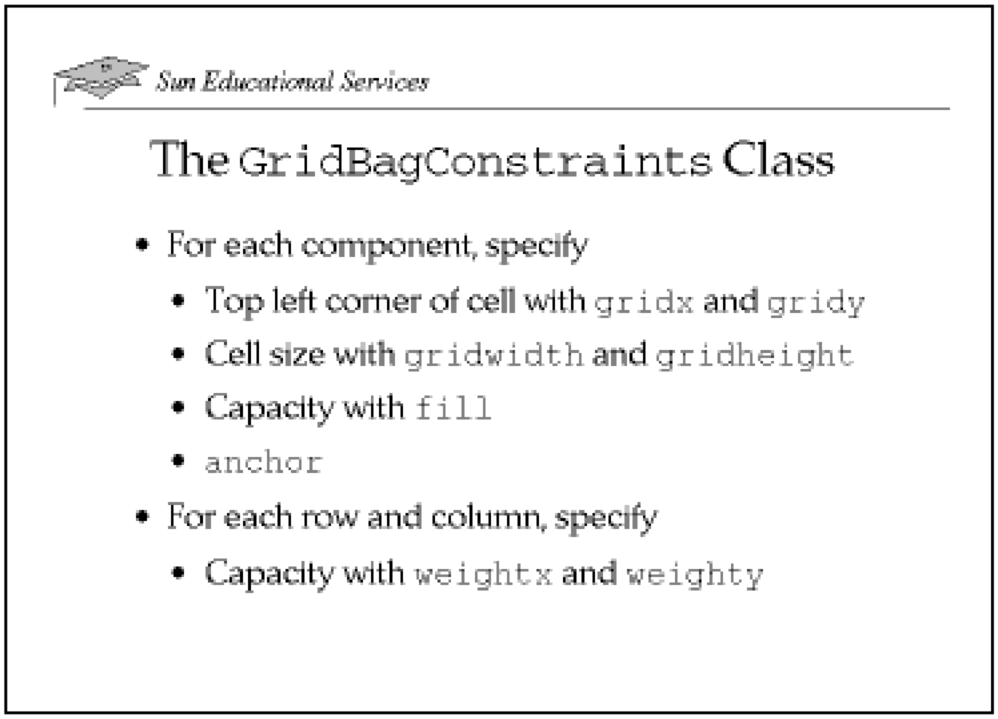
**Figure B-3** Sample GridBagLayout Showing the Effect of Anchor

Where a component does not fill the entire region allocated to it, its position within that region can be controlled using a concept called *anchor*. Anchor takes one of nine values. Eight of these values are compass points, NORTH, SOUTHWEST, and so on. The ninth is CENTER. If a component has its natural size and an anchor of NORTHWEST, then it will be positioned at the top left of its allocated region.

✓ ***In some systems (XMotif for example) the effect of anchor is referred to as*** *gravity.*

In Figure B-3, the two examples have differing anchor settings. Specifically, the button labelled "5" has a CENTER anchor in the left-hand example, but a SOUTH anchor in the right-hand example. The button labelled "8" has a CENTER anchor in the left-hand example, but a WEST anchor in the right-hand example.

Clearly, there is some interaction between anchor and the fill of a component. If the fill specifies that the component occupies the entire region alloted to it, then anchor has no significance. If the fill value specifies that a component occupies the allocated region entirely in the horizontal direction, then the only anchor values that are useful are NORTH, CENTER, and SOUTH



Sun Educational Services

The GridBagConstraints Class

- For each component, specify
  - Top left corner of cell with gridx and gridy
  - Cell size with gridwidth and gridheight
  - Capacity with fill
  - anchor
- For each row and column, specify
  - Capacity with weightx and weighty

## *The* GridBagConstraints *Class*

You have seen the principles by which the GridBagLayout makes positioning decisions, but not how those preferences are supplied to it. This is done using an object of the class GridBagConstraints. Each time you add a Component to a Container that has a GridBagLayout, you provide an instance of GridBagConstraints that contains the values needed to describe the layout of that Component.

The most significant fields of the GridBagConstraints object are:

- gridx and gridy. These integer fields are used to specify the row and column numbers at the top left of the component's region. They are effectively the component's coordinates.

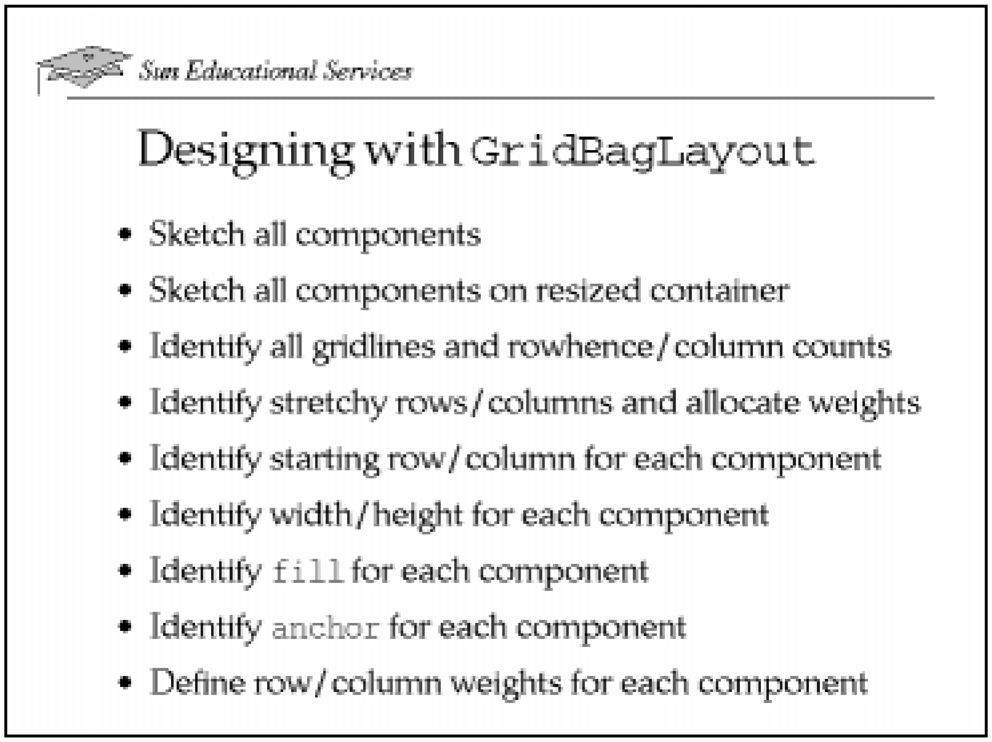
- gridwidth and gridheight. These integer fields describe the number of columns and rows, respectively, over which the component's region extends.

- fill. This field indicates how the component is sized within its region. Values for this field are constants in the GridBagConstraints class. The four symbolic values are: NONE, HORIZONTAL, VERTICAL, and BOTH.

## *The* `GridBagConstraints` *Class*

- `anchor`. This field indicates the anchor applied to the component. Values are constants in the `GridBagConstraints` class. The nine symbolic values are: `NORTH`, `SOUTH`, `EAST`, `WEST`, `NORTHEAST`, `NORTHWEST`, `SOUTHEAST`, `SOUTHWEST`, and `CENTER`.
- `weightx` and `weighty`. These fields are somewhat unusual in that they apply to the column and row to which the component is being added, not the component itself. The weight values are used to distribute "spare" space when the layout has more screen area available to it than it needs. The actual values of weight are significant only in a relative sense. That is, it doesn't matter if a particular value is 5 or 0.5. What matters is the proportion of the weight allocated to the sum of all weights allocated.

**Note –** Avoid setting weights on the same row or column for more than one component. Doing so will confuse anyone reading the program.



Sun Educational Services

## Designing with GridBagLayout

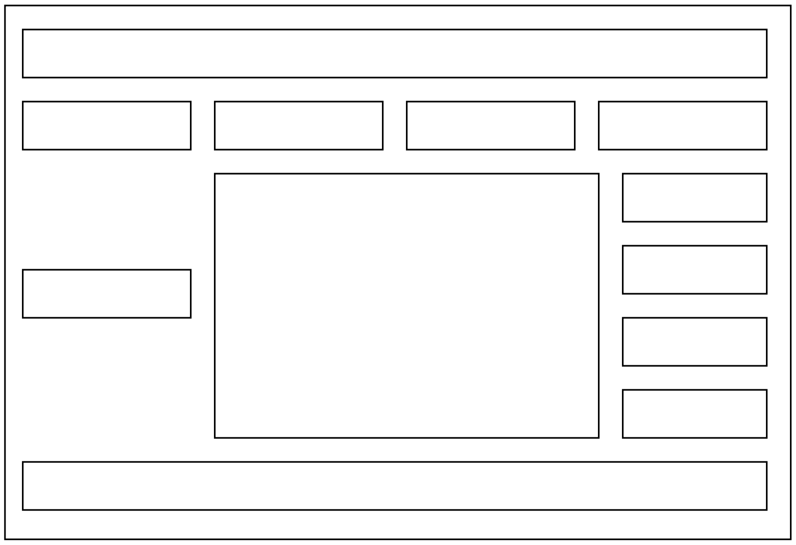
- Sketch all components
- Sketch all components on resized container
- Identify all gridlines and rowhence/column counts
- Identify stretchy rows/columns and allocate weights
- Identify starting row/column for each component
- Identify width/height for each component
- Identify `fill` for each component
- Identify `anchor` for each component
- Define row/column weights for each component

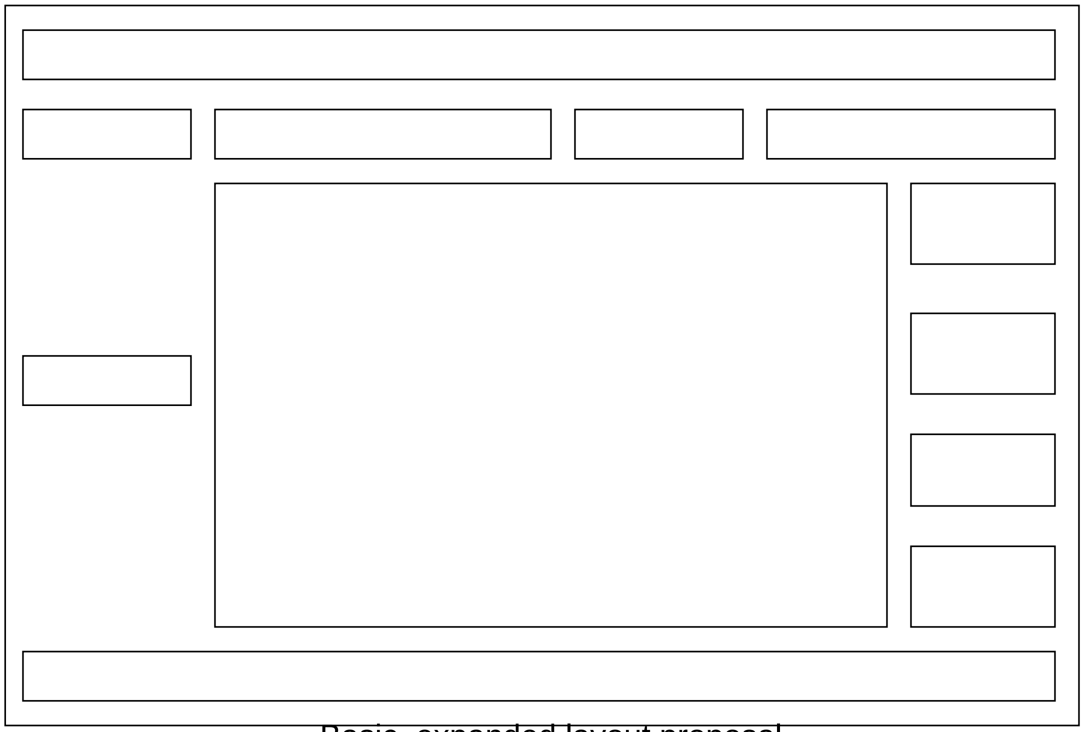
## *Designing With* `GridBagLayout`

### *Design Steps*

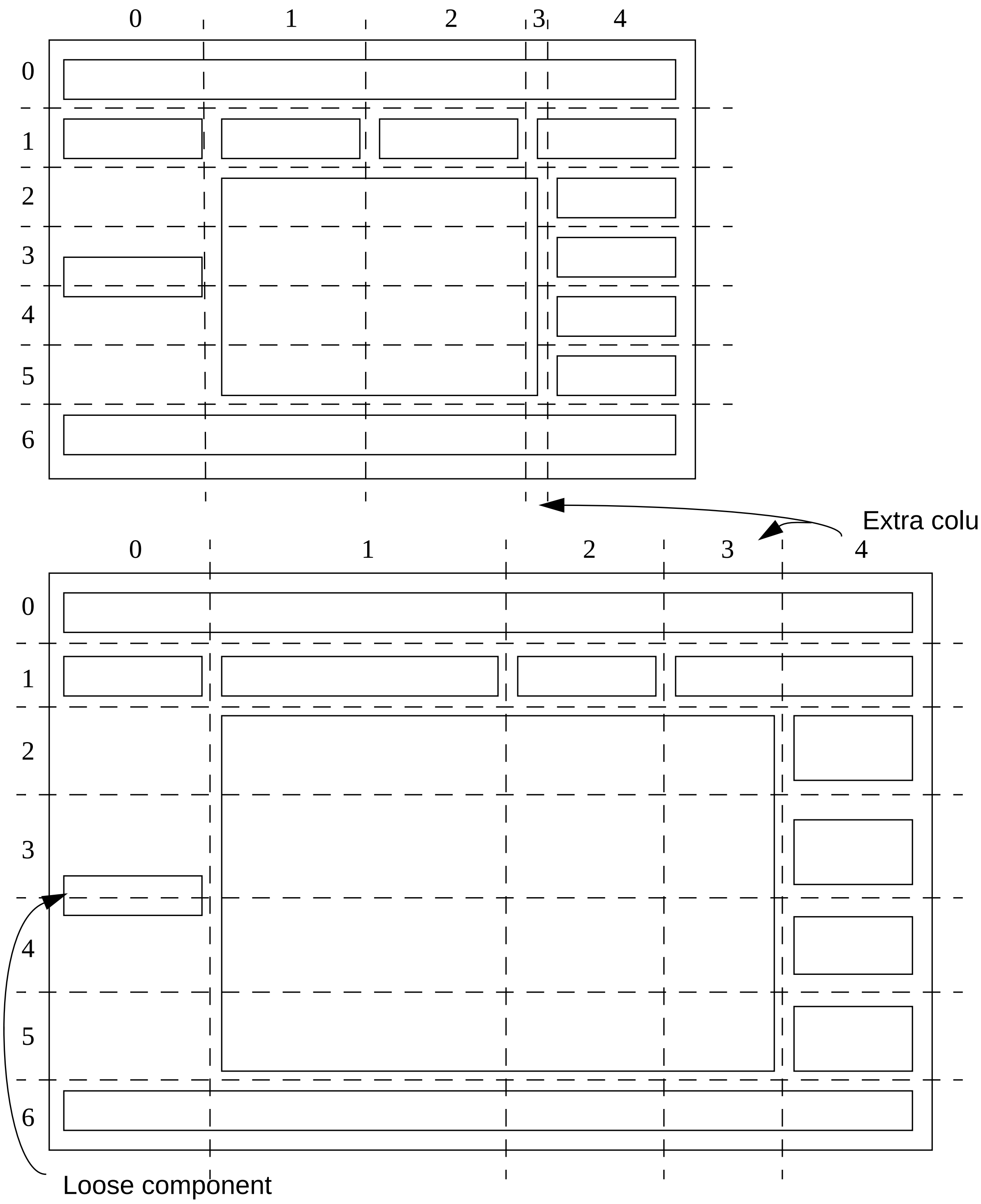
When designing with a `GridBagLayout`,

1. Sketch the components as you want them to appear.
2. Make another sketch with the window enlarged, and plan how you want the extra space to be allocated.


Basic, unexpanded layout proposal


Basic, expanded layout proposal

3. Identify the gridlines based on the edges of components in your pictures. Be particularly careful if your diagram shows two component edges in nearly the same alignment—did you mean them to be aligned? When you have identified the gridlines on one drawing, do this again on the second sketch.



4. Decide how the extra space is to be allocated. In some cases, it might be easiest to do this in terms of percentages. Once you have determined your percentages, you can use them as `weightx` and `weighty` values directly (even if they do not finally add up to 100).

The expanded version brings out the existence of an extra column, which is not really noticeable until the display is expanded. You would be unlikely to recognize this column's existence in the unexpanded diagram.

The "loose component" in column 0, third row down, does not match any of the grid cell boundaries. Rather, it appears to overlap rows 4 and 5. The component actually is located in a region that extends over rows 3 through to 6 inclusive, and is vertically centered in that region.

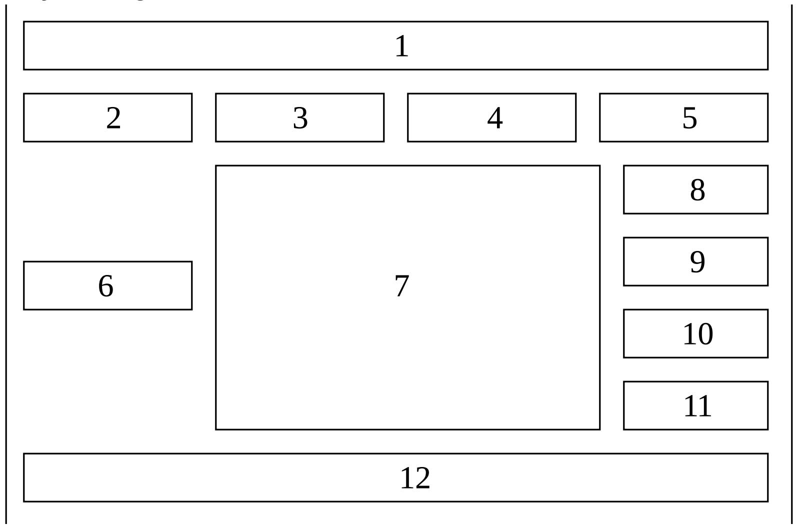
Columns 0, 2, and 4 do not change size, but columns 1 and 3 do. It is not entirely clear how the space is shared, but a reasonable working guess is that new space is allocated equally between them.

Rows 0, 1, and 6 do not change size, but rows 2 through 5 all stretch equally.

# *Designing With* `GridBagLayout`

## *Design Steps (Continued)*

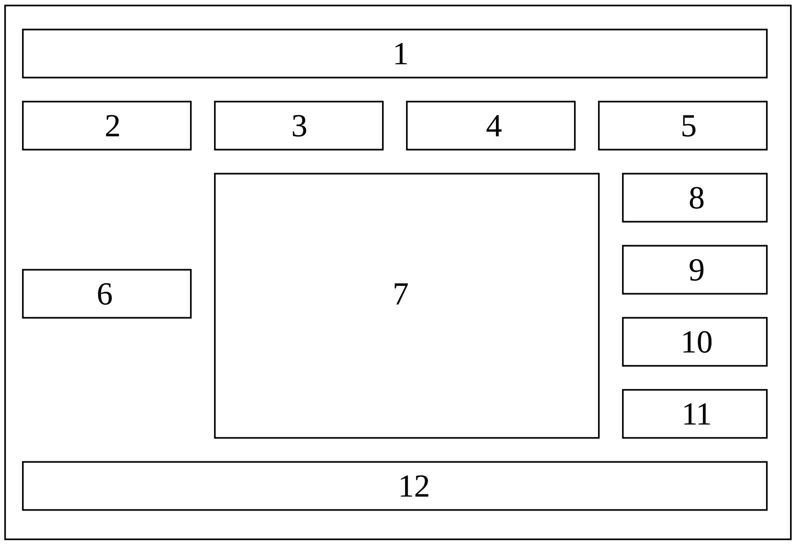
5. Now that you have designed the underlying grid, you can start to position each component over that grid. Start by identifying the top left row and column for each component region; this gives you the `gridy` and `gridx` values for each.



6. Determine the width and height of the region in terms of columns and rows; these are the `gridwidth` and `gridheight` values.

| **Component** | `gridx` | `gridy` | `gridwidth` | `gridheight` |
|---:|---:|---:|---:|---:|
| 1 | 0 | 0 | 5 | 1 |
| 2 | 0 | 1 | 1 | 1 |
| 3 | 1 | 1 | 1 | 1 |
| 4 | 2 | 1 | 1 | 1 |
| 5 | 3 | 1 | 2 | 1 |
| 6 | 0 | 2 | 1 | 4 |
| 7 | 1 | 2 | 3 | 4 |
| 8 | 4 | 2 | 1 | 1 |
| 9 | 4 | 3 | 1 | 1 |
| 10 | 4 | 4 | 1 | 1 |
| 11 | 4 | 5 | 1 | 1 |
| 12 | 0 | 6 | 5 | 1 |

7. For each component, consider how it occupies the region allocated to it. If it fills the region entirely, it has a `fill` value of `BOTH`. If it fills the region from side to side but not vertically, then its `fill` value is `HORIZONTAL`. If it fills its region vertically but not horizontally, then its `fill` value is `VERTICAL`. If it does not fill the region in either direction, then its fill value is `NONE`.



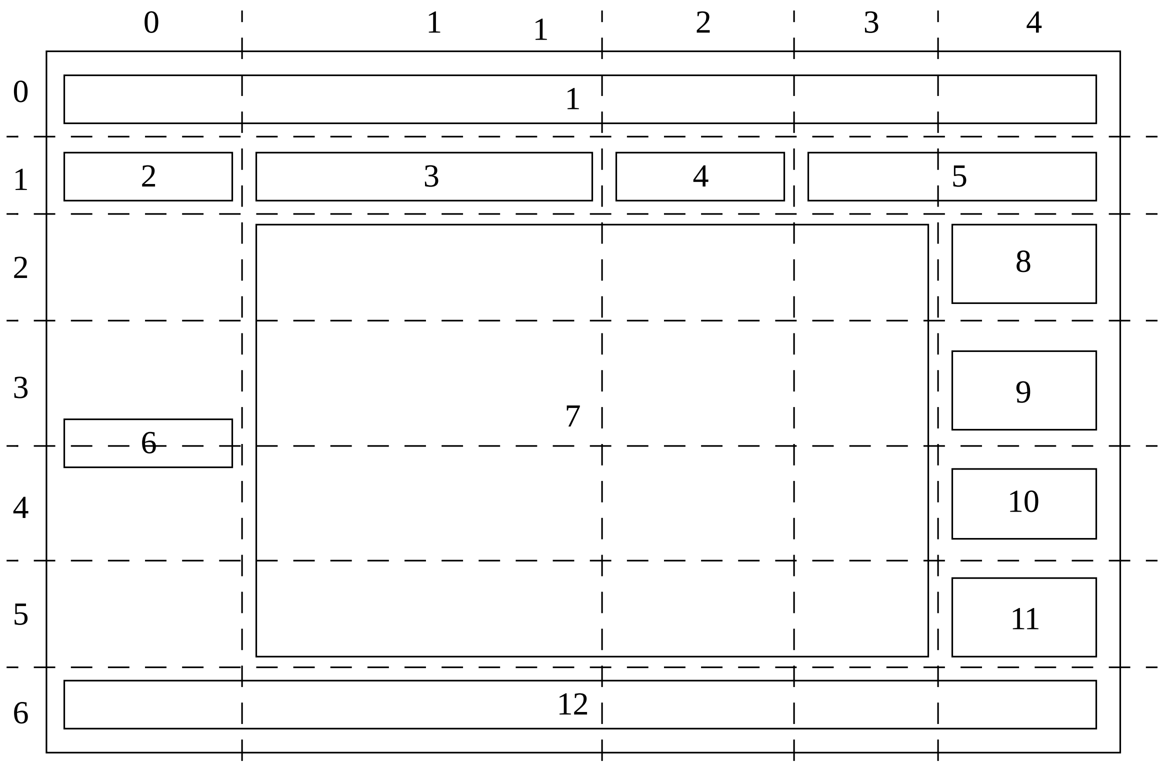
The fill value should be `BOTH` for all components that take the full size of their available regions. This is important even if the region does not stretch. For example, the cells occupied by components 8, 9, 10, and 11 do not stretch horizontally, so you might think that a horizontal component of fill was unnecessary. However, if you only specify a fill of `VERTICAL`, you will find that the components are given their preferred sizes, and since their labels are shorter, components 8 and 9 will be slightly smaller than components 10 and 11.

So, in this example, the only component that is not set to fill `BOTH` is component 6. This should have a fill value of `HORIZONTAL`, to ensure that it takes up the full width of its region.

8. For each component, consider how it is positioned within the region allocated to it and hence the `anchor` value for the component. If a component has a `fill` value of `BOTH` then the `anchor` value is irrelevant. Components with `HORIZONTAL fill` should have an `anchor` of `NORTH`, `CENTER`, or `SOUTH`. Components with `VERTICAL fill` should be anchored `WEST`, `CENTER`, or `EAST`. Components with a `fill` of `NONE`, can have an `anchor` of any of the nine values.

Anchor values are only significant where a component's region is larger than the component itself. In this example, this applies only to component 6. Here the component must be centered vertically, although it fills the available width. In consequence, any of the anchor values `EAST`, `WEST`, or `CENTER` would result in the required behavior, but `CENTER` is probably the most reasonable since it most directly expresses the required result.

9. Add the components and allocate the weights to the rows and columns. Choose one component for each row and one component for each column for the weight values. These components should only occupy one column if they are providing `weightx`, and one row if they are providing `weighty`. If possible, use components on the top row to specify `weightx` and components in the left column to specify `weighty`.



## *Designing With* `GridBagLayout`

### *Example (Continued)*

To allocate weights to the rows and columns, identify one component in each column that needs to stretch horizontally, and one component in each row that stretches vertically. These components should occupy only a single cell along the axis of stretch, and be near the edges of the layout. This improves the consistency and readability of your code.

For this example, the stretch is in columns 1 and 3, and rows 2 through 5.

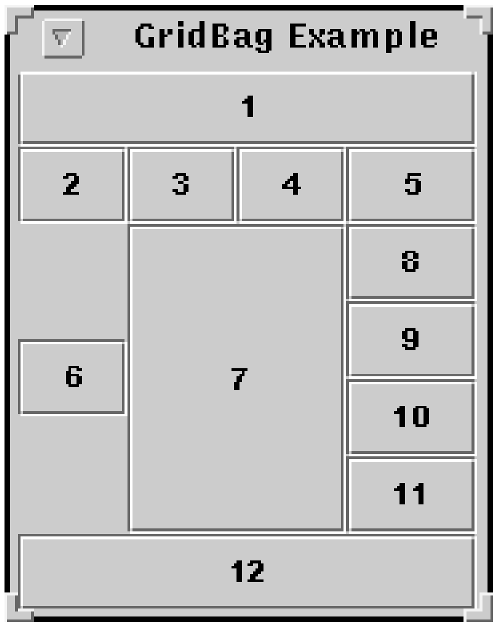
Components 8, 9, 10, and 11 are suitable to apply the vertical weight values for rows 2 through 5, and component 3 is appropriate to apply the horizontal weight for column 1. However, there is no obvious component with which a horizontal weight can be applied to column 3.

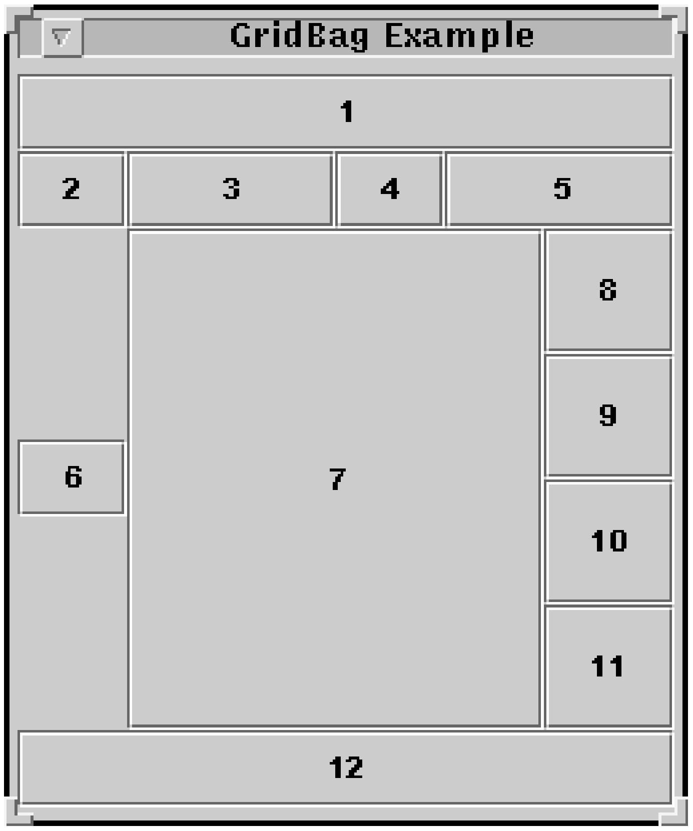
One way to approach this is to add a dummy component to the cell at row 2, column 3. This component must have zero by zero size so that it does not obscure component 8. A `Canvas` is suitable for this because its preferred size is zero by zero, unless explicitly set otherwise. Once added into row 3 column 4, it will remain at zero size provided it has a fill value of `NONE`.

***This approach can be used to simplify any layout. Create an extra row and an extra column along the bottom and right hand edges of your layout. Populate these cells with zero-sized canvases and use them to apply weights.***

## Designing With GridBagLayout

### Example (Continued)





Once the `GridBagConstraints` values have been applied to components added to a `GridBagLayout`, the desired behavior is achieved. The screen shots shown here are derived from the implementation program listed on the next page.

## *Designing With* `GridBagLayout`

### *Example (Continued)*

The main part of the program for this example, with the values used in the `GridBagConstraints`, is shown here.

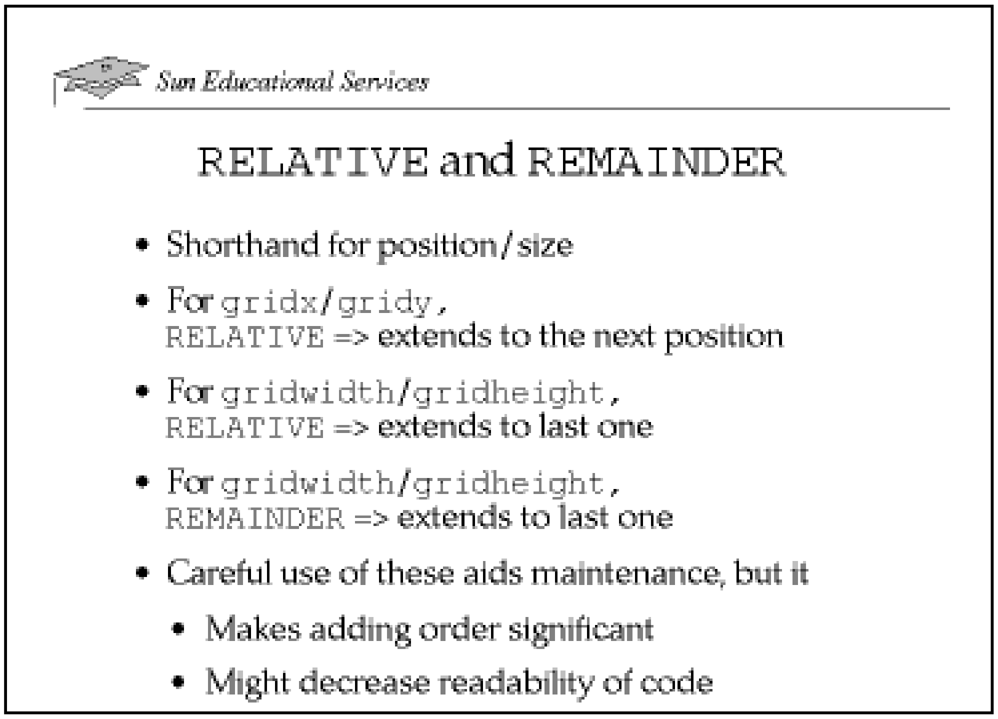
```
import java.awt.*;
import javax.swing.*;

public class ExampleGB {
  public static void main(String args[]) {
    JFrame f = new JFrame("GridBag Example");
    Container c = f.getContentPane();
    c.setLayout(new GridBagLayout());
    GridBagAdder.add(c, new Canvas(), 3, 2, 1, 1, 1, 0,
      GridBagConstraints.NONE, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("1"), 0, 0, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("2"), 0, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("3"), 1, 1, 1, 1, 1, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("4"), 2, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("5"), 3, 1, 2, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("6"), 0, 2, 1, 4, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("7"), 1, 2, 3, 4, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("8"), 4, 2, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("9"), 4, 3, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("10"), 4, 4, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("11"), 4, 5, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("12"), 0, 6, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    f.pack();
    f.setVisible(true);
  }
```

## Designing With GridBagLayout

### Example (Continued)

Supporting the code on the previous page is this inner class. It provides the add method that simplifies setting up the GridBagConstraints values.

```
  static class GridBagAdder {
    // OK to reuse this as we overwrite all elements every time
    // Note that this is not threadsafe however!
    static GridBagConstraints cons = new GridBagConstraints();
    public static void add(Container cont,Component comp,int x, int y,
       int width,int height,int weightx,int weighty,
       int fill,int anchor) {

      cons.gridx = x;
      cons.gridy = y;
      cons.gridwidth = width;
      cons.gridheight = height;
      cons.weightx = weightx;
      cons.weighty = weighty;
      cons.fill = fill;
      cons.anchor = anchor;
      cont.add(comp, cons);
    }
  }
}
```


Sun Educational Services

RELATIVE and REMAINDER

- Shorthand for position/size
- For gridx/gridy, RELATIVE => extends to the next position
- For gridwidth/gridheight, RELATIVE => extends to last one
- For gridwidth/gridheight, REMAINDER => extends to last one
- Careful use of these aids maintenance, but it
  - Makes adding order significant
  - Might decrease readability of code

## `RELATIVE` *and* `REMAINDER`

Where a layout involves a large number of components in a fairly simple layout, it can be quite time consuming to set up the `gridx` and `gridy` values for each one. This situation is aggravated when maintenance is needed; for example, to insert one new component.

To help with this situation, you can use the value `RELATIVE` to indicate that a component should be positioned just to the right, or just underneath, the one previously added.

In addition, you can use `RELATIVE` as a value in the `gridwidth` and `gridheight` fields, to make the component extend over all rows below, or all columns to the right, of the one to which the component is added, *except the last row or column*.

If you set the value `REMAINDER` in a `gridwidth` or `gridheight` fields, the component extends to the very last row or column.

## `RELATIVE` and `REMAINDER`

Careful use of these shorthand features can make code easier to write and shorter, which can make it easier to read. However, in some situations, the layout is dependent on the order of adding components and actually makes code more difficult to read.

# *Java Native Interface* C

## *Native Methods*

You have seen many features and basic functions within the Java programming language. However, you might want to perform tasks with applications written in Java programming language that you cannot accomplish with the Java programming language alone. In these instances, you can use native methods to link in C programs that handle your specific needs.

This added functionality comes at a price—your applications are no longer easily portable. Other machines sharing your architecture must have a local copy of your compiled C programs. Other machines with different architectures will require porting your C programs to those architectures where they can be compiled by a native compiler.

This process can be difficult for complicated programs or impossible for programs that rely on features found in the underlying operating system. However, if you are dealing with a single architecture or you want to add a well-defined adaptable feature, native methods can be the best option to meet your needs.

## *Native* `HelloWorld`

The first task will be to call a native method from a Java program. To do this, create a native method for the `HelloWorld` program.

✓ ***`HelloWorld` requires no arguments and supplies no return values. How to handle arguments is discussed later.***

The following is an overview of the four basic steps required to integrate native methods into your Java programs:

- Define a Java class with the appropriate native method declarations.
- Create a header file for use with your C modules. Use the `javah` utility to do this.
- Write the C modules containing the native methods.
- Compile the C code into a dynamic loadable library.

✓ ***JNI, unlike NMI of the past, does not require a stub file to interface to native code.***

### *Defining Native Methods*

Like other methods, you must declare all native methods you plan to use, and they must be defined within a class.

You can define your native `HelloWorld` method like this:

```
public native void nativeHelloWorld();
```

There are two changes from the other `public void` methods you have written:

- The key word `native` is used as a method modifier.
- The body of the method (the actual implementation) is not defined here; it is replaced with a semicolon (;).

## *A Native* HelloWorld

### *Defining Native Methods (Continued)*

You must place the native method declaration inside a class definition. The class containing the native method will also contain a `static` code block that loads the dynamic library with the implementation of your method. Here is an example of a class definition for the simple `nativeHelloWorld()` method:

```
class NativeHello {
   public native void nativeHelloWorld();
   static {
      System.loadLibrary(“hello1”);
   }
}
```

The Java runtime environment executes the defined `static` code block when the class is loaded. In the example above, the `hello1` library is loaded when the class `NativeHello` is loaded.

### *Calling Native Methods*

Once you have wrapped your native methods into a class, you can create objects of that class to access the native methods, just as you would with regular class methods. Here, for example, is a program that creates a new `NativeHello` object and calls your `nativeHelloWorld` method

```
class UseNative {
   public static void main (String args[])
{
      NativeHello nh = new NativeHello();
      nh.nativeHelloWorld();
   }
}
```

Use `javac` to compile the `.java` files. The `.class` files are used when creating header files..

## *A Native* HelloWorld

### *The* javah *Utility*

You can create a C header file with the javah utility, based on the NativeHello.class file. Invoke javah as follows:

```
% javah -jni NativeHello
```

The generated file, NativeHello.h, provides you with the information you need to write the C program. Here is the file as generated by javah for this example:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeHello */

#ifndef _Included_NativeHello
#define _Included_NativeHello
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      NativeHello
 * Method:     nativeHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_NativeHello_nativeHelloWorld
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif

```

The portion in bold characters gives the signature of the native method yet to be implemented.

## *A Native* `HelloWorld`

### *Coding C Functions for Native Methods*

At this point, the C program is the only piece of code missing. The C code you write must include the header file above, plus `jni.h`, supplied with the JDK in the `$JAVA_HOME/include` directory. (`$JAVA_HOME` refers to the "root" directory of the JDK.) Of course, include any other header files necessary for your functions as well.

For each function declared in the header file, you provide the body. For this example, the C file, called `MyNativeHello.c`, will look like:

```
#include <jni.h>
#include "NativeHello.h"
#include <stdio.h>

void Java_NativeHello_nativeHelloWorld
     (JNIEnv *env, jobject obj) {
  printf ("Hello from C");
}

```

## Putting It Together

Now that you have all of the pieces, you must tell the system how to assemble them. First, compile your C program. You might have to specify the location of the include files.

The following example uses the C compiler in the Solaris operating system:

```
% cc -I$JAVA_HOME/include -
I$JAVA_HOME/include/solaris -G MyNativeHello.c -o
libhello1.so
```

The two include directories you must access to compile this code are `$JAVA_HOME/include` and `$JAVA_HOME/include/solaris`. You can specify them on the command line or you can modify your `INCLUDE` environment variable.

This example uses the Microsoft C compiler for the Windows 95 or NT platforms:

```
C:\> cl MyNativeHello.c -Fehello1.dll -MD -LD javai.lib
```

Once you have created the library file, you can run your native method test program:

```
% java UseNative
Hello Native World!
```

or

```
C:\> java UseNative
Hello Native World!
```

If you get a `java.lang.UnsatisfiedLinkError`, you might need to update the system `LD_LIBRARY_PATH` variable to include the current directory (so JVM can find `libhello1.so`).

## Passing Information to a Native Method

The previous sample native method does not handle information accessed from the defining class, nor does it accept any arguments. Both of the following tasks occur regularly in programming.

### Passing a Java Primitive as an Argument

Arguments can be supplied to native methods in a Java program, just as they are supplied to other methods. Suppose the following code declaration exists in a file called `NativeHello2.java` for a native method which prints `count` times:

```
public native void nativeHelloWorld2(int count);
```

This declaration will produce the following entry in the header file `NativeHello2.h`:

```
JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
     (JNIEnv *, jobject, jint);
```

Now rewrite your C method to loop for the supplied number of times (which comes in as the method's third argument).

```
#include <jni.h>
#include "NativeHello2.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
     (JNIEnv *env, jobject obj, jint countMax) {
  int count;
  for (count = 0; count < countMax; count++) {
    printf ("Hello from C, count = %d\n",count);
  }
}
```

## Passing Information to a Native Method

### Accessing a Java Primitive as an Object Data Member

The most common requirement in a native method is access to the class data members. The `jni.h` file (in `$JAVA_HOME/include`) contains several interface functions for use with objects inside a native code module.

For example, consider writing a class that has two `int` variables, one of which is `static`, that are accessed and modified by a native method:

```
class NativeHello4 {
    static int statInt = 2;
    int instInt = 4;
    public native int nativeHelloWorld4();
    static {
        System.loadLibrary("hello4");
    }
}
```

Inside your C program, you can access these variables using `functions in <jni.h>`.

```
#include <jni.h>
#include "NativeHello4.h"
#include <stdio.h>

/* The names of the Java object fields to be accessed. */
#define STAT_FIELD_NAME "statInt"
#define INST_FIELD_NAME "instInt"

/* This method displays the statInt and instInt fields and
returns the product of the two. */
jint Java_NativeHello4_nativeHelloWorld4
    (JNIEnv *env, jobject obj) {

  /* Class object. Used to find all fields and access
  static ones.
  jclass class = (*env)->GetObjectClass(env,obj);

  jfieldID fid;    /* A field reference. */
  jint staticInt;  /* A C copy of the static int. */
  jint instanceInt;/* A C copy of the int. */

```

## Passing Information to a Native Method

### Accessing a Java Primitive as an Object Data Member (Continued)

```

      /* Get reference to the static field. The "class"
      argument connects the field to a class. The third
      argument is the field’s name, and the last argument is
      the field’s type. See the union jvalue entry in jni.h,
      then capitalize for the proper primitive value. */
      fid = (*env)->GetStaticFieldID(env, class,
            STAT_FIELD_NAME, "I");
      if (fid == 0)
        return;

      /* Get that field’s data. */
     staticInt = (*env)->GetStaticIntField(env, class, fid);

      /* Process it, change it... */
      printf
            ("In C, doubling original %s value of %d to %d\n",
            STAT_FIELD_NAME, staticInt, staticInt*2);
      staticInt *= 2;

      /* ... and store it back into the class object. */
      (*env)->SetStaticIntField(env, class, fid, staticInt);


      /* Now for the nonstatic int, part of the current
      object. Get the field reference as before... */
      fid = (*env)->GetFieldID
            (env, class, INST_FIELD_NAME, "I");
      if (fid == 0)
         return;

      /* Get the field. Refer to the object, not the class. */
      instanceInt = (*env)->GetIntField(env, obj, fid);

      /* Process it, change it... */
      printf
            ("In C, tripling original %s value of %d to %d\n",
            INST_FIELD_NAME, instanceInt, instanceInt*3);
     instanceInt *= 3;

```

## Passing Information to a Native Method

### Accessing a Java Primitive as an Object Data Member (Continued)

```
    /* ... and store it back. */
   (*env)->SetIntField(env, obj, fid, instanceInt);

    /* Return the product of the two. */
    return (staticInt * instanceInt);
}

```

## Passing Information to a Native Method

### Accessing Strings

As you may recall, strings in the Java programming language consist of 16-bit Unicode characters.
C strings, however, consist of 8-bit American Standard Code for Information Interchange (ASCII) characters. Because strings are common objects which are passed between Java code and native code, several functions have been defined in `jni.h` to help make string manipulation easier . The C datatype for strings in the Java programming language is `jstring`.

The type `unicode` is a simple `typedef` for an `unsigned short`.

Suppose a native method which prints out a string is declared in a Java program as follows:

```
public void nativeHelloWorld3(String printMe);
```

The following C function implements the native code for it. Note the importance of calling `ReleaseStringUTFChars` to free the memory allocated for the C string when done.

```
#include <jni.h>
#include "NativeHello3.h"
#include <stdio.h>

void Java_NativeHello3_nativeHelloWorld3 (
    JNIEnv *env, jobject obj, jstring javaString) {

  const char * CString;

   /* Convert java string to C string. */
   CString = (*env)->GetStringUTFChars(env, javaString, 0);

   printf ("In C, string is %s\n", CString);

   /* Tell VM to release mem for CString as done w/ it. */
   (*env)->ReleaseStringUTFChars(env, javaString, CString);
}
```

## Passing Information to a Native Method

### Accessing Strings

As a final example, consider accessing strings as object data members. Suppose a class is defined with a String field and native method as follows:

```
class NativeHello5 {
    String stringField = "original";
    public native String nativeHelloWorld5();
    static {
        System.loadLibrary("hello5");
    }
}
```

This class is instantiated and the native method called from the following program:

```
class UseNative5 {
    public static void main (String args[]) {
       String changedString;
       NativeHello5 nh = new NativeHello5();

       System.out.println ("In Java, nh’s string says '"
               + nh.stringField + "'");

       /* Call native method to print and change string in
       the current object, and return a third string. */
       changedString = nh.nativeHelloWorld5();

       System.out.println ("In Java, nh’s string says '"
               + nh.stringField + "'");

       System.out.println ("Native method returned '" +
               changedString + "'");

    }
}
```

The following native code extracts the string from the object, prints it out, changes and stores the new version, and then returns another new string in the function’s return:

```
#include <jni.h>
#include "NativeHello5.h"
#include <stdio.h>
```

## Passing Information to a Native Method

### Accessing Strings (Continued)

```

#define NEW_STRING1 "Revised"
#define NEW_STRING2 "Revised again"

jstring Java_NativeHello5_nativeHelloWorld5
     (JNIEnv *env, jobject obj) {

  jclass class = (*env)->GetObjectClass(env,obj);
  jfieldID fid;
  jstring javaString;
  const char *CString;

  fid = (*env)->GetFieldID
       (env, class, "stringField", "Ljava/lang/String;");
  if (fid == 0)
    return;

  /* Get the field reference for the java string. */
  javaString = (*env)->GetObjectField(env, obj, fid);

  /* Retrieve the C string from the object. */
  CString = (*env)->GetStringUTFChars(env, javaString, 0);

  printf ("In C, changing string from %s to %s\n",
       CString, NEW_STRING1);

  /* Tell VM to release memory for CString as done w/it. */
  (*env)->ReleaseStringUTFChars(env, javaString, CString);

  /* Alloc a new Java string to store back in the obj. */
  javaString = (*env)->NewStringUTF(env, NEW_STRING1);

  /* Store it back. */
  (*env)->SetObjectField(env, obj, fid, javaString);

  /* Allocate one more new Java string to return. */
  javaString = (*env)->NewStringUTF(env, NEW_STRING2);

  return (javaString);
}

```

## Summary

The exact process for integrating native methods is:

1. Create a program containing the native method declarations and the static code to load the dynamic library:

   ```
   vi NativeHello.java
   ```

2. Create a program containing calls to the native methods:

   ```
   vi UseNative.java
   ```

3. Compile the `.java` files:

   ```
   javac NativeHello.java UseNative.java
   ```

4. Create the C header file:

   ```
   javah -jni NativeHello
   ```

5. Create the C program implementing your native methods:

   ```
   vi MyNativeHello.c
   ```

6. Compile the dynamic library:

   ```
   cc -I$JAVA_HOME/include -
   I$JAVA_HOME/include/solaris -G MyNativeHello.c -o
   libhello.so
   ```

7. Set the `LD_LIBRARY_PATH` variable:

   ```
   setenv LD_LIBRARY_PATH .:$LD_LIBRARY_PATH
   ```

8. Run the program:

   ```
   java UseNative
   ```

See the JNI tutorial on the JavaSoft web site for information on other JNI functionality. It can be accessed via hyperlink from your locally loaded master Java API index.

# *Event 1.0.x to Event 1.1 Conversion* *D*

This appendix gives an overview of the event handling under JDK 1.0.*x* and JDK 1.1 and provides a table that maps 1.0.*x* events and corresponding methods to their 1.1 counterparts.

## *References*

**Additional Resources –** The contents of this appendix were obtained from the Web page "How to Convert Programs to the 1.1 AWT API." [Online]. Available: `http://www.javasoft.com/products/JDK/1.1/docs/guide/awt/HowToUpgrade.html`.

## Event Handling

### Event Handling Before JDK 1.1

Before JDK 1.1, the `Component` `handleEvent` method (along with the methods it called, such as the `action` method) was the center of event handling. Only `Component` objects could handle events, and the component that handled an event had to be either the component in which the event occurred or a component above it in the component containment hierarchy.

### Event Handling in JDK 1.1

In JDK 1.1, event handling is no longer restricted to objects in the component containment hierarchy, and the `handleEvent` method is no longer the center of event handling. Instead, objects of any type can register as event listeners. Event listeners receive notification only about the types of events they have registered their interest in. You do not have to create a `Component` subclass to handle events.

When upgrading to the JDK 1.1 release, the easiest way to convert event-handling code is to leave it in the same class, and make it a listener for that type of event.

Another possibility is to centralize event-handling code in one or more non-component listeners (adaptors or filters). This approach lets you separate the GUI of your program from implementation details. It requires that you modify your existing code so that the listeners can get whatever state information they require from the components. This approach can be worthwhile if you are trying to keep your program's architecture clean.

## Converting 1.0 Event Handling to 1.1

The biggest part of converting most 1.0 AWT-using programs to the 1.1 API is converting the event-handling code. The process can be straightforward, once you figure out which events a program handles and which components generate the events. Searching for “Event” in a source file lets you find the event-handling code.

**Note –** While you are looking at the code you should note whether any classes exist solely for the purpose of handling events; you might be able to eliminate such classes.

Table D-1 can be used in the conversion process to help map 1.0 events and methods to their 1.1 counterparts.

- The first column lists each 1.0 event type, along with the name of the method (if any) that is specific to the event.

  Where no method is listed, the event is always handled by the `handleEvent` method.

- The second column lists the 1.0 components that can generate the event type.
- The third column lists the listener interface that helps you handle the 1.1 equivalents of the listed events.
- The fourth column lists the methods in each listener interface.

**Table D-1** Event Conversion Table

| **1.0.*x*** | | **1.1** | |
|---|---|---|---|
| **Event/Method** | **Generated by** | **Interface** | **Methods** |
| ACTION_EVENT/action | Button<br>List<br>MenuItem<br>TextField | ActionListener | actionPerformed(ActionEvent) |
| | Checkbox<br>CheckboxMenuItem<br>Choice | ItemListener | itemStateChanged(ItemEvent) |
| WINDOW_DESTROY<br>WINDOW_EXPOSE<br>WINDOW_ICONIFY<br>WINDOW_DEICONIFY | Dialog<br>Frame | WindowListener | windowClosing(WindowEvent)<br>windowOpened(WindowEvent)<br>windowIconified(WindowEvent)<br>windowDeiconified(WindowEvent)<br><br>windowClosed(WindowEvent)*[a]<br>windowActivated(WindowEvent)*<br>windowDeactivated(WindowEvent)* |
| WINDOW_MOVED | Dialog<br>Frame | ComponentListener | componentMoved(ComponentEvent)<br>ComponentHidden(ComponentEvent)*<br>componentResized(ComponentEvent)*<br>componentShown(ComponentEvent)* |

**Table D-1** Event Conversion Table (Continued)

| **1.0.*x*** | | **1.1** | |
|---|---|---|---|
| **Event/Method** | **Generated by** | **Interface** | **Methods** |
| SCROLL_LINE_UP<br>SCROLL_LINE_DOWN<br>SCROLL_PAGE_UP<br>SCROLL_PAGE_DOWN<br>SCROLL_ABSOLUTE<br>SCROLL_BEGIN<br>SCROLL_END | Scrollbar | AdjustmentListener<br>(Or use the new ScrollPane class) | adjustmentValueChanged(AdjustmentEvent) |
| LIST_SELECT<br>LIST_DESELECT | Checkbox<br>CheckboxMenuItem<br>Choice<br>List | ItemListener | itemStateChanged(ItemEvent) |
| MOUSE_DRAG/mouseDrag<br>MOUSE_MOVE/mouseMove | Canvas<br>Dialog<br>Frame<br>Panel<br>Window | MouseMotionListener | mouseDragged(MouseEvent)<br>mouseMoved(MouseEvent) |
| MOUSE_DOWN/mouseDown<br>MOUSE_UP/mouseUp<br>MOUSE_ENTER/mouseEnter<br>MOUSE_EXIT/mouseExit | Canvas<br>Dialog<br>Frame<br>Panel<br>Window | MouseListener | mousePressed(MouseEvent)<br>mouseReleased(MouseEvent)<br>moseEntered(MouseEvent)<br>mouseExited(MouseEvent)<br>mouseClicked(MouseEvent)* |

**Table D-1** Event Conversion Table (Continued)

| **1.0.*x*** | | **1.1** | |
|---|---|---|---|
| **Event/Method** | **Generated by** | **Interface** | **Methods** |
| `KEY_PRESS/keyDown` `KEY_RELEASE/keyUp` `KEY_ACTION/keyDown` `KEY_ACTION_RELEASE/keyUp` | Component | KeyListener | keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)* |
| `GOT_FOCUS/gotFocus` `LOST_FOCUS/lostFocus` | Component | FocusListener | focusGained(FocusEvent) focusLost(FocusEvent) |
| *No 1.0 equivalent* | | ContainerListener | componentAdded(ContainerEvent) componentRemoved(ContainerEvent) |
| *No 1.0 equivalent* | | TextListener | textValueChanged(TextEvent) |

a. No 1.0 equivalent

## *Making a Component a Listener*

Follow the general steps below to convert a 1.0 component into a 1.1 listener:

1. Change the source file so that it imports the `java.awt.event` package:

   ```
   import java.awt.event.*
   ```

2. Use Table D-1 to determine which components generate each event type.

   For example, if you are converting event code that is in an `action` method, you should look for `Button`, `List`, `MenuItem`, `TextField`, `Checkbox`, `CheckboxMenuItem`, and `Choice` objects.

3. Change the class declaration so that the class implements the appropriate listener interfaces as indicated in Table D-1.

   For example, if you are trying to handle an `action` event generated by a `Button`, Table D-1 informs you that you must implement the `ActionListener` interface.

   ```
   public class MyClass extends SomeComponent
   implements ActionListener {
   ```

4. Determine where the components that generate the events are created. Just after the code that creates each one, register `this` as the appropriate type of listener. For example:

```
newComponentObject.addActionListener(this);
```

## Making a Component a Listener

5. Create empty implementations of all the methods in the listener interfaces your class must implement. Copy the event-handling code into the appropriate methods.

   For example, `ActionListener` has just one method, `actionPerformed`. An easier way to creating the new method and copy the event-handling code to it is to change the signature of an `action` method from

   ```
   public boolean action(Event event, Object arg) {
   ```

   to

   ```
   public void actionPerformed(ActionEvent event) {
   ```

6. Modify the event-handling code as follows:

   a. Delete all `return` statements.

   b. Change references to **`event`**`.target` to **`event`**`.getSource()`.

   c. Delete any code that unnecessarily tests for which component the event came from. (Now that events are forwarded only if the generating component has a listener, you do not have to worry about receiving events from an unwanted component.)

   d. Perform any other modifications required to make the program compile cleanly and execute correctly.

# Index

Please
Recycle

Adobe PostScript