Advanced Java

Preface

There are numerous good computer languages. There are relatively few languages with the momentum to make a real difference in software development. Java is one of those languages. The developers of Java had a chance to look at existing computer languages and address their deficiencies. Coming ten years after the introduction of C++, Java takes advantages of vast improvements in hardware and software technologies to address the issues raised by C++. Java allows inexperienced users to write high quality code. It incorporates user interface directly in the language. Java and Object-Oriented technology are a major paradigm shift. This course aims to introduce some advanced topics of java to enable the students to appreciate the power of these technologies. This course exposes the student to Advanced Java features such as Network Programming, Servlet Programming, the Java Database Connectivity, Remote Method Invocation, and Swing.

Unit 1 gives you a quick introduction to Java programming. It tells you what Java is and provides you with an opportunity to compile and run some simple Java programs. The unit gives you the background knowledge you need to understand how the advanced java programs work. Unit 2 gives an introduction to java GUI programming. Applets, AWT and Swings are covered by this unit. Unit 3 covers the networking and connectivity topics of java such as Networking, RMI, JDBC and Servlets. This course gives you an introduction to all the basic and advanced topics of Java. Each of these topics, by them selves are large enough to become an independent course. Students are encouraged to explore each of these interesting topics in much more detail.

BASIC JAVA

1.1 INTRODUCTION

The Java programming language originated as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the language of choice. But over time the difficulties encountered with C++ grew to the point where the problems could best be addressed by creating an entirely new language platform. Design and architecture decisions drew from a variety of languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result is a language platform that has proven ideal for developing secure, distributed, network-based end-user applications in environments ranging from network-embedded devices to the World-Wide Web and the desktop. The Java programming language is designed to meet the challenges of application development in the context of heterogeneous, network-wide distributed environments. Java technology is both a programming language and a platform.

1.2 OBJECTIVES

In this unit you will learn

- What is Java?
- What are the features of Java
- Java Language Syntax
- Object Oriented Features of Java
- What are Exceptions?
- Java Input/Output
- Multithreading

1.3 INTRODUCTION TO JAVA

The Java Programming Language

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

Simple	Architecture neutral
Object oriented	Portable
Distributed	High performance
Interpreted	Multithreaded
Robust	Dynamic
Secure	

Simple, Object Oriented, and Familiar

Primary characteristics of the Java programming language include a simple language that can be programmed without extensive programmer training. The Java programming language is designed to be object oriented from the ground up. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems must adopt object-oriented concepts. Java technology provides a clean and efficient object-based development platform. Java language has retained the familiar good features of C and C++ and has eliminated the most of the inefficient features of theses languages.

Robust and Secure

The Java programming language is designed for creating highly reliable software. It provides extensive compile-time checking, followed by a second level of run-time checking. Language features guide programmers towards reliable programming habits.

The memory management model is extremely simple: objects are created with a new operator. There are no explicit programmer-defined pointer data types, no pointer arithmetic, and automatic garbage collection. Java technology is designed to operate in distributed environments, which means that security is of paramount importance. With security features designed into the language and run-time system, Java technology lets you construct applications that can't be invaded from outside. In the network environment, applications written in the Java programming language are secure from intrusion by unauthorized code attempting to get behind the scenes and create viruses or invade file systems.

Architecture Neutral and Portable

Java technology is designed to support applications that will be deployed into heterogeneous network environments. In such environments, applications must be capable of executing on a variety of hardware architectures. Within this variety of hardware platforms, applications must execute atop a variety of operating systems and interoperate with multiple programming language interfaces. To accommodate the diversity of operating environments, the Java Compiler product generates bytecodes--an architecture neutral intermediate format designed to transport code efficiently to multiple hardware and software platforms. The interpreted nature of Java technology solves both the binary distribution problem and the version problem; the same Java programming language byte codes will run on any platform.

Architecture neutrality is just one part of a truly portable system. Java technology takes portability a stage further by being strict in its definition of the basic language. Java technology specifies the sizes of its basic data types and the behavior of its arithmetic

operators. Java programs are the same on every platform--there are no data type incompatibilities across hardware and software architectures.

The architecture-neutral and portable language platform of Java technology is known as the Java virtual machine. It's the specification of an abstract machine for which Java programming language compilers can generate code. Specific implementations of the Java virtual machine for specific hardware and software platforms then provide the concrete realization of the virtual machine.

High Performance

Performance is always a consideration. The Java platform achieves superior performance by adopting a scheme by which the interpreter can run at full speed without needing to check the run-time environment. The automatic garbage collector runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance. Applications requiring large amounts of compute power can be designed such that compute-intensive sections can be rewritten in native machine code as required and interfaced with the Java platform.

Interpreted, Threaded, and Dynamic

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter and run-time system have been ported. In an interpreted platform such as Java technology-based system, the link phase of a program is simple, incremental, and lightweight. Programmers benefit from much faster development cycles--prototyping, experimentation, and rapid development are the normal case, versus the traditional heavyweight compile, link, and test cycles.

Java technology's multithreading capability provides the means to build applications with many concurrent threads of activity. Multithreading thus results in a high degree of

interactivity for the end user. The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives: the language library provides the Thread class, and the run-time system provides monitor and condition lock primitives. At the library level, moreover, Java technology's high-level system libraries have been written to be thread safe: the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution.

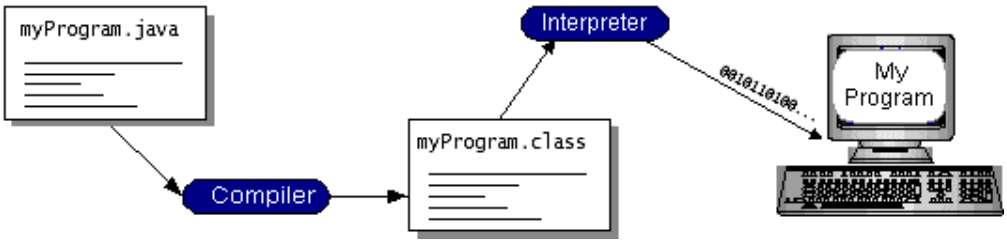
While the Java Compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages. Classes are linked only as needed. New code modules can be linked in on demand from a variety of sources, even from sources across a network..

The Java Platform

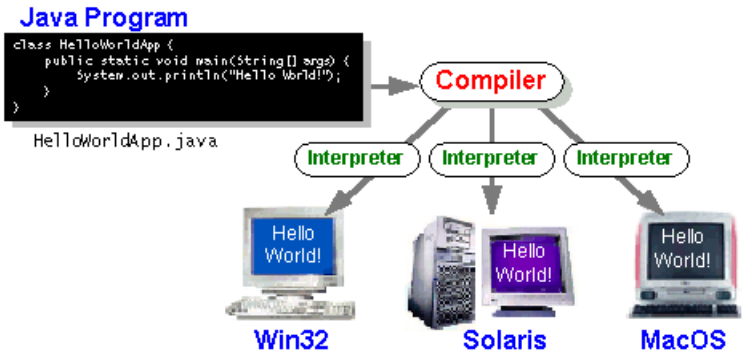
Taken individually, the characteristics discussed above can be found in a variety of software development platforms. What's completely new is the manner in which Java technology and its runtime environment have combined them to produce a flexible and powerful programming system.

Developing applications using the Java programming language results in software that is portable across multiple machine architectures, operating systems, and graphical user interfaces, secure, and high performance.

With most programming languages, you either compile or interpret a program so that you can run it on your computer. The Java programming language is unusual in that a program is both compiled and interpreted. With the compiler, first you translate a program into an intermediate language called Java bytecodes —the platform-independent codes interpreted by the interpreter on the Java platform. The interpreter parses and runs each Java bytecode instruction on the computer. Compilation happens just once; interpretation occurs each time the program is executed. The following figure illustrates how this works.



Java bytecodes can be thought as the machine code instructions for the Java Virtual Machine (Java VM). Every Java interpreter, whether it's a development tool or a Web browser that can run applets, is an implementation of the Java VM. Java bytecodes help make "write once, run anywhere" possible. You can compile your program into bytecodes on any platform that has a Java compiler. The bytecodes can then be run on any implementation of the Java VM. That means that as long as a computer has a Java VM, the same program written in the Java programming language can run on Windows 2000, a Solaris workstation, or on an iMac.

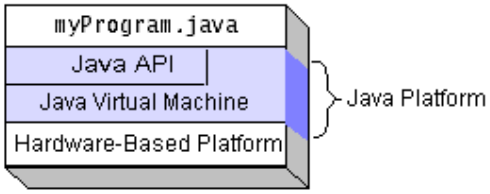


A platform is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Windows 2000, Linux, Solaris, and MacOS. Most platforms can be described as a combination of the operating system and hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms. The Java platform has two components:

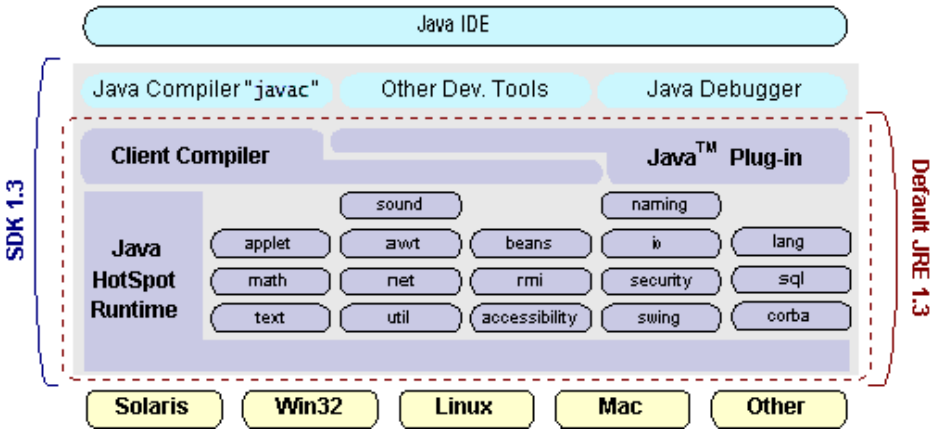
- The Java Virtual Machine (Java VM)
- The Java Application Programming Interface (Java API)

Java VM is the base for the Java platform and is ported onto various hardware-based platforms. The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets. The Java API is grouped into libraries of related classes and interfaces; these libraries are known as packages.

The following figure depicts a program that's running on the Java platform. As the figure shows, the Java API and the virtual machine insulate the program from the hardware.



Native code is code that after you compile it, the compiled code runs on a specific hardware platform. As a platform-independent environment, the Java platform can be a bit slower than native code. However, smart compilers, well-tuned interpreters, and just-in-time bytecode compilers can bring performance close to that of native code without threatening portability.



First Program In Java: Hello World

The following instructions will help you write your first program. These instructions are for users of Win32 platforms, which include Windows 95/98 and Windows NT/2000. We start with a checklist of what you need to write your first program.

To write your first program, you need:

1. The Java 2 Platform, Standard Edition.
2. A text editor

These two items are all you need to write your first Java program. Your first program, HelloWorldApp, will simply display the greeting "Hello world!". To create this program, you will:

- **Create a source file.** A source file contains text, written in the Java programming language, that you and other programmers can understand. You can use any text editor to create and edit source files.

1. Start NotePad. in a new document, type in the following code:

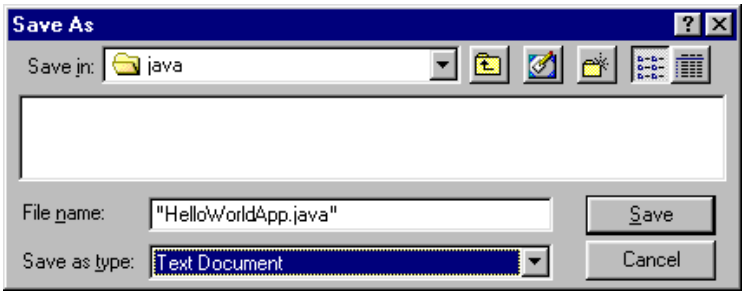
```
/**
 * The HelloWorldApp class implements an application that
 * displays "Hello World!" to the standard output.
 */
public class HelloWorldApp {
    public static void main(String[] args) {
        // Display "Hello World!"
        System.out.println("Hello World!");
    }
}
```

Type all code, commands, and file names exactly as shown. The Java compiler and interpreter are case-sensitive, so you must capitalize consistently

2. Save this code to a file. From the menu bar, select File > Save As. In the Save As dialog box:

- Using the Save in drop-down menu, specify the folder (directory) where you'll save your file. In this example, the directory is java on the C drive.
- In the File name text box, type "HelloWorldApp.java", including the double quotation marks.
- From the Save as type drop-down menu, choose Text Document.

When you're finished, the dialog box should look like this:



Now click Save, and exit NotePad.

- **Compile the source file into a bytecode file.** The compiler, javac, takes your source file and translates its text into instructions that the Java Virtual Machine (Java VM) can understand. The compiler converts these instructions into a bytecode file.

From the Start menu, select the MS-DOS Prompt application (Windows 95/98) or Command Prompt application (Windows NT). To compile your source code file, change your current directory to the directory where your file is located. For example, if your source directory is java on the C drive, you would type the following command at the prompt and press Enter:

```
cd c:\java
```

Now the prompt should change to C:\java>. Now you can compile. At the prompt, type the following command and press Enter:

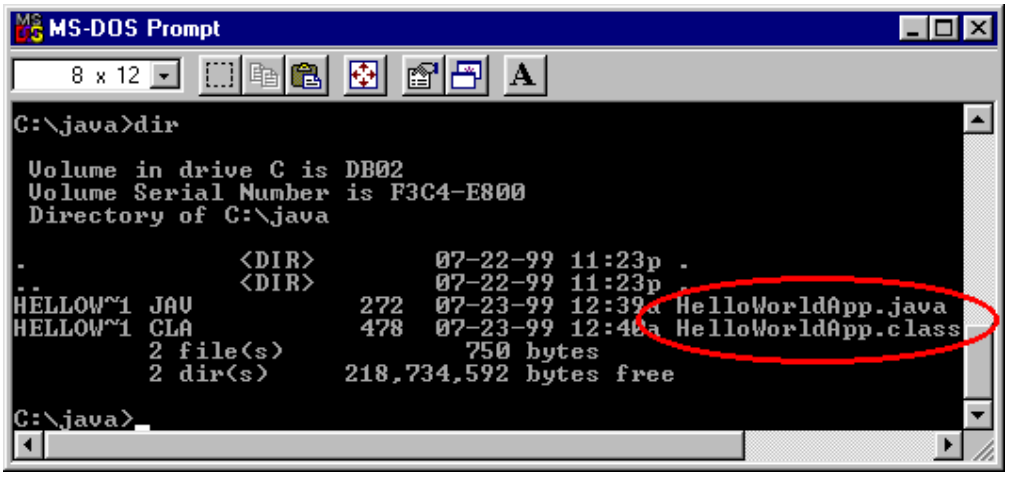
```
javac HelloWorldApp.java
```

If your prompt reappears without error messages, congratulations. You have successfully compiled your program.

The compiler has generated a Java bytecode file, HelloWorldApp.class. At the prompt,

Error Explanation
Bad command or file name *(Windows 95/98)*
The name specified is not recognized as an internal or external command, operable program or batch file *(Windows NT)*
If you receive this error, Windows cannot find the Java compiler, javac.
Here's one way to tell Windows where to find javac. Suppose you installed the Java 2 Software Development Kit in C:\jdk1.3. At the prompt you would type the following command and press Enter:
C:\jdk1.3\bin\javac HelloWorldApp.java
Note: If you choose this option, each time you compile or run a program, you'll have to precede your javac and java commands with C:\jdk1.3\bin\. To avoid this extra typing, consult the section *set the class path variable correctly*.

type dir to see the new file that was generated:



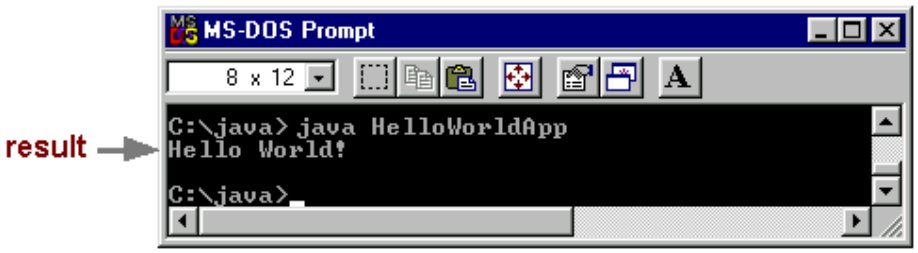
- Now that you have a .class file, you can run your program.

- **Run the program contained in the bytecode file.** The Java interpreter installed on your computer implements the Java VM. This interpreter takes your bytecode file and carries out the instructions by translating them into instructions that your computer can understand.

In the same directory, enter at the prompt:

```
java HelloWorldApp
```

Now you should see:



Error Explanation

Exception in thread "main" java.lang.NoClassDefFoundError: HelloWorldApp

If you receive this error, java cannot find your bytecode file, HelloWorldApp.class.

One of the places java tries to find your bytecode file is your current directory. So, if your bytecode file is in C:\java, you should change your current directory to that. To change your directory, type the following command at the prompt and press **Enter**:

```
cd c:\java
```

The prompt should change to C:\java>. If you enter dir at the prompt, you should see your .java and .class files. Now enter java HelloWorldApp again.
If you still have problems, you might have to change your CLASSPATH variable. To see if this is necessary, try "clobbering" the classpath with the following command:

```
set CLASSPATH=
```

Now enter java HelloWorldApp again.

Now that you've seen a Java application (and perhaps even compiled and run it), you might be wondering how it works. This section dissects the "Hello World" application you've already seen. Here, again, is its code:

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

Comments in Java Code

The "Hello World" application has two blocks of comments. The first block, at the top of the program, uses /** and */ delimiters. Later, a line of code is explained with a comment that's marked by // characters. The Java language supports a third kind of comment, as well -- the familiar C-style comment, which is delimited with /* and */.

Defining a Class

In the Java language, each method (function) and variable exists within a class or an object (an instance of a class). The Java language does not support global functions or variables. Thus, the skeleton of any Java program is a class definition.

A class--the basic building block of an object-oriented language such as Java--is a template that describes the data and behavior associated with instances of that class. When you instantiate a class you create an object that looks and feels like other instances of the same class. The data associated with a class or object is stored in variables; the behavior associated with a class or object is implemented with methods. Methods are similar to the functions or procedures in procedural languages such as C.

In the Java language, the simplest form of a class definition is

```
class name {
   . . .
}
```

The keyword `class` begins the class definition for a class named `name`. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The "Hello World" application has no variables and has a single method named `main`.

The `main` Method

The entry point of every Java application is its main method. When you run an application with the Java interpreter, you specify the name of the class that you want to run. The interpreter invokes the main method defined within that class. The main method controls the flow of the program, allocates whatever resources are needed, and runs any other methods that provide the functionality for the application. Every Java application must contain a main method whose signature looks like this:

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

The method signature for the main method contains three modifiers:

- public indicates that the main method can be called by any object.
- static indicates that the main method is a class method.
- void indicates that the main method doesn't return any value.

How the main Method Gets Called

The main method in the Java language is similar to the main function in C and C++. When the Java interpreter executes an application (by being invoked upon the application's controlling class), it starts by calling the class's main method. The main method then calls all the other methods required to run your application. If you try to invoke the Java interpreter on a class that does not have a main method, the interpreter refuses to run your program and displays an error message similar to this:

In class NoMain: void main(String argv[]) is not defined

Arguments to the main Method

The main method accepts a single argument: an array of elements of type String. public static void main(**String[] args**)

This array is the mechanism through which the runtime system passes information to your application. Each String in the array is called a command-line argument. Command-line arguments let users affect the operation of the application without recompiling it. The "Hello World" application ignores its command-line arguments.

Using Classes and Objects

The other components of a Java application are the supporting objects, classes, methods, and Java language statements that you write to implement the application.

The "Hello World" application is about the simplest Java program you can write that actually does something. Because it is such a simple program, it doesn't need to define any classes except for HelloWorldApp. The "Hello World" application does use another class--the System class--that is part of the API (application programming interface) provided with the

Java environment. The System class provides system-independent access to system-dependent functionality.

Let's take a look at the first segment of the statement:

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

The construct System.out is the full name of the out variable in the System class. Notice that the application never instantiates the System class and that out is referred to directly from the class name. This is because out is a class variable--a variable associated with the class rather than with an instance of the class. You can also associate methods with a class--class methods.

To refer to class variables and methods, you join the class name and the name of the class method or class variable together with a period (".").

Using an Instance Method or Variable

Methods and variables that are not class methods or class variables are known as instance methods and instance variables. To refer to instance methods and variables, you must reference the methods and variables from an object.

While System's out variable is a class variable, it refers to an instance of the PrintStream class (a class provided with the Java development environment) that implements the standard output stream. When the System class is loaded into the application, it instantiates PrintStream and assigns the new PrintStream object to the out class variable. Now that you have an instance of a class, you can call one of its instance methods:

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

As you can see, you refer to instance methods and variables similarly to the way you refer to class methods and variables. You join an object reference (out) and the name of the instance method or variable (println) together with a period ("."). The Java compiler allows you to cascade references to class and instance methods and variables together, resulting in constructs like the one that appears in the sample program:

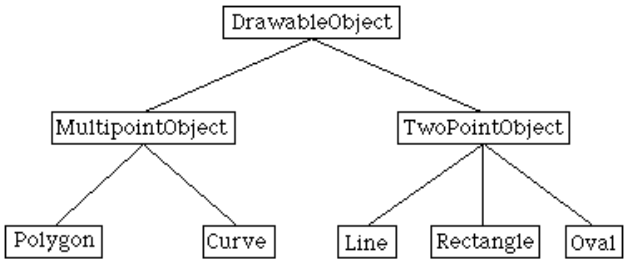
System.out.println("Hello World!");

This line of code displays "Hello World!" to the application's standard output stream.

Objects and Object-oriented Programming

The central concept of object-oriented programming is the object, which is a kind of module containing data and functions. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal state (the data it contains) and that can respond (behavior) to messages (calls to its methods). The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other.

You should think of objects as "knowing" how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a "print" message would produce very different results, depending on the object it is sent to. This property of objects -- that different objects can respond to the same message in different ways -- is called polymorphism. It is common for objects to bear a kind of "family relationship" to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same class. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.



DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be subclasses of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to inherit the properties of that class. The subclass can add to its inheritance and it can even "override" part of that inheritance (by defining a different response to some method). Nevertheless, lines, rectangles, and so on **are** drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

The reusable components should be as "modular" as possible. A module is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be "plugged into" a system. The details of what goes on inside the module are not important to the system as a whole, as long as the module fulfills its assigned role correctly. This is called information hiding, and it is one of the most important principles of object oriented programming. One common format for software modules is to contain some data, along with some functions for manipulating that data. (ie objects) . Objects provide an interface, hiding the details of implementation, thus providing information hiding. The putting together of data and associated functions can be called as encapsulation.

1.4 LANGUAGE FUNDAMENTALS

Data Types

The Java type system supports a variety of primitive (built-in) data types, such as int for representing integer data, float for representing floating-point values, and others, as well as class-defined data types that exist in supporting libraries (Java packages). All Java primitive data types have lowercase letters. The Java programming language has the following primitive types:

Primitive Type	Description
boolean	True/false
byte	8 bits
char	16 bits (UNICODE)
short	16 bits
int	32 bits
long	64 bits
float	32 bits IEEE 754-1985
Double	64 bits IEEE 754-1985

Variable Definition and Assignment

A data definition operation specifies a data type and a variable name, and optionally, an initial value:

Data Definition

<data-type> <variable>;
<data-type> <variable-1>, <variable-2>, ..., <variable-n>;
<data-type> <variable> = <data-value>;

The data type may be a primitive or built-in type or a user-defined type such as Dog. The value may be a literal value or an instance of a user-defined type such as Dog. Several examples of data definitions follow:

Data Definition Examples
int x;
int x = 9;
Boolean terminate = false;
Dog dog = new Dog();

An assignment operation can occur in the following contexts:

Assignment Operation
<data-type> <variable> = <data-value>;
<data-type> <variable>; <other-statements>... <variable> = <data-value>;

The data value to the right of the assignment operator can be a literal value, or any operation that produces a scalar value. Several examples follow:

Assignment Example	**Comment**
int x = 4;	Data definition with assignment
x = 9;	Assumes prior definition of x
temperature = 21.4;	Assumes prior definition of temperature
dog = new Dog();	Assumes prior definition of dog

Data Type	Default Initialization Value
Boolean	false
byte	0
char	\u0000
Short	0
int	0
long	0
Float	0.0
double	0.0
<user-defined-type>	null

Arrays

An array is a linear collection of data elements, each element directly accessible via its index. The first element has index 0; the last element has index n - 1. The array has the form:

Generic Array Object
elements
element type
element 0
element 1

...
element n - 1

Java type system provides built-in, language-level syntactic support for arrays. Although language-level support for arrays increases the complexity of the language definition, it's justified because array usage is entrenched in traditional programming. The syntax for creating an array object is:

Array Definition
<data-type>[] <variable-name>;

This declaration defines the array object--it does not allocate memory for the array object, nor does it allocate the elements of the array. Also, you may not specify a size within the square brackets. To allocate an array, use the new operator:

```
int[] x = new int[5];    // array of five elements
```

The array x of Java primitives has the form:

new int[5]
5
int
0
0
0
0
0

Consider an array definition for a user-defined type such as Dog:

```
Dog[] dog = new Dog[5];
```

This definition creates the array object itself, but not the elements. Subsequently, you can use the new operator to create objects in order to initialize the array elements (which are reference variables):

dog[0] = new Dog();

...

dog[4] = new Dog();

To create multidimensional arrays, simply create arrays of arrays, for example,

T[][] t = new T[10][5];

This definition creates ten arrays of references to arrays of references for objects of type T. This definition does not allocate memory for instances of T. There is a short-hand notation for defining an array and initializing its elements in one step, using comma-separated data values between curly brackets:

Array Definition and Initialization
<data-type>[] <variable-name> = { <expression>, <expression>, ... };

The following table provides examples:

Examples of Array Definition and Initialization
int x = 4; int[] anArray = {3, x, 9, 2};
String[] seasons = {"winter", "spring", "summer", "fall"};

Note that the array size is determined from the number of initializers. Accessing an undefined array element causes a runtime exception called ArrayIndexOutOfBoundsException.

Accessing a defined array element that has not yet been assigned to an object results in a runtime exception called NullPointerException.

Strings

String is a system-defined class--not a primitive--defined in java.lang.The lang package also provides a complementary class, java.lang.StringBuffer. String instances are immutable; that is, they cannot be modified. To use equivalent terminology, String operations are nondestructive. A programmer simply creates strings, uses them, and when there is no further reference to them, the Java interpreter's garbage collection facility recovers the storage space. Most of the string-oriented tasks necessary for normal programming can be accomplished with instances of String, for example, creating string constants, concatenating strings, and so on.

StringBuffer, on the other hand, is more powerful. It includes many methods for destructive string operations, for example, substring and character-level manipulation of strings such as splicing one or more characters into the middle of a string. A good rule of thumb is to use String wherever possible, and consider StringBuffer only when the functionality provided by String is inadequate.

```
System.out.println("x = " + x);
```

This simple line of code demonstrates several string-related issues. First, note that println accepts one argument, which is satisfied by the result of the expression evaluation that includes +. In this context, + performs a string concatenation. Because + is recognized by the Java compiler as a string concatenation operator, the compiler automatically generates the code to convert any non-String operands to String instances. In this case, if x is an int with the value 5, its value will be converted, generating the string constant "5". The latter is concatenated with "x = " producing "x = 5", the single argument to println.

Because String is a class, the general way to create a string instance is:

String prompt = new String("xyz ");

Note that you have to provide a string in order to create a string! As a convenience for the programmer, the Java programming language always recognizes a sequence of characters between double quotes as a string constant; hence, you can use the following short-cut to create a String instance and assign it to the reference variable prompt:

String prompt = "x = ";

String barkSound = "Woof.";

Conditional Execution

Like other languages, the Java programming language provides constructs for conditional execution, specifically, if, switch, and the conditional operator ?.

Conditional Constructs

```
if (<boolean-expression>)
  <statement>...
else
  <statement>...
```

```
switch (<expression>) {
  case <const-expression>:
    <statements>...
    break;

  more-case-statement-break-groups...

  default:
    <statements>...
}
```

```
(<boolean-expression>) ? <if-true-expression>
```

```
: <if-false-expression>
```

The more general construct, if has the syntax:

if (<boolean-expression>)

  <statement>...

where <statement>... can be one statement, for example,

x = 4;

or multiple statements grouped within curly brackets (a statement group), for example,

{

  x = 4;

  y = 6;

}

and <boolean-expression> is any expression that evaluates to a boolean value, for example,

Boolean Expression	**Interpretation**
X < 3	x is less than 3
X == y	x is equal to y
X >= y	x is greater than or equal to y
X != 10	x is not equal to 10
<variable>	variable is true

If the boolean expression evaluates to true, the statement (or statement group) following the if clause is executed. The Java programming language also supports an optional else clause; the syntax is:

if (<boolean-expression>)

```
  <statements>...
else
  <statements>...
```

If the boolean expression evaluates to true, the statement (or statement group) following the if clause is executed; otherwise, the statement (or statement group) following the else clause is executed. Boolean expressions often include one or more comparison operators, which are listed in the following table:

Comparison Operator	**Interpretation**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

With the Java programming language, boolean values are literals (case is significant), and boolean variables accept either value.

Iterative Execution

The Java programming language provides the while, do-while, and for constructs for iterating over a statement (or statement group) multiple times. while is the more general iterative construct; for is the more syntactically powerful.

Iterative Constructs
while (<boolean-expression>)   <statements>...
Do   <statements>... while (<boolean-expression>);
for (<init-stmts>...; <boolean-expression>; <exprs>...)   <statements>...

Multifunction Operators

Java supports several multifunction operators described in the following table:

Multifunction Operator	**Interpretation**
++	increment (by 1)
--	decrement (by 1)
+=	increment (by specified value)
-=	decrement (by specified value)
*=	multiply (by specified value)
/=	divide (by specified value)
&=	bitwise and (with specified value)
|=	bitwise inclusive or (with specified value)
^=	bitwise exclusive or (with specified value)

%=	integer remainder (by specified value)

These operators are multifunction operators in the sense that they combine multiple operations: expression evaluation followed by variable assignment. The following table includes examples and interpretations:

Multifunction Operator	**Example**	**Pedestrian Equivalent**
++	x++, ++x	x = x + 1
--	x--, --x	x = x - 1
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
&=	x &= y	x = x & y
|=	x |= y	x = x | y
^=	x ^= y	x = x ^ y
%=	x %= y	x = x % y

Note that the Java programming language restricts bitwise operations with &, |, and ^ to integer values, which makes sense.

Comment Syntax

The syntax of the Java programming language supports three types of commenting, illustrated in the following table:

Comment Example	Description
int x; // a comment	Remainder of line beginning with "//" is a comment area
/* The variable x is an integer: */ int x;	All text between the "/*" and "*/", inclusive, is ignored by compiler
/** x -- an integer representing the x coordinate */ int x;	All text between the "/**" and "*/", inclusive, is ignored by compiler and intended for javadoc documentation utility

Reference Variable Usage

It's common to use the term reference variable for any variable that holds a reference to dynamically allocated storage for a class instance, for example, fido in the following code:
Dog fido = new Dog();

In reality, all variables in high-level languages provide a symbolic reference to a low-level data storage area. Consider the following code:
int x;
Dog fido;

Each of these variables represents a data storage area that can hold one scalar value. Using x you can store an integer value such as 5 for subsequent retrieval (reference). Using fido you can store a data value that is the low-level address (in memory) of a dynamically allocated instance of a user-defined data type. The critical point is that, in both cases, the

variable "holds" a scalar value. In both cases, you can use an assignment operation to store a data value:

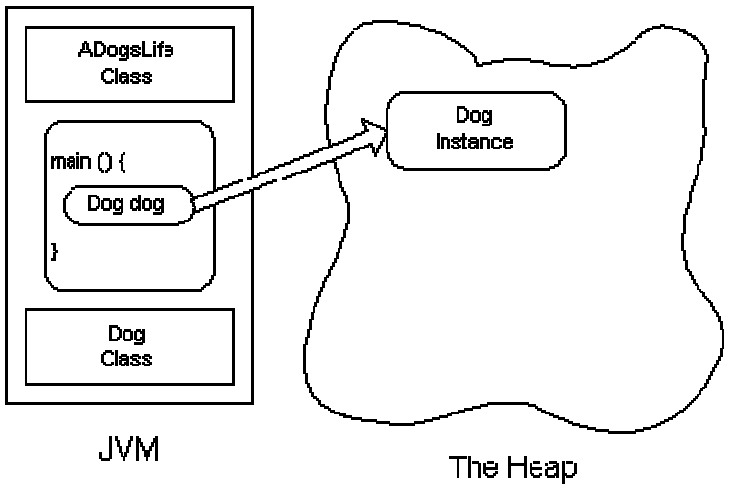
```
int x = 5;          // 1.
int y = x;          // 2. x's value also stored in y
Dog fido = new Dog();  // 3.
Dog myDog = fido;      // 4. fido's value also stored  in myDog
Dog spot = null;       // 5.
```

In the second line, `y` is initialized with the current value of `x`. In the fourth line, `myDog` is initialized with the current value of `fido`. Note, however, that the value in `fido` is not the instance of `Dog`; it is the Java interpreter's "recollection" of where (in memory) it stored the instance of `Dog`. Thus, you can use either reference variable to access this one instance of `Dog`.

With objects, the context in which you use the variable determines whether it simply evaluates to the memory address of an object or actually initiates more powerful operations. When the usage involves dot notation, for example, `fido.bark`, the evaluation includes binding the object to the appropriate method from the class definition, that is, invoking a method and performing the implied actions. But, when the usage is something like `... = fido;`, the evaluation is simply the address.

Garbage Collection

One of the really powerful features of the Java virtual machine is its memory-management strategy. The Java virtual machine allocates objects on the heap dynamically as requested by the `new` operator:



Other languages put the burden on the programmer to free these objects when they're no longer needed with an operator such as delete or a library function such as free. The Java programming language does not provide this functionality for the programmer because the Java runtime environment automatically reclaims the memory for objects that are no longer associated with a reference variable. This memory reclamation process is called garbage collection.

Garbage collection involves (1) keeping a count of all references to each object and (2) periodically reclaiming memory for all objects with a reference count of zero. Garbage collection is an old computing technology; it has been used with several computing languages and with environments such as text editors.

Consider the following method:

```
...
  void aMethod() {
    Dog dog = new Dog();
    // do something with the instance dog
    ...
  }
...
```

dog is a local variable within aMethod, allocated automatically upon method invocation. The new operation creates an instance of Dog; its memory address is stored in dog; and, its reference count is incremented to 1. When this method finishes/returns, the reference variable dog goes out of scope and the reference count is decremented to 0. At this point, the Dog instance is subject to reclamation by the garbage collector.

Next, consider the following code segment:

```
...
while (true)
  Dog dog = new Dog();
...
```

Each iteration of the while loop (which continues forever) allocates a new instance of Dog, storing the reference in the variable dog and replacing the reference to the Dog instance allocated in the previous iteration. At this point, the previously allocated instance is subject to reclamation by the garbage collector.

The garbage collector automatically runs periodically. You can manually invoke the garbage collector at any time with System.gc. However, this is only a suggestion that system runs the garbage collector, not a forced execution.

Runtime Environments and Class Path Settings

The Java runtime environment dynamically loads classes upon the first reference to the class. It searches for classes based on the directories listed in the environment variable CLASSPATH. If you use an IDE, it may automatically handle CLASSPATH internally, or write a classpath setting to the appropriate system file during installation.

If you do not use an IDE, for example, if you're using the Java 2 SDK, you may have to set a classpath before running the Java compiler and interpreter, javac and java, respectively.

Also, note that in some situations the installation procedure will automatically update the PATH environment variable, but if you're unable to run javac or java, be aware that this setting could be unchanged.

PATH environment variable settings vary across operating systems and vendors. In a Windows environment, the following setting augments the old/existing PATH setting (%PATH%) with C:\java\bin, the default installation directory for the JDK:

```
set PATH=%PATH%;C:\java\bin
```

When attempting to run a Java IDE, or the Java 2 SDK's compiler or interpreter, Windows includes the directory C:\java\bin in the search for the executable program. Of course, this setting (C:\java\bin) will vary from one developer environment to the next. Note that the path separator character is ; for Windows environments and : for UNIX environments.

If you find it necessary to set the CLASSPATH environment variable, for example, if you're using an early JDK from Sun, it should include all directories on your computer system where you have Java class files that you want the Java compiler and interpreter to locate. As you add new class-file directories, you will typically augment this classpath setting.

Windows 9x and NT/2000 users can set classpaths manually with a text editor in the file autoexec.bat, plus Windows NT/2000 users can set a classpath via the control panel's System dialog. UNIX users can set a classpath manually in the appropriate shell script's configuration file. Please refer to the appropriate system reference books and documentation that describe how to set an environment variable.

1.5 CLASSES AND OBJECTS AND OTHER OOP FEATURES

Objects, Instance Methods, and Instance Variables

Object Oriented Programming represents an attempt to make programs more closely model the way people think about and deal with the world. In the older styles of programming, a programmer who is faced with some problem must identify a computing task that needs to be performed in order to solve the problem. Programming then consists of finding a sequence of instructions that will accomplish that task. But at the heart of object-oriented programming, instead of tasks we find objects -- entities that have behaviors, that hold information, and that can interact with one another. Programming consists of designing a set of objects that somehow model the problem at hand. Software objects in the program can represent real or abstract entities in the problem domain. This is supposed to make the design of the program more natural and hence easier to get right and easier to understand. An object-oriented programming language such as Java includes a number of features. In order to make effective use of those features, you have to "orient" your thinking correctly.

Objects are real world entities. Objects are closely related to classes. A class can contain variables and methods. Classes "describe" objects, or more exactly objects belong to classes, but this might not be much clearer. It is more exact to say that classes are used to create objects. A class is a kind of factory for constructing objects. The non-static parts of the class specify, or describe, what variables and methods the objects will contain. Objects are created and destroyed as the program runs, and there can be many objects with the same structure, if they are created using the same class.

Consider a simple class whose job is to group together a few static member variables. For example, the following class could be used to store information about the person who is using the program:

```
class UserData {
   static String name;
   static int age;
}
```

In a program that uses this class, there is only one copy of each variable in the class, UserData.name and UserData.age. The class, UserData, and the variables it contains exist as long as the program runs. Now, consider a similar class that includes non-static variables:

```
class PlayerData {
  String name;
  int age;
}
```

In this case, there is no such variable as PlayerData.name or PlayerData.age, since name and age are not static members of PlayerData. So, there is nothing much in the class at all -- except the potential to create objects. But, it's a lot of potential, since it can be used to create any number of objects! Each of those objects will have its **own** variables called name and age. A program might use this class to store information about multiple players in a game. Each player has a name and an age. When a player joins the game, a new PlayerData object can be created to represent that player. If a player leaves the game, the PlayerData object that represents that player can be destroyed. A system of objects in the program is being used to dynamically model what is happening in the game. You can't do this with "static" variables!

An object that belongs to a class is said to be an instance of that class. The variables that the object contains are called instance variables. The functions that the object contains are called instance methods. For example, if the PlayerData class, as defined above, is used to create an object, then that object is an instance of the PlayerData class, and name and age are instance variables in the object. It is important to remember that the class of an object determines the **types** of the instance variables; however, the actual data is contained inside the individual objects, not the class. Thus, each object has its own set of data.

The static and the non-static portions of a class are very different things and serve very different purposes. Many classes contain only static members, or only non-static. However, it is possible to mix static and non-static members in a single class. By the way, static member variables and static member functions in a class are sometimes called class

variables and class methods, since they belong to the class itself, rather than to instances of that class.

Let's look at a specific example to see how it works. Consider this extremely simplified version of a Student class, which could be used to store information about students taking a course:

```
class Student {

  String name;                    // Student's name.
  double test1, test2, test3;   // Grades on three tests.

  double getAverage() {           // compute average test grade
    return (test1 + test2 + test3) / 3;
  }

}                                 // end of class Student
```

None of the members of this class are declared to be static, so the class exists only for creating objects. This class definition says that any object that is an instance of the Student class will include instance variables named name, test1, test2, and test3, and it will include an instance method named getAverage(). The names and tests in different objects will generally have different values. When called for a particular student, the method getAverage() will compute an average using **that student's** test grades. Different students can have different averages. (Again, this is what it means to say that an instance method belongs to an individual object, not to the class.)

In Java, a class is a **type**, similar to the built-in types such as int and boolean. So, a class name can be used to specify the type of a variable in a declaration statement, the type of a formal parameter, or the return type of a function. For example, a program could define a variable named std of type Student with the statement

```
Student std;
```

However, declaring a variable does **not** create an object! This is an important point, which is related to this Very Important Fact:

In Java, no variable can ever hold an object. A variable can only hold a reference to an object.

You should think of objects as floating around independently in the computer's memory. In fact, there is a portion of memory called the heap where objects live. Instead of holding an object itself, a variable holds the information necessary to find the object in memory. This information is called a reference or pointer to the object. In effect, a reference to an object is the address of the memory location where the object is stored. When you use a variable of class type, the computer uses the reference in the variable to find the actual object.

Objects are actually created by an operator called new, which creates an object and returns a reference to that object. For example, assuming that std is a variable of type Student, declared as above, the assignment statement

```
std = new Student();
```

would create a new object which is an instance of the class Student, and it would store a reference to that object in the variable std. The value of the variable is a reference to the object, not the object itself. It is not quite true, then, to say that the object is the "value of the variable std". It is certainly **not at all true** to say that the object is "stored in the variable std." The proper terminology is that "the variable std refers to the object," .

So, suppose that the variable std refers to an object belonging to the class Student. That object has instance variables name, test1, test2, and test3. These instance variables can be referred to as std.name, std.test1, std.test2, and std.test3. Similarly, std can be used to call the getAverage() instance method in the object by saying std.getAverage(). To print out the student's average, you could say:

```
System.out.println( "Your average is " + std.getAverage() );
```

More generally, you could use std.name any place where a variable of type String is legal. You can use it in expressions. You can assign a value to it. You can even use it to call

methods from the String class. For example, std.name.length() is the number of characters in the student's name.

It is possible for a variable like std, whose type is given by a class, to refer to no object at all. We say in this case that std holds a null reference. The null reference can be written in Java as "null". You could assign a null reference to the variable std by saying

```
std = null;
```

and you could test whether the value of std is null by testing

```
if (std == null) . . .
```

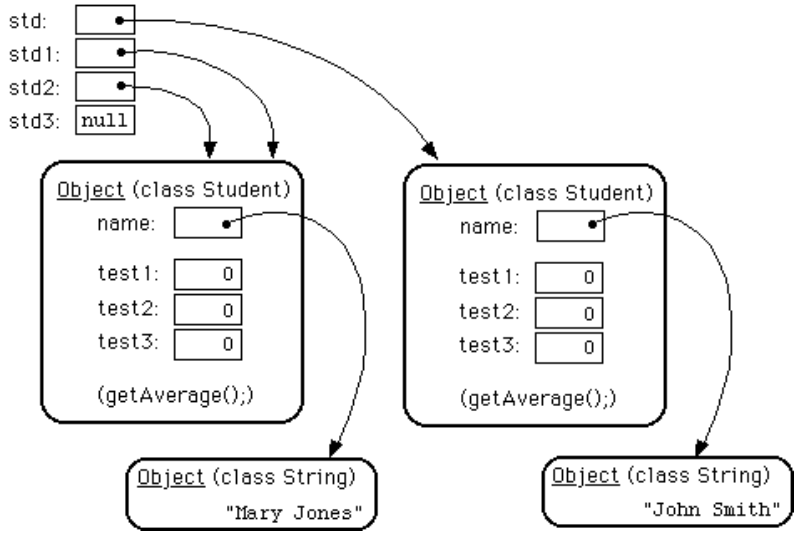
If the value of a variable is null, then it is, of course, illegal to refer to instance variables or instance methods through that variable -- since there is no object, and hence no instance variables to refer to. For example, if the value of the variable std is null, then it would be illegal to refer to std.test1. If your program attempts to use a null reference illegally like this, the result is an error called a null pointer exception.

Let's look at a sequence of statements that work with objects:

```
Student std, std1, std2, std3;        // Declare four variables of type Student.
    std = new Student();              // Create a new object belonging to the class Student, and    object in the //
                                      variable std.
                                       //   store a reference to that
  std1 = new Student();                // Create a second Student object
                                       //   and store a reference to
                                      //   it in the variable std1.
    std2 = std1;                      // Copy the reference value in std1
                                      //   into the variable std2.
    std3 = null;                       // Store a null reference in the
                                       //   variable std3.

    std.name = "John Smith";          // Set values of some instance variables.
    std1.name = "Mary Jones";         // (Other instance variables have default
```

```
                                    //    initial values of zero.)
```

After the computer executes these statements, the situation in the computer's memory looks like this:



This picture shows variables as little boxes, labeled with the names of the variables. Objects are shown as boxes with round corners. When a variable contains a reference to an object, the value of that variable is shown as an arrow pointing to the object. The variable std3, with a value of null, doesn't point anywhere. The arrows from std1 and std2 both point to the same object. This illustrates a Very Important Point:

When one object variable is assigned to another, only a reference is copied. The object referred to is not copied.

When the assignment "std2 = std1;" was executed, no new object was created. Instead, std2 is set to refer to the same object that std1 refers to. This has some consequences that might be surprising. For example, std1.name and std2.name refer to exactly the same variable, namely the instance variable in the object that both std1 and std2 refer to. After the string "Mary Jones" is assigned to the variable **std1**.name, it is also be true that the value of **std2**.name is "Mary Jones". "The object is not in the variable. The variable just holds a pointer to the object."

You can test objects for equality and inequality using the operators == and !=, but here again, the semantics are different from what you are used to. When you make a test "if (std1 == std2)", you are testing whether the values stored in std1 and std2 are the same. But the values are references to objects, not objects. So, you are testing whether std1 and std2 refer to the same object, that is, whether they point to the same location in memory. This is fine, if its what you want to do. But sometimes, what you want to check is whether the instance variables in the objects have the same values. To do that, you would need to ask whether "std1.test1 == std2.test1 && std1.test2 == std2.test2 && std1.test3 == std3.test1 && std1.name.equals(std2.name)"

Suppose that a variable that refers to an object is declared to be final. This means that the value stored in the variable can never be changed, once the variable has been initialized. The value stored in the variable is a reference to the object. So the variable will continue to refer to the same object as long as the variable exists. However, this does not prevent the data **in the object** from changing. The variable is final, not the object. It's perfectly legal to say

```
final Student stu = new Student();
stu.name = "John Doe";  // Change data in the object;
                / The value stored in stu is not changed.
```

Next, suppose that obj is a variable that refers to an object. Let's consider at what happens when obj is passed as an actual parameter to a method. The value of obj is assigned to a formal parameter in the method, and the method is executed. The method has no power to change the value stored in the variable, obj. It only has a copy of that value. However, that value is a reference to an object. Since the method has a reference to the object, it can change the data stored in the object. After the method ends, obj still points to the same object, but the data stored **in the object** might have changed. Suppose x is a variable of type int and stu is a variable of type Student. Compare:

```
void dontChange(int z) {
   z = 42;
}
```

```
void change(Student s) {
        s.name = "Fred";
}
```

The lines:

```
x = 17;
dontChange(x);
System.out.println(x);
```

output the value 17.

The value of x is not changed by the subroutine, which is equivalent to

```
z = x;
z = 42;
```

The lines:

```
stu.name = "Jane";
change(stu);
System.out.println(stu.name);
```

output the value "Fred".

The value of stu is not changed, but stu.name is. This is equivalent to

```
s = stu;
s.name = "Fred";
```

Constructors and Object Initialization

Object types in java are very different from the primitive types. Simply declaring a variable whose type is given as a class does not automatically create an object of that class. Objects must be explicitly constructed. For the computer, the process of constructing an object means, first, finding some unused memory in the heap that can be used to hold the object and, second, filling in the object's instance variables. As a programmer, you don't care where in memory the object is stored, but you will usually want to exercise some control over what initial values are stored in a new object's instance variables. In many cases, you will also want to do more complicated initialization or bookkeeping every time an object is created.

An instance variable can be assigned an initial value in its declaration, just like any other variable. For example, consider a class named PairOfDice. An object of this class will

represent a pair of dice. It will contain two instance variables to represent the numbers showing on the dice and an instance method for rolling the dice:

```
public class PairOfDice {


  public int die1 = 3;   // Number showing on the first die.
  public int die2 = 4;   // Number showing on the second die.


  public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
     die1 = (int)(Math.random()*6) + 1;
     die2 = (int)(Math.random()*6) + 1;
  }


} // end class PairOfDice
```

The instance variables die1 and die2 are initialized to the values 3 and 4 respectively. These initializations are executed whenever a PairOfDice object is constructed. It's important to understand when and how this happens. There can be many PairOfDice objects. Each time one is created, it gets its own instance variables, and the assignments "die1 = 3" and "die2 = 4" are executed to fill in the values of those variables. To make this clearer, consider a variation of the PairOfDice class:

```
public class PairOfDice {


  public int die1 = (int)(Math.random()*6) + 1;
  public int die2 = (int)(Math.random()*6) + 1;
  public void roll() {
     die1 = (int)(Math.random()*6) + 1;
     die2 = (int)(Math.random()*6) + 1;
  }
} // end class PairOfDice
```

Here, the dice are initialized to random values, as if a new pair of dice were being thrown onto the gaming table. Since the initialization is executed for each new object, a set of random initial values will be computed for each new pair of dice. Different pairs of dice can have different initial values. For initialization of **static** member variables, of course, the situation is quite different. There is only one copy of a static variable, and initialization of that variable is executed just once, when the class is first loaded.

If you don't provide any initial value for an instance variable, a default initial value is provided automatically. Objects are created with the operator, new. For example, a program that wants to use a PairOfDice object could say:

```
PairOfDice dice;   // Declare a variable of type PairOfDice.
dice = new PairOfDice();  // Construct a new object and store a
                          //   reference to it in the variable.
```

In this example, "new PairOfDice()" is an expression that allocates memory for the object, initializes the object's instance variables, and then returns a reference to the object. This reference is the value of the expression, and that value is stored by the assignment statement in the variable, dice. Part of this expression, "PairOfDice()", looks like function or method call, and that is no accident. It is, in fact, a call to a special type of method called a constructor. This might puzzle you, since there is no such method in the class definition. However, every class has a constructor. If the programmer doesn't provide one, then the system will provide a default constructor. This default constructor does nothing beyond the basics: allocate memory and initialize instance variables. If you want more than that to happen when an object is created, you can include one or more constructors in the class definition.

The definition of a constructor looks much like the definition of any other method, with three exceptions. A constructor does not have any return type (not even void). The name of the constructor must be the same as the name of the class in which it is defined. The only

modifiers that can be used on a constructor definition are the access modifiers public, private, and protected. (In particular, a constructor can't be declared static.)

However, a constructor does have a method body of the usual form, a block of statements. There are no restrictions on what statements can be used. And it can have a list of formal parameters. In fact, the ability to include parameters is one of the main reasons for using constructors. The parameters can provide data to be used in the construction of the object. For example, a constructor for the PairOfDice class could provide the values that are initially showing on the dice. Here is what the class would look like in that case:

```
public class PairOfDice {
        public int die1;   // Number showing on the first die.
   public int die2;   // Number showing on the second die.

   public PairOfDice(int val1, int val2) {
        // Constructor.  Creates a pair of dice that
        // are initially showing the values val1 and val2.
      die1 = val1;  // Assign specified values
      die2 = val2;  //              to the instance variables.
   }

   public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
      die1 = (int)(Math.random()*6) + 1;
      die2 = (int)(Math.random()*6) + 1;
   }

} // end class PairOfDice
```

The constructor is declared as "public PairOfDice(int val1, int val2)...", with no return type and with the same name as the name of the class. This is how the Java compiler recognizes a constructor. The constructor has two parameters, and values for these parameters must be provided when the constructor is called. For example, the expression "new PairOfDice(3,4)" would create a PairOfDice object in which the values of the instance variables die1 and die2 are initially 3 and 4. Of course, in a program, the value returned by the constructor should be used in some way, as in

```
PairOfDice dice;          // Declare a variable of type PairOfDice.
dice = new PairOfDice(1,1); // Let dice refer to a new PairOfDice
                            //   object that initially shows 1, 1.
```

Now that we've added a constructor to the PairOfDice class, we can no longer create an object by saying "new PairOfDice()"! The system provides a default constructor for a class **only** if the class definition does not already include a constructor. However, this is not a big problem, since we can add a second constructor to the class, one that has no parameters. In fact, you can have as many different constructors as you want, as long as their signatures are different, that is, as long as they have different numbers or types of formal parameters. In the PairOfDice class, we might have a constructor with no parameters which produces a pair of dice showing random numbers:

```
public class PairOfDice {

   public int die1;   // Number showing on the first die.
   public int die2;   // Number showing on the second die.

   public PairOfDice() {
         // Constructor.  Rolls the dice, so that they initially
         // show some random values.
      roll();  // Call the roll() method to roll the dice.
   }

   public PairOfDice(int val1, int val2) {
         // Constructor.  Creates a pair of dice that
         // are initially showing the values val1 and val2.
```

```
        die1 = val1;  // Assign specified values
        die2 = val2;  //           to the instance variables.
    }

    public void roll() {
        // Roll the dice by setting each of the dice to be
        // a random number between 1 and 6.
        die1 = (int)(Math.random()*6) + 1;
        die2 = (int)(Math.random()*6) + 1;
    }

} // end class PairOfDice
```

Now we have the option of constructing a PairOfDice object either with "new PairOfDice()" or with "new PairOfDice(x,y)", where x and y are int-valued expressions.

This class, once it is written, can be used in any program that needs to work with one or more pairs of dice. None of those programs will ever have to use the obscure incantation "(int)(Math.random()*6)+1", because it's done inside the PairOfDice class. And the programmer, having once gotten the dice-rolling thing straight will never have to worry about it again. Here, for example, is a main program that uses the PairOfDice class to count how many times two pairs of dice are rolled before the two pairs come up showing the same value:

```
public class RollTwoPairs {

    public static void main(String[] args) {

        PairOfDice firstDice;  // Refers to the first pair of dice.
        firstDice = new PairOfDice();

        PairOfDice secondDice; // Refers to the second pair of dice.
        secondDice = new PairOfDice();
```

```
        int countRolls;  // Counts how many times the two pairs of
                         //    dice have been rolled.

        int total1;      // Total showing on first pair of dice.
        int total2;      // Total showing on second pair of dice.

        countRolls = 0;

        do {  // Roll the two pairs of dice until totals are the same.

           firstDice.roll();    // Roll the first pair of dice.
           total1 = firstDice.die1 + secondDice.die2;   // Get total.
           System.out.println("First pair comes up  " + total1);

           secondDice.roll();    // Roll the first pair of dice.
           total2 = secondDice.die1 + secondDice.die2;   // Get total.
           System.out.println("Second pair comes up " + total2);

           countRolls++;   // Count this roll.

           System.out.println();  // Blank line.

        } while (total1 != total2);

        System.out.println("It took " + countRolls
                           + " rolls until the totals were the same.");

     } // end main()

} // end class RollTwoPairs
```

Constructors are methods, but they are subroutines of a special type. Unlike other methods, a constructor can only be called using the new operator, in an expression that has the form

new **class-name** (**parameter-list**)

where the **parameter-list** is possibly empty.

A constructor call is more complicated than an ordinary subroutine or function call. It is helpful to understand the exact steps that the computer goes through to execute a constructor call:

1. First, the computer gets a block of unused memory in the heap, large enough to hold an object of the specified type.
2. It initializes the instance variables of the object. If the declaration of an instance variable specifies an initial value, then that value is computed and stored in the instance variable. Otherwise, the default initial value is used.
3. The actual parameters in the constructor, if any, are evaluated, and the values are assigned to the formal parameters of the constructor.
4. The statements in the body of the constructor, if any, are executed.
5. A reference to the object is returned as the value of the constructor call.

The end result of this is that you have a reference to a newly constructed object. You can use this reference to get at the instance variables in that object or to call its instance methods.

Garbage Collection

In Java, the destruction of objects takes place automatically. An object exists on the heap, and it can be accessed only through variables that hold references to the object. What should be done with an object if there are no variables that refer to it? Such things can happen. Consider the following two statements (though in reality, you'd never do anything like this):

```
Student std = new Student("John Smith");
```

```
std = null;
```

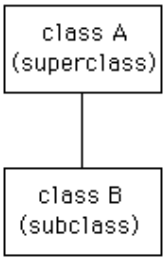
In the first line, a reference to a newly created Student object is stored in the variable std. But in the next line, the value of std is changed, and the reference to the Student object is gone. In fact, there are now no references whatsoever to that object stored in any variable. So there is no way for the program ever to use the object again. It might as well not exist. In fact, the memory occupied by the object should be reclaimed to be used for another purpose.

Java uses a procedure called garbage collection to reclaim memory occupied by objects that are no longer accessible to a program. It is the responsibility of the system, not the programmer, to keep track of which objects are "garbage". In the above example, it was very easy to see that the Student object had become garbage. Usually, it's much harder. If an object has been used for a while, there might be several references to the object stored in several variables. The object doesn't become garbage until all those references have been dropped.

In many other programing languages, it's the programmer's responsibility to delete the garbage. Unfortunately, keeping track of memory usage is very error-prone, and many serious program bugs are caused by such errors. A programmer might accidently delete an object even though there are still references to that object. This is called a dangling pointer error, and it leads to problems when the program tries to access an object that is no longer there. Another type of error is a memory leak, where a programmer neglects to delete objects that are no longer in use. This can lead to filling memory with objects that are completely inaccessible, and the program might run out of memory even though, in fact, large amounts of memory are being wasted. Because Java uses garbage collection, such errors are simply impossible.

Inheritance, Polymorphism, and Abstract Classes

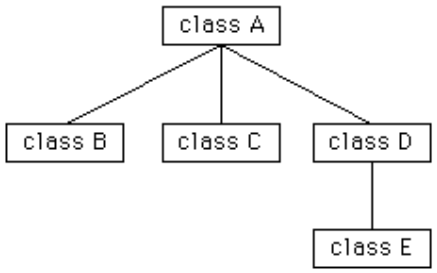
A class represents a set of objects which share the same structure and behaviors. The class determines the structure of objects by specifying variables that are contained in each instance of the class, and it determines behavior by providing the instance methods that express the behavior of the objects. However, something like this can be done in most programming languages. The central new idea in object-oriented programming -- the idea that really distinguishes it from traditional programming -- is to allow classes to express the similarities among objects that share **some**, but not all, of their structure and behavior. Such similarities can expressed using inheritance and polymorphism.



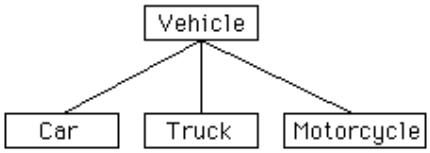
The term inheritance refers to the fact that one class can inherit part or all of its structure and behavior from another class. The class that does the inheriting is said to be a subclass of the class from which it inherits. If class B is a subclass of class A, we also say that class A is a superclass of class B. (Sometimes the terms derived class and base class are used instead of subclass and superclass.) A subclass can add to the structure and behavior that it inherits. It can also replace or modify inherited behavior (though not inherited structure). The relationship between subclass and superclass is sometimes shown by a diagram in which the subclass is shown below, and connected to, its superclass.

In Java, when you create a new class, you can declare that it is a subclass of an existing class. If you are defining a class named "B" and you want it to be a subclass of a class named "A", you would write

```
class B extends A {
   .
   .  // additions to, and modifications of,
   .  // stuff inherited from class A
   .
}
```



Several classes can be declared as subclasses of the same superclass. The subclasses, which might be referred to as "sibling classes," share some structures and behaviors -- namely, the ones they inherit from their common superclass. The superclass expresses these shared structures and behaviors. In the diagram to the left, classes B, C, and D are sibling classes. Inheritance can also extend over several "generations" of classes. This is shown in the diagram, where class E is a subclass of class D which is itself a subclass of class A. In this case, class E is considered to be a subclass of class A, even though it is not a direct subclass.



Let's look at an example. Suppose that a program has to deal with motor vehicles, including cars, trucks, and motorcycles. (This might be a program used by a Department of Motor Vehicles to keep track of registrations.) The program could use a class named Vehicle to represent all types of vehicles. The Vehicle class could include instance variables such as registrationNumber and owner and instance methods such as transferOwnership(). These are variables and methods common to all vehicles. Three subclasses of Vehicle -- Car, Truck, and Motorcycle -- could then be used to hold variables and methods specific to particular types of vehicles. The Car class might add an instance variable numberOfDoors, the Truck class might have numberOfAxels, and the Motorcycle class could have a boolean variable hasSidecar. (Well, it could in theory at least, even if it might give a chuckle to the people at the Department of Motor Vehicles.) The declarations of these classes in Java program would look, in outline, like this:

```
class Vehicle {
  int registrationNumber;
  Person owner;  // (assuming that a Person class has been defined)
  void transferOwnership(Person newOwner) {
     . . .
  }
  . . .
}
class Car extends Vehicle {
  int numberOfDoors;
  . . .
```

```
}
class Truck extends Vehicle {
  int numberOfAxels;
  . . .
}
class Motorcycle extends Vehicle {
  boolean hasSidecar;
  . . .
}
```

Suppose that myCar is a variable of type Car that has been declared and initialized with the statement

```
Car myCar = new Car();
```

Given this declaration, a program could to refer to myCar.numberOfDoors, since numberOfDoors is an instance variable in the class Car. But since class Car extends class Vehicle, a car also has all the structure and behavior of a vehicle. This means that myCar.registrationNumber, myCar.owner, and myCar.transferOwnership() also exist.

Now, in the real world, cars, trucks, and motorcycles are in fact vehicles. The same is true in a program. That is, an object of type Car or Truck or Motorcycle is automatically an object of type Vehicle. This brings us to the following Important Fact:

A variable that can hold a reference to an object of class A can also hold a reference to an object belonging to any subclass of A.

The practical effect of this in our example is that an object of type Car can be assigned to a variable of type Vehicle. That is, it would be legal to say

```
Vehicle myVehicle = myCar;
```

or even

```
Vehicle myVehicle = new Car();
```

After either of these statements, the variable myVehicle holds a reference to a Vehicle object that happens to be an instance of the subclass, Car. The object "remembers" that it is in fact a Car, and not **just** a Vehicle. Information about the actual class of an object is stored as part of that object. It is even possible to test whether a given object belongs to a given class, using the instanceof operator. The test:

```
if (myVehicle instanceof Car) ...
```

determines whether the object referred to by myVehicle is in fact a car.

On the other hand, if myVehicle is a variable of type Vehicle the assignment statement

```
myCar = myVehicle;
```

would be illegal because myVehicle could potentially refer to other types of vehicles that are not cars. The computer will not allow you to assign an int value to a variable of type short, because not every int is a short. Similarly, it will not allow you to assign a value of type Vehicle to a variable of type Car because not every vehicle is a car. As in the case of ints and shorts, the solution here is to use type-casting. If, for some reason, you happen to know that myVehicle does in fact refer to a Car, you can use the type cast (Car)myVehicle to tell the computer to treat myVehicle as if it were actually of type Car. So, you could say

```
myCar = (Car)myVehicle;
```

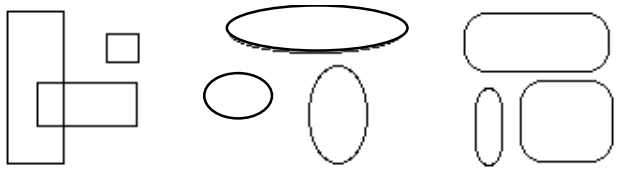
and you could even refer to ((Car)myVehicle).numberOfDoors. As an example of how this could be used in a program, suppose that you want to print out relevant data about a vehicle. You could say:

```
 System.out.println("Vehicle Data:");
System.out.println("Registration number:  "
                        + myVehicle.registrationNumber);
if (myVehicle instanceof Car) {
   System.out.println("Type of vehicle:  Car");
   Car c;
   c = (Car)myVehicle;
   System.out.println("Number of doors:  " + c.numberOfDoors);
}
else if (myVehicle instanceof Truck) {
   System.out.println("Type of vehicle:  Truck");
```

```
    Truck t;
    t = (Truck)myVehicle;
    System.out.println("Number of axels:  " + t.numberOfAxels);
  }
  else if (myVehicle instanceof Motorcycle) {
    System.out.println("Type of vehicle:  Motorcycle");
    Motorcycle m;
    m = (Motorcycle)myVehicle;
    System.out.println("Has a sidecar:    " + m.hasSidecar);
  }
```

Note that for object types, when the computer executes a program, it checks whether type-casts are valid. So, for example, if myVehicle refers to an object of type Truck, then the type cast (Car)myVehicle will produce an error.

As another example, consider a program that deals with shapes drawn on the screen. Let's say that the shapes include rectangles, ovals, and roundrects of various colors.



Three classes, Rectangle, Oval, and RoundRect, could be used to represent the three types of shapes. These three classes would have a common superclass, Shape, to represent features that all three shapes have in common. The Shape class could include instance variables to represent the color, position, and size of a shape. It could include instance methods for changing the color, position, and size of a shape. Changing the color, for example, might involve changing the value of an instance variable, and then redrawing the shape in its new color:

```
class Shape {


  Color color;   // Color of the shape.  (Recall that class Color
                 // is defined in package java.awt.  Assume
                 // that this class has been imported.)
```

```
    void setColor(Color newColor) {
        // Method to change the color of the shape.
      color = newColor; // change value of instance variable
      redraw(); // redraw shape, which will appear in new color
    }

    void redraw() {
        // method for drawing the shape
      ? ? ?  // what commands should go here?
    }

    . . .         // more instance variables and methods

} // end of class Shape
```

Now, you might see a problem here with the method redraw(). The problem is that each different type of shape is drawn differently. The method setColor() can be called for any type of shape. How does the computer know which shape to draw when it executes the redraw()? Informally, we can answer the question like this: The computer executes redraw() by asking the shape to redraw **itself**. Every shape object knows what it has to do to redraw itself.

In practice, this means that each of the specific shape classes has its own redraw() method:

```
class Rectangle extends Shape {
  void redraw() {
    . . .  // commands for drawing a rectangle
  }
  . . . // possibly, more methods and variables
}
class Oval extends Shape {
  void redraw() {
    . . .  // commands for drawing an oval
```

```
    }
    . . . // possibly, more methods and variables
}
class RoundRect extends Shape {
    void redraw() {
        . . .  // commands for drawing a rounded rectangle
    }
    . . . // possibly, more methods and variables
}
```

If oneShape is a variable of type Shape, it could refer to an object of any of the types, Rectangle, Oval, or RoundRect. As a program executes, and the value of oneShape changes, it could even refer to objects of different types at different times! Whenever the statement

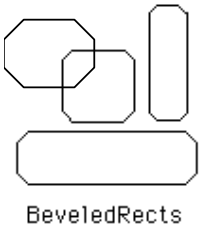
```
oneShape.redraw();
```

is executed, the redraw method that is actually called is the one appropriate for the type of object to which oneShape actually refers. There may be no way of telling, from looking at the text of the program, what shape this statement will draw, since it depends on the value that oneShape happens to have when the program is executed. Even more is true. Suppose the statement is in a loop and gets executed many times. If the value of oneShape changes as the loop is executed, it is possible that the very same statement "oneShape.redraw();" will call different methods and draw different shapes as it is executed over and over. We say that the redraw() method is polymorphic. A method is polymorphic if the action performed by the method depends on the actual type of the object to which the method is applied. Polymorphism is one of the major distinguishing features of object-oriented programming.

In object-oriented programming, calling a method is often referred to as sending a message to an object. The object responds to the message by executing the appropriate method. The statement "oneShape.redraw();" is a message to the object referred to by oneShape. Since that object knows what type of object it is, it knows how it should respond to the message. From this point of view, the computer always executes "oneShape.redraw();" in the same way: by sending a message. The response to the message depends, naturally, on who receives it. From this point of view, objects are active entities that send and receive messages,

and polymorphism is a natural, even necessary, part of this view. Polymorphism just means that different objects can respond to the same message in different ways.

One of the most beautiful things about polymorphism is that it lets code that you write do things that you didn't even conceive of, at the time you wrote it. If for some reason, you decide that you want to add beveled rectangles to the types of shapes your program can deal with, you can write a new subclass, BeveledRect, of class Shape and give it its own redraw() method. Automatically, code that we wrote previously -- such as the statement oneShape.redraw() -- can now suddenly start drawing beveled rectangles, even though the beveled rectangle class didn't exist when we wrote the statement!



In the statement "oneShape.redraw();", the redraw message is sent to the object oneShape. Look back at the method from the Shape class for changing the color of a shape:

```
void setColor(Color newColor) {
  color = newColor; // change value of instance variable
  redraw(); // redraw shape, which will appear in new color
}
```

A redraw message is sent here, but which object is it sent to? Well, the setColor method is itself a message that was sent to some object. The answer is that the redraw message is sent to that same object, the one that received the setColor message. If that object is a rectangle, then it is the redraw() method from the Rectangle class that is executed. If the object is an oval, then it is the redraw() method from the Oval class. This is what you should expect, but it means that the redraw(); statement in the setColor() method does **not** necessarily call the redraw() method in the Shape class! The redraw() method that is executed could be in any subclass of Shape.

Again, this is not a real surprise if you think about it in the right way. Remember that an instance method is always contained in an object. The class only contains the source code for the method. When a Rectangle object is created, it contains a redraw() method. The source

code for that method is in the Rectangle class. The object also contains a setColor() method. Since the Rectangle class does not define a setColor() method, the source code for the rectangle's setColor() method comes from the superclass, Shape. But even though the source codes for the two methods are in different classes, the methods themselves are part of the same object. When the rectangle's setColor() method is executed and calls redraw(), the redraw() method that is executed is the one in the same object.

Whenever a Rectangle, Oval, or RoundRect object has to draw itself, it is the redraw() method in the appropriate class that is executed. This leaves open the question, What does the redraw() method in the Shape class do? How should it be defined?

The answer may be surprising: We should leave it blank! The fact is that the class Shape represents the abstract idea of a shape, and there is no way to draw such a thing. Only particular, concrete shapes like rectangles and ovals can be drawn. So, why should there even be a redraw() method in the Shape class? Well, it has to be there, or it would be illegal to call it in the setColor() method of the Shape class, and it would be illegal to write "oneShape.redraw();", where oneShape is a variable of type Shape. The computer would say, oneShape is a variable of type Shape and there's no redraw() method in the Shape class.

Nevertheless the version of redraw() in the Shape class will never be called. In fact, if you think about it, there can never be any reason to construct an actual object of type Shape! You can have **variables** of type Shape, but the objects they refer to will always belong to one of the subclasses of Shape. We say that Shape is an abstract class. An abstract class is one that is not used to construct objects, but only as a basis for making subclasses. An abstract class exists **only** to express the common properties of all its subclasses.

Similarly, we could say that the redraw() method in class Shape is an abstract method, since it is never meant to be called. In fact, there is nothing for it to do -- any actual redrawing is done by redraw() methods in the subclasses of Shape. The redraw() method in Shape has to be there. But it is there only to tell the computer that all Shapes understand the redraw

message. As an abstract method, it exists merely to specify the common interface of all the actual, concrete versions of redraw() in the subclasses of Shape. There is no reason for the abstract redraw() in class Shape to contain any code at all.

Shape and its redraw() method are semantically abstract. You can also tell the computer, syntactically, that they are abstract by adding the modifier "abstract" to their definitions. For an abstract method, the block of code that gives the implementation of an ordinary method is replaced by a semicolon. An implementation must be provided for the abstract method in any concrete subclass of the abstract class. Here's what the Shape class would look like as an abstract class:

```
abstract class Shape {
Color color;   // color of shape.
      void setColor(Color newColor) {
      // method to change the color of the shape
    color = newColor; // change value of instance variable
    redraw(); // redraw shape, which will appear in new color
  }
  abstract void redraw();
      // abstract method -- must be defined in
      // concrete subclasses

  . . .        // more instance variables and methods

} // end of class Shape
```

Once you have done this, it becomes illegal to try to create actual objects of type Shape, and the computer will report an error if you try to do so.

In Java, every class that you declare has a superclass. If you don't specify a superclass, then the superclass is automatically taken to be Object, a predefined class that is part of the package java.lang. (The class Object itself has no superclass, but it is the only class that has this property.) Thus,

```
class myClass { . . .
```

is exactly equivalent to

```
class myClass extends Object { . . .
```

Every other class is, directly or indirectly, a subclass of Object. This means that any object, belonging to any class whatsoever, can be assigned to a variable of type Object. The class Object represents very general properties that are shared by all objects, belonging to any class. Object is the most abstract class of all!

The Object class actually finds a use in some cases where objects of a very general sort are being manipulated. For example, java has a standard class, java.util.Vector, that represents a list of Objects. (The Vector class is in the package java.util. If you want to use this class in a program you write, you would ordinarily use an import statement to make it possible to use the short name, Vector, instead of the full name, java.util.Vector.) The Vector class is very convenient, because a Vector can hold any number of objects, and it will grow, when necessary, as objects are added to it. Since the items in the list are of type Object, the list can actually hold objects of any type.

A program that wants to keep track of various Shapes that have been drawn on the screen can store those shapes in a Vector. Suppose that the Vector is named listOfShapes. A shape, oneShape, can be added to the end of the list by calling the instance method "listOfShapes.addElement(oneShape);". The shape could be removed from the list with "listOfShapes.removeElement(oneShape);". The number of shapes in the list is given by the function "listOfShapes.size()". And it is possible to retrieve the i-th object from the list with the function call "listOfShapes.elementAt(i)". (Items in the list are numbered from 0 to listOfShapes.size() - 1.) However, note that this method returns an Object, not a Shape. (Of course, the people who wrote the Vector class didn't even know about Shapes, so the method they wrote could hardly have a return type of Shape!) Since you know that the items in the list are, in fact, Shapes and not just Objects, you can type-cast the Object returned by listOfShapes.elementAt(i) to be a value of type Shape:

```
oneShape = (Shape)listOfShapes.elementAt(i);
```

Let's say, for example, that you want to redraw all the shapes in the list. You could do this with a simple for loop, which is lovely example of object-oriented programming and polymorphism:

```
for (int i = 0; i < listOfShapes.size(); i++) {
  Shape s;  // i-th element of the list, considered as a Shape
  s = (Shape)listOfShapes.elementAt(i);
  s.redraw();
}
```

More Details of Classes

Extending Existing Classes

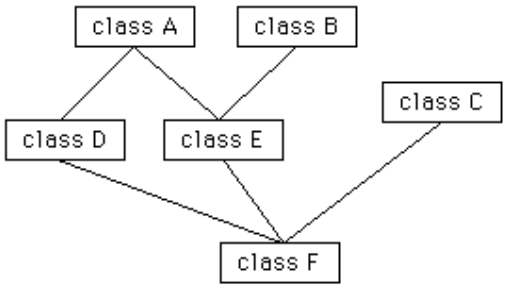
In day-to-day programming, especially for programmers who are just beginning to work with objects, subclassing is used mainly in one situation. There is an existing class that can be adapted with a few changes or additions. This is much more common than designing groups of classes and subclasses from scratch. The existing class can be extended to make a subclass. The syntax for this is

```
class subclass-name extends existing-class-name {
  .
  .  // Changes and additions.
  .
}
```

(Of course, the class can optionally be declared to be public.)

Interfaces

Some object-oriented programming languages, such as C++, allow a class to extend two or more superclasses. This is called multiple inheritance. In the illustration below, for example, class E is shown as having both class A and class B as direct superclasses, while class F has three direct superclasses.



Multiple Inheritance (NOT allowed in Java)

Such multiple inheritance is **not** allowed in Java. The designers of Java wanted to keep the language reasonably simple, and felt that the benefits of multiple inheritance were not worth the cost in increased complexity. However, Java does have a feature that can be used to accomplish many of the same goals as multiple inheritance: interfaces.

The interface of a Method consists of the name of the method, its return type, and the number and types of its parameters. This is the information you need to know if you want to call the method. A method also has an implementation: the block of code which defines it and which is executed when the subroutine is called.

In Java, interface is a reserved word with an additional meaning. An "interface" in Java consists of a set of subroutine interfaces, without any associated implementations. A class can implement an interface by providing an implementation for each of the subroutines specified by the interface. Here is an example of a very simple Java interface:

```
public interface Drawable {
  public void draw();
}
```

This looks much like a class definition, except that the implementation of the method draw() is omitted. A class that implements the interface, Drawable, must provide an implementation for this method. Of course, the class can also include other methods and variables. For example,

```
class Line implements Drawable {
  public void draw() {
```

```
        . . . // do something -- presumably, draw a line
    }
    . . . // other methods and variables
}
```

While a class can **extend** only one other class, it can **implement** any number of interfaces. In fact, a class can both extend another class and implement one or more interfaces. So, we can have things like

```
class FilledCircle extends Circle
                implements Drawable, Fillable {
  . . .
}
```

An interface is very much like an abstract class, that is, a class that can never be used for constructing objects, but can be used as a basis for building other classes. The methods in an interface are abstract methods, which must be implemented in any concrete class that implements the interface. And as with abstract classes, even though you can't construct an object from an interface, you can declare a variable whose type is given by the interface. For example, if Drawable is an interface, and if Line and FilledCircle are classes that implement Drawable, then you could say:

```
Drawable figure;  // Declare a variable of type Drawable.  It can
                  //    refer to any object that implements the
                  //    Drawable interface.

figure = new Line();  // figure now refers to an object of class Line
figure.draw();   // calls draw() method from class Line

figure = new FilledCircle();   // Now, figure refers to an object
                        //   of class FilledCircle.
figure.draw();   // calls draw() method from class FilledCircle
```

A variable of type Drawable can refer to any object of any class that implements the Drawable interface. A statement like figure.draw(), above, is legal because any such class has a draw() method.

Note that a type is something that can be used to declare variables. A type can also be used to specify the type of a parameter in a method, or the return type of a function. In Java, a type can be either a class, an interface, or one of the eight built-in primitive types. These are the only possibilities. Of these, however, only classes can be used to construct new objects.

The Special Variables this and super

A static member of a class has a simple name, which can only be used inside the class definition. For use outside the class, it has a full name of the form **class-name**.**simple-name**. For example, "System.out" is a static member variable with simple name "out" in the class "System". It's always legal to use the full name of a static member, even within the class where it's defined. Sometimes it's even necessary, as when the simple name of a static member variable is hidden by a local variable of the same name.

Instance variables and instance methods also have simple names that can be used inside the class where the variable or method is defined. But a class does not actually contain instance variables or methods, only their source code. Actual instance variables and methods are contained in objects. To get at an instance variable or method from outside the class definition, you need a variable that refers to the object. Then the full name is of the form **variable-name**.**simple-name**. But suppose you are writing a class definition, and you want to refer to the object that contains the instance method you are writing? Suppose you want to use a full name for an instance variable, because its simple name is hidden by a local variable?

Java provides a special, predefined variable named "this" that you can use for these purposes. The variable, this, can be used in the source code of an instance method to refer to

the object that contains the method. If x is an instance variable, then this.x can be used as a full name for that variable. Whenever the computer executes an instance method, it sets the variable, this, to refer to the object that contains the method.

One common use of this is in constructors. For example:

```
public class Student {

  private String name;  // Name of the student.

  public Student(String name) {
     // Constructor.  Create a student with specified name.
    this.name = name;
  }
   .
   .   // More variables and methods.
   .
}
```

In the constructor, the instance variable called name is hidden by a formal parameter. However, the instance variable can still be referred to by its full name, this.name. In the assignment statement, the value of the formal parameter, name, is assigned to the instance variable, this.name. This is considered to be acceptable style: There is no need to dream up cute new names for formal parameters that are just used to initialize instance variables. You can use the same name for the parameter as for the instance variable.

There is another common use for this. Sometimes, when you are writing an instance method, you need to pass the object that contains the method to a subroutine, as an actual parameter. In that case, you can use this as the actual parameter. For example, if you wanted to print out a string representation of the object, you could say "System.out.println(this);".

Java also defines another special variable, named "super", for use in the definitions of instance methods. The variable super is for use in a subclass. Like this, super refers to the object that contains the method. But it's forgetful. It forgets that the object belongs to the class you

are writing, and it remembers only that it belongs to the superclass of that class. The point is that the class can contain additions and modifications to the superclass. super doesn't know about any of those additions and modifications. Let's say that the class that you are writing contains an instance method named doSomething(). Consider the subroutine call statement super.doSomething(). Now, super doesn't know anything about the doSomething() method in the subclass. It only knows about things in the superclass, so it tries to execute a method named doSomething() from the superclass. If there is none -- if the doSomething() method was an addition rather than a modification -- you'll get a syntax error.

The reason super exists is so you can get access to things in the superclass that are hidden by things in the subclass. For example, super.x always refers to an instance variable named x in the superclass. This can be useful for the following reason: If a class contains an instance variable with the same name as an instance variable in its superclass, then an object of that class will actually contain two variables with the same name: one defined as part of the class itself and one defined as part of the superclass. The variable in the subclass does not **replace** the variable of the same name in the superclass; it merely **hides** it. The variable from the superclass can still be accessed, using super.

When you write a method in a subclass that has the same signature as a method in its superclass, the method from the superclass is hidden in the same way. We say that the method in the subclass overrides the method from the superclass. Again, however, super can be used to access the method from the superclass.

The major use of super is to override a method with a new method that **extends** the behavior of the inherited method, instead of **replacing** that behavior entirely. The new method can use super to call the method from the superclass, and then it can add additional code to provide additional behavior. As an example, suppose you have a PairOfDice class that includes a roll() method. Suppose that you want a subclass, GraphicalDice, to represent a pair of dice drawn on the computer screen. The roll() method in the GraphicalDice method should

change the values of the dice and redraw the dice to show the new values. The GraphicalDice class might look something like this:

```
public class GraphicalDice extends PairOfDice {


  public void roll() {
        // Roll the dice, and redraw them.
     super.roll();  // Call the roll method from PairOfDice.
     redraw();      // Call a method to draw the dice.
  }
    .
    .  // More stuff, including definition of redraw().
    .
}
```

Constructors in Subclasses

Constructors are not inherited. That is, if you extend an existing class to make a subclass, the constructors in the superclass do not become part of the subclass. If you want constructors in the subclass, you have to define new ones from scratch. If you don't define any constructors in the subclass, then the computer will make up a default constructor, with no parameters, for you. This could be a problem, if there is a constructor in the superclass that does a lot of necessary work. It looks like you might have to repeat all that work in the subclass! This could be a **real** problem if you don't have the source code to the superclass, and don't know how it works, or if the constructor in the superclass initializes private member variables that you don't even have access to in the subclass!

Obviously, there has to be some fix for this, and there is. It involves the special variable, super, which was introduced in the previous subsection. As the very first statement in a constructor, you can use super to call a constructor from the superclass. The notation for this is a bit ugly and misleading, and it can only be used in this one particular circumstance: It looks like you are calling super as a subroutine (even though super is not a subroutine and you

can't call constructors the same way you call other subroutines anyway). As an example, assume that the PairOfDice class has a constructor that takes two integers as parameters. Consider a subclass:

```
public class GraphicalDice extends PairOfDice {

   public GraphicalDice() {  // Constructor for this class.

      super(3,4);  // Call the constructor from the
                   //   PairOfDice class, with parameters 3, 4.

      initializeGraphics();  // Do some initialization specific
                             //   to the GraphicalDice class.
   }
     .
     .  // More constructors, methods, variables...
     .
}
```

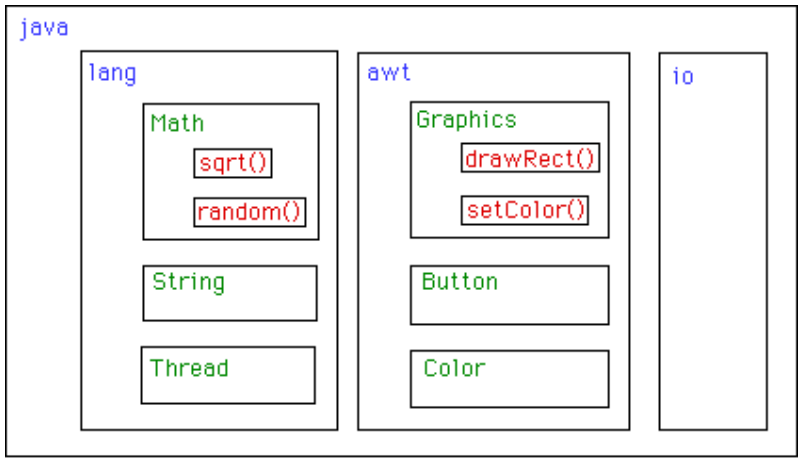
You can use the special variable this in exactly the same way to call another constructor in the same class. This can be useful since it can save you from repeating the same code in several constructors.

Packages

To provide larger-scale organization, classes in Java can be grouped into packages. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented as one large package, which is named "java". The java package, in turn, is made up of several other packages, and each of those packages contains a number of classes.

The java package includes several other sub-packages, such as java.io, which provides facilities for input/output, java.net, which deals with network communication, and java.applet,

which implements the basic functionality of applets. The most basic package is called java.lang, which includes fundamental classes such as String and Math.



Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt()

Let's say that you want to use the class java.awt.Button in a program that you are writing. One way to do this is to use the full name of the class. For example,

```
java.awt.Button  stopBttn;
```

to declare a variable named stopBttn whose type is java.awt.Button. Of course, this can get tiresome, so Java makes it possible to avoid using the full names of classes. If you put

```
import java.awt.Button;
```

at the beginning of your program, before you start writing any class, then you can abbreviate the full name java.awt.Button to just the name of the class, Button. This would allow you to say just

```
Button  stopBttn;
```

to declare the variable stopBttn. (The only effect of the import statement is to allow you to use simple class names instead of full "package.class" names; you aren't really importing anything substantial. If you leave out the import statement, you can still access the class -- you just have to use its full name.) There is a shortcut for importing all the classes from a given package. You can import all the classes from java.awt by saying

```
import java.awt.*;
```

In fact, any Java program that uses a graphical user interface is likely to begin with this line. A program might also include lines such as "import java.net.*;" or "import java.io.*;" to get easy access to networking and input/output classes. Because the package java.lang is so fundamental, all the classes in java.lang are automatically imported into every program. It's as if every program began with the statement "import java.lang.*;". This is why we have been able to use the class name String instead of java.lang.String, and Math.sqrt() instead of java.lang.Math.sqrt(). It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named utilities. Then the source code file that defines those classes must begin with the line "package utilities;". Any program that uses the classes should include the directive "import utilities.*;" to obtain access to all the classes in the utilities package. Unfortunately, things are a little more complicated than this. Remember that if a program uses a class, then the class must be "available" when the program is compiled and when it is executed. Exactly what this means depends on which Java environment you are using. Most commonly, classes in a package named utilities should be in a directory with the name utilities, and that directory should be located in the same place as the program that uses the classes. If a class is not specifically placed in a package, then it is put in something called the default package, which has no name.

More about Access Modifiers

private

The most restrictive access level is private. A private member is accessible only to the class in which it is defined. Use this access to declare members that should only be used by the class.

protected

The next access level specifier is protected, which allows the class itself, subclasses, and all classes in the same package to access the members.

public

Any class, in any package, has access to a class's public members. Declare public members only if such access cannot produce undesirable results if an outsider uses them.

Package

The package access level is what you get if you don't explicitly set a member's access to one of the other levels. This access level allows classes in the same package as your class to access the members. This level of access assumes that classes in the same package are trusted friends.

static

Java technology defines away of sharing sharing data between objects, on a class scope basis. Such data is not 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.

final variables :

You can declare a variable in any scope to be final. The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other programming languages.

Ex. final int aFinalVar = 0;

The previous statement declares a final variable and initializes it, all at once. Subsequent attempts to assign a value to aFinalVar result in a compiler error. You may, if necessary, defer initialization of a final local variable. Simply declare the local variable and initialize it later, like this:

```
final int blankfinal;
. . .
blankfinal = 0;
```

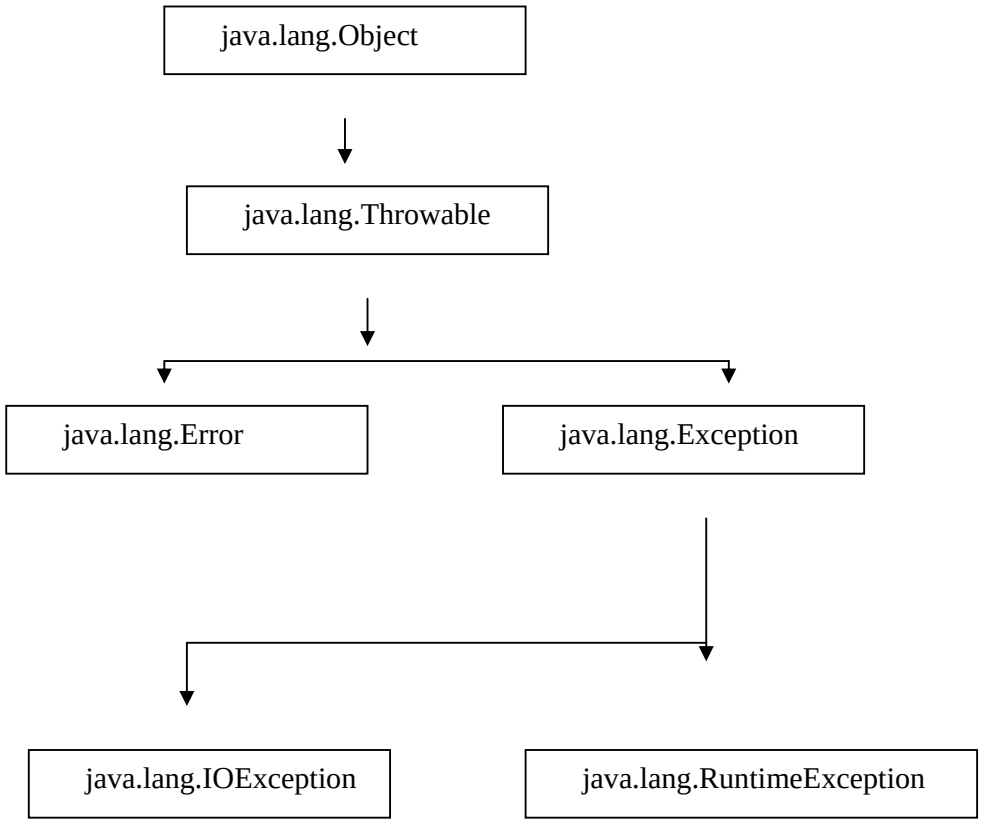
volatile

The volatile keyword is used to prevent the compiler from performing certain optimizations on a member.

1.6 JAVA EXCEPTIONS

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. Any abnormal condition that disturbs the normal program flow while the program is in execution is an error or an exception. The errors are serious conditions from which we better avoid recovery of the program. Exceptions are mild error conditions and rather than terminating the program we can handle such exceptions and continue program execution.

- **User input errors -** This is some incorrect data entered by the user
- **Device error -** The hardware may have some problems which need to be reported to the user. For example, printer may be off, printer out of paper etc...
- **Physical limitation -** Program runs out of available memory or Hard disk is full.
- **Code errors -** A method may not perform correctly.



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. The Throwable class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or of one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement. Similarly, only this class or one of its subclasses can be the argument type in a catch clause.

An Error is a subclass of Throwable that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions. The ThreadDeath error, though a "normal" condition, is also a subclass of Error because most applications should not try to catch it.

The class Exception and its subclasses are a form of Throwable that indicates conditions that a reasonable application might want to catch. RuntimeException is the superclass of those exceptions that can be thrown during the normal operation of the Java Virtual Machine. A method is not required to declare in its throws clause any subclasses of RuntimeException that might be thrown during the execution of the method but not caught. IOException signals that an I/O exception of some sort has occurred. This class is the general class of exceptions produced by failed or interrupted I/O operations.

Error indicates a severe problem from which recovery is difficult, if not impossible eg. running out of memory where as RuntimeException indicates a design or implementation error. When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not catch Errors. In addition, it's unlikely that typical Java programs will ever throw Errors either.

Runtime exceptions represent problems that are detected by the runtime system. This includes arithmetic exceptions (such as when dividing by zero), pointer exceptions (such as trying to access an object through a null reference), and indexing exceptions (such as attempting to access an array element through an index that is too large or too small).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. Typically, the cost of checking for runtime exceptions exceeds the benefit of catching or specifying them. Thus the compiler does not require that you catch or specify runtime exceptions, although you can. Because runtime exceptions are so ubiquitous and attempting to catch or specify all of them all the time would be a fruitless exercise (and a fruitful source of unreadable and unmaintainable code), the compiler allows runtime exceptions to go uncaught and unspecified.

A method can detect and throw a RuntimeException when it's encountered an error in the virtual machine runtime. However, it's typically easier to just let the virtual machine detect and throw it. Normally, the methods you write should throw Exceptions, not RuntimeException.

Similarly, you create a subclass of RuntimeException when you are creating an error in the virtual machine runtime (which you probably aren't). Otherwise you should subclass Exception. Do not throw a runtime exception or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Checked Exceptions

ClassNotFoundException	Class not found
InstantiationException	Attempt to create an object of an abstract class or interface.
IllegealAccessException	Access to a class is denied.
NoSuchMethodException	A requested method does not exsist.
NoSuchFieldException	A requested field does not exsist
InterruptedException	One thread has been interrupted by another thread.
CloneNotSupportedException	Attempt to clone an object that does not

	implement Cloneable interface

Checked exceptions represent useful information about the operation of a legally specified request that the caller may have had no control over and that the caller needs to be informed about--for example, the file system is now full, or the remote end has closed the connection, or the access privileges don't allow this action.

Unchecked Exceptions

ArithmeticException	Arithmetic error, such as divide by zero.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
IllegealArgumentException	Illegeal argument used to invoke a method.
ClassCastException	Invalid class cast

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. The cost of checking for runtime exceptions often exceeds the benefit of catching or specifying them. Thus the compiler does not require that you catch or specify runtime exceptions, although you can. Checked exceptions are exceptions that are not runtime exceptions and are checked by the compiler; the compiler checks that these exceptions are caught or specified.

The Throwable Class

By convention, class Throwable and its subclasses have two constructors, one that takes no arguments and one that takes a String argument that can be used to produce an error message. A Throwable class contains a snapshot of the execution stack of its thread at the time it was created. It can also contain a message string that gives more information about the error.

fillInStackTrace()

Fills in the execution stack trace. This method records within this Throwable object information about the current state of the stack frames for the current thread. This method is useful when an application is re-throwing an error or exception.

fillInStackTrace()

Fills in the execution stack trace. This method records within this Throwable object information about the current state of the stack frames for the current thread. This method is useful when an application is re-throwing an error or exception.

getMessage()

Returns the errort message string of this throwable object.

Throwable fiilInStackTrace()	Returns a Throwable object containing a complete stack trace.
void printStackTrace()	Displays the stack trace
String toString()	Returns a String object containing a description of the exception, called by println().
String getMessage()	Returns a description of the exception.

```
public class Demo        {
        public static void main(String [] args)     {
                int i = 0;
                String greetings [] = { “Hello”,”Hi”,”Bye” };

                while ( i < 4 )     {
```

```
                System.out.println(greetings[i]);
                i++;
            }
        }
}
```

Note that the continuation condition, "i < A.length", implies that the last value of i that is actually processed is greetings.length-1, which is the index of the final item in the array. It's important to use "<" here, not "<=", since "<=" would give an array out of bounds error.

- try

  use the try statement with the code that might throw an exception.

- catch

  use the catch statement to specify the exception to catch and the code to execute if the specified exception is thrown.

- finally

  used to define a block of code that we always want to execute, regardless of whether an exception was caught or not.

- throw

  typically used for throwing user defined exceptions.

- throws

  lists the types of exceptions a method can throw, so that the callers of the method can guard themselves against the exception.

Try, Catch and Finally are three components of exception handler

try

The first step in writing an exception handler is to enclose the statements that might throw an exception within a try block. The try block is said to govern the statements enclosed within it and defines the scope of any exception handlers (established by subsequent catch blocks) associated with it.

catch

Next, you associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

There can be no intervening code between the end of the try statement and the beginning of the first catch statement.

The catch block contains a series of legal Java statements. These statements are executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown.

finally

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block.

All Java methods use the throw statement to throw an exception. The throw statement requires a single argument: a throwable object. In the Java system, throwable objects are instances of any subclass of the Throwable

The throws clause specifies that the method can throw an EmptyStackException. As you know, the Java language requires that methods either catch or specify all checked exceptions that can be thrown within the scope of that method. You do this with the throws clause of the method declaration.

Try and catch

```
public class Demo      {
        public static void main(String [] args)          {
                int x,y;
        try      {
                x = 0;
                y = 10/x;
                System.out.println(“Now What ???”);
        }
        catch(ArithmeticException e) {
                System.out.println(“Division by zero”);
        }
        System.out.println(“Hi I am back !!!”);
        }
}
```

ArithmeticException

Thrown when an exceptional arithmetic condition has occurred. For example, an integer "divide by zero" throws an instance of this class. In the example x=0.

When 10 is divided by 0, this exception is thrown. It is caught in ArithmeticException and the statement in catch block is executed. Even if no try and catch block is provided then also JVM throws an exception for divide by 0. The Java language allows you to write general exception handlers that handle multiple types of exceptions. Each catch clause catches a particular type of exception

```
public class Demo{
```

```
public static void main(String [] args){
        try{
        int a = args.length;
        System.out.println("a = " + a);
        int b = 10/a;
        int c [] = { 1 };
        c [10] = 100;
        }
        catch(ArithmeticException e){
                System.out.println("Divide by zero");
        }
        catch(ArrayIndexOutOfBoundsException e){
                System.out.println("Index out of bounds");
        }
        }
}
```

Nested Try

```
public class Demo      {
public static void main(String [] args){
                try{
                    int a = args.length;
                    int b = 10/a;
                    try{
                        if ( a == 1)
                                a = a/(a-a);
                        if ( a == 2 )     {
catch(ArithmeticException e) {
System.out.println("Div by 0");
}//catch
}//main
                }//Demo
```

```
                    int c [] = { 1 };
                    c[10] = 100;
                }//if
            }//try
        catch( ArrayIndexOutOfBoundsException e){
            System.out.println("Out of Bounds");
        }//catch          }//try
```

try statements can be nested to form a layered approach to handling exceptions. When an exception occurs in a nested try block, a matching handler is sought following the nested try block. If a matching catch is not found, then a search occurs following the next layer try block. This continues until a match occurs, or until all catch statements have been viewed. If no match occurs, then the default handler is used. Note that the nesting of try blocks may not be clearly seen. For example, a method may be called from within a try block. Inside the called method, another try block exists, thus nesting the new block inside the existing one.

Throwable Class

```
public class Demo    {
    public static void main(String [] args)        {
        try  {  fun1();      }
        catch(Exception e)   {
            System.out.println("In Main");
        }
        fun2();
    }//main
public static fun1() throws Exception {
    try    {
```

```
            System.out.println("Try of fun1");
            throw new Exception();
        }
        catch(Exception e)     {
            System.out.println(Catch of fun 1");
        }
finally  {
System.out.println("Finally of fun 1");
}
}// fun 1
public static fun 2()    {
        System.out.println("Hello");
}// fun 2
}// Demo
```

When a throw statement executes, all subsequent statements in the try block are skipped and the catch statements are evaluated in order to find a suitable handler. The throws keyword is used to denote the fact that certain exceptions may not be handled by the current method. throws lists the types of exceptions that may be thrown inside a method for which no catch statement can be found inside the same class. If the throws keyword is missing, then a compiler error will result.

1.7 JAVA I/O

Java I/O is defined in terms of streams. Streams are ordered sequences of data that have: A source (input streams) or A destination (output stream). Streams are also classified in to two categories

1) Character streams
2) Byte streams

Input streams and Output streams are of the type Byte streams. Reader and Writer streams are of the type Character streams. All the classes required to handle input and output are given in java.io package.

Stream Types

Each type of stream results in four actual streams: Byte based input, Byte based output, Character based input (Reader), Character based output (Writer).

To read and write 8-bit bytes, programs should use the byte streams, descendants of InputStream and OutputStream. InputStream and OutputStream provide the API and partial implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes). These streams are typically used to read and write binary data such as images and sounds. Two of the byte stream classes, ObjectInputStream and ObjectOutputStream, are used for object serialization

Reader and Writer are the abstract superclasses for character streams in java.io. Reader provides the API and partial implementation for readers--streams that read 16-bit characters--and Writer provides the API and partial implementation for writers--streams that write 16-bit characters. Stream stores types in binary not ASCII or Unicode.

Piped Streams : Useful in communications between threads

ByteArray Streams : Read/write byte arrays

Filter Streams: Abstract classes, filter bytes from another stream. Can be chained together

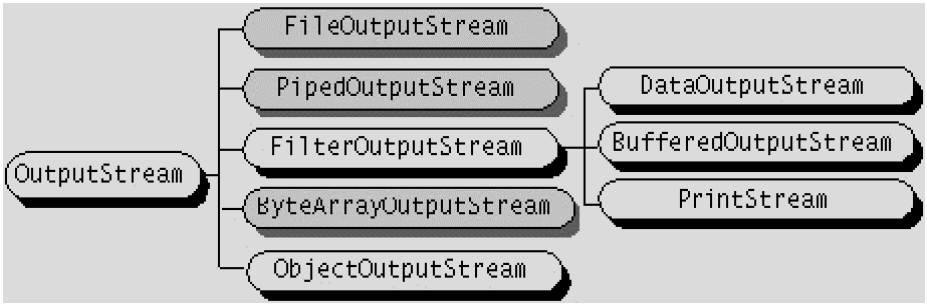
Buffered Streams : Adds buffering

File Streams : Used for file IO

Data Streams : Reads/writes built-in types (int, float, etc.)

Byte Streams

OutputStream



This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of OutputStream must always provide at least a method that writes one byte of output.

BufferedOutputStream: This class implements a buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written. The data is written into an internal buffer, and then written to the underlying stream if the buffer reaches its capacity, the buffer output stream is closed, or the buffer output stream is explicitly flushed.

FilteredOutputStream : This class is the superclass of all classes that filter output streams. These streams sit on top of an already existing output stream (the underlying output stream)

which it uses as its basic sink of data, but possibly transforming the data along the way or providing additional functionality.

ByteArrayOutputStream: This class implements an output stream in which the data is written into a byte array. The buffer automatically grows as data is written to it. The data can be retrieved using toByteArray() and toString().

ObjectOutputStream: An ObjectOutputStream writes primitive data types and graphs of Java objects to an OutputStream. The objects can be read (reconstituted) using an ObjectInputStream. Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be reconsituted on another host or in another process.

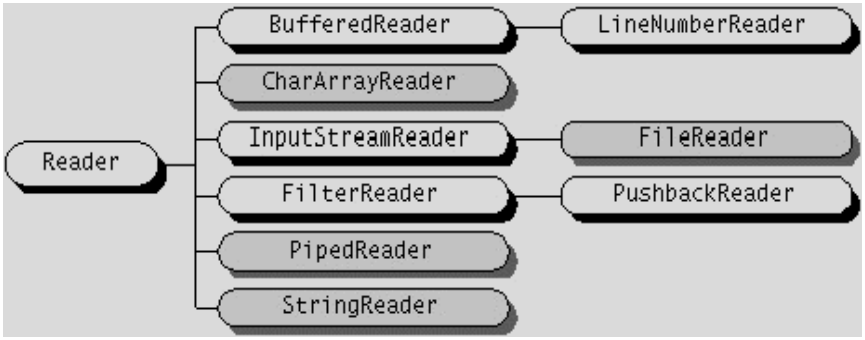
FileOutputStream: A file output stream is an output stream for writing data to a File or to a FileDescriptor. What files are available or may be created depends on the host environment.

PipedOutputStream: A piped output stream can be connected to a piped input stream to create a communications pipe. The piped output stream is the sending end of the pipe. Typically, data is written to a PipedOutputStream object by one thread and data is read from the connected PipedInputStream by some other thread. Attempting to use both objects from a single thread is not recommended as it may deadlock the thread.

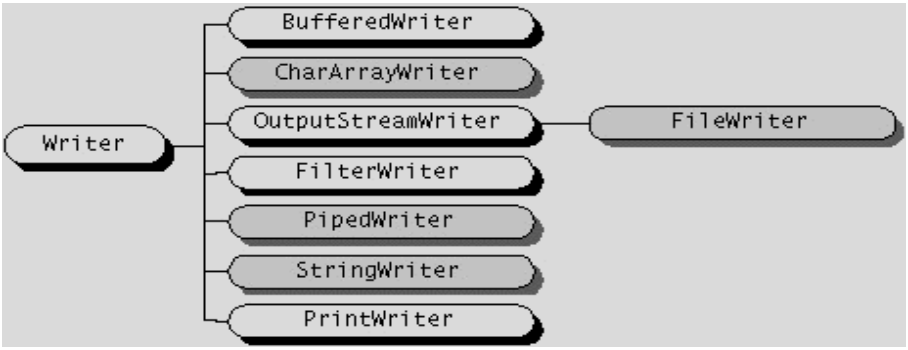
Character Streams

The primary advantage of character streams is that they make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easy to internationalize. Java stores strings in Unicode, an international standard character encoding that is capable of representing most of the world's written languages. Typical user-readable

text files, however, use encodings that are not necessarily related to Unicode, or even to ASCII, and there are many such encodings. Character streams hide the complexity of dealing with these encodings by providing two classes that serve as bridges between byte streams and character streams. A second advantage of character streams is that they are potentially much more efficient than byte streams



Reader: Abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.



Writer: Abstract class for writing to character streams. The only methods that a subclass must implement are write(char[], int, int), flush(), and close(). Most subclasses, however, will override some of the methods defined here in order to provide higher efficiency, additional functionality, or both.

BufferedWriter: Write text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.

CharArrayWriter: This class implements a character buffer that can be used as an Writer. The buffer automatically grows when data is written to the stream. The data can be retrieved using toCharArray() and toString().

OutputStreamWriter: Write characters to an output stream, translating characters into bytes according to a specified character encoding. Each OutputStreamWriter incorporates its own CharToByteConverter, and is thus a bridge from character streams to byte streams.

FilterWriter: Abstract class for writing filtered character streams.

PipedWriter : Piped character-output streams.

StringWriter: A character stream that collects its output in a string buffer, which can then be used to construct a string.

PrintWriter: Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams

FileWriter: Convenient class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable.

Streams

There two categories of streams namely: Data sink streams and Processing streams. Data sink streams read from or write to specialized data sinks such as strings, files, or pipes. Processing streams perform some sort of operation, such as buffering or character encoding, as they read and write.

Reading From an Input Stream

public abstract class **InputStream** extends Object

This abstract class is the superclass of all classes representing an input stream of bytes. Applications that need to define a subclass of **InputStream** must always provide a method that returns the next byte of input.

public abstract int read( ) throws IOException

Reads the next byte of data from the input stream. The value byte is returned as an int in the range 0 to 255. If the end of the stream has been reached, the value -1 is returned. This method blocks until input data is available.

Returns: the next byte of data, or -1 if the end of the stream is reached.

Throws: IOException - if an I/O error occurs.

public int read(byte[] b) throws IOException

Reads some number of bytes from the input stream and stores them into the buffer array b. The number of bytes actually read is returned as an integer. This method blocks until input data is available. If b is null, a NullPointerException is thrown. If the length of b is zero, then no bytes are read and 0 is returned. If no byte is available because the stream is at end of file, the value -1 is returned.

Parameters: b - the buffer into which the data is read.

the total number of bytes read into the buffer, or -1 is there is no more data because the end of the stream has been reached.

Throws: IOException - if an I/O error occurs.

public int read(byte[] b, int off, int len) throws IOException

Reads up to len bytes of data from the input stream into an array of bytes. The number of bytes actually read is returned as an integer.

Parameters: b - the buffer into which the data is read.

off - the start offset in array b at which the data is written.

len - the maximum number of bytes to read.

Returns: the total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

Throws: IOException - if an I/O error occurs.

long skip(long n) : Skips over and discards n bytes of data from this input stream.

int available() : Returns the number of bytes that can be read (or skipped over) from this input stream without blocking by the next caller of a method for this input stream.

void close() : Closes this input stream and releases any system resources associated with the stream.

public abstract class **Reader** extends Object

Abstract class for reading character streams. The only methods that a subclass must implement are read(char[], int, int) and close(). The Reader class provides a similar set of methods but they can read in unit of characters and array of characters.

int read()

int read(char cbuf[])

int read(char cbuf[], int offset, int length)

Bytes can be read one at a time or many at once :

```
InputStream is;
byte b;
while ( b = (byte) is.read() ) != -1 )        {
// Process
}
```

```
InputStream is;
int length;
byte [] b = new byte[512];
int bytesRead = is.read(b);
```

Reading a File

```
import java.io.*;
public class FileViewer     {
        public FileViewer(String name) throws IOException  {
                File f = new File(name);
                int size = (int) f.length();
                FileInputStream fis = new FileInputStream(f);
                byte [] data = new byte[size];
```

```
                while(bytes_read < size)    {
                        bytes_read += fis.read(data, bytes_read, size-bytes_read);
                        System.out.println(new String (data) );
                }
        }
        public static void main(String [] args) throws IOException      {
                if(args.length != 1)            {
                        System.out.println("USAGE:java FileViewer <filename>");
                        System.exit(0);
                }
                new FileViewer(args[0] );  }
}
```

Writing to an Output Stream

public abstract class **OutputStream** extends Object

This abstract class is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink. Applications that need to define a subclass of **OutputStream** must always provide **at least one** method that writes one byte of output.

public abstract void **write**(int b) throws IOException

public void write(byte[] b) throws IOException

public void **write**(byte[] b, int off, int len) throws IOException

public abstract class **Writer** extends Object

Abstract class for writing to character streams. The only methods that a subclass must implement are write(char[], int, int), flush(), and close(). Writer defines these methods for writing characters and arrays of characters

int write(int c)

int write(char cbuf[])

int write(char cbuf[], int offset, int length)

Using File Reader and File Writer

```
import java.io.*;
public class Copy {
        public static void main(String[] args) throws  IOException {
                File inputFile = new File(”Source.txt");
                File outputFile = new File(”Target.txt");
                FileReader in = new FileReader(inputFile);
                FileWriter out = new FileWriter(outputFile);
                int c;
                while ((c = in.read()) != -1)
                        out.write(c);
                in.close();
                out.close();
          }
}
```

Its the Convenient class for writing character files. The constructors of this class assume that the default character encoding and the default byte-buffer size are acceptable. This is the Abstract class for reading filtered character streams.

public int **read**() throws IOException

Reads a single character.

public int **read**(char[] cbuf, int off, int len) throws IOException Read characters into a portion of an array.

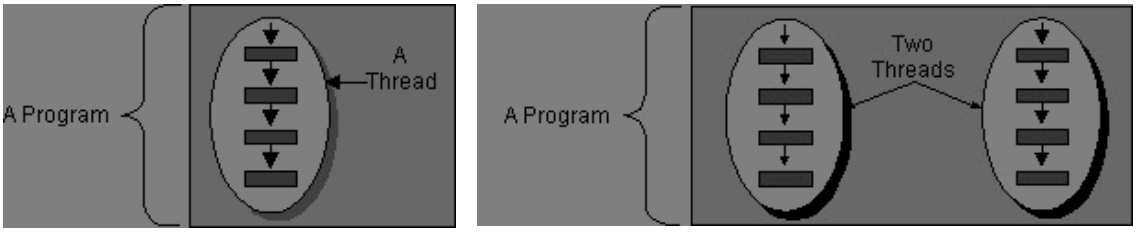
public void **write**(int c) throws IOException

Write a single character.

1.8 MULTITHREADING

Thread is a logical sequence of execution of code lines. The entire code can be divided in to different logical groups and each group can be called as a thread of execution. The Java thread is a lightweight process because it does not consume time and other

resources for launching it. A thread is similar to the sequential programs described previously. A single thread also has a beginning, a sequence, and an end and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; it cannot run on its own. Rather, it runs within a program. There is nothing new in the concept of a single thread. The real issue surrounding threads is not about a single sequential thread. Rather, it's about the use of multiple threads in a single program, running at the same time and performing different tasks. This is illustrated in the figure below.



In java there are two techniques for creating a thread:

- Implementing the Runnable Interface

  This gives the advantage of better object-oriented design. It uses only single inheritance.

- Subclassing Threading and Overriding run

  The option of extending from Thread class will lead to simple code. Since java technology allows only single inheritance, it is not possible to extend from any other class, if already we have extended from Thread. In some situations, this forces us to take the approach of implementing Runnable.

The Runnable Interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run. This interface is designed to provide a common protocol for objects that wish to execute code while they are active. For example, Runnable is implemented by class Thread. Being

active simply means that a thread has been started and has not yet been stopped. In addition, Runnable provides the means for a class to be active while not subclassing Thread. A class that implements Runnable can run without subclassing Thread by instantiating a Thread instance and passing itself in as the target. In most cases, the Runnable interface should be used if you are only planning to override the run() method and no other Thread methods. This is important because classes should not be subclassed unless the programmer intends on modifying or enhancing the fundamental behavior of the class.

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently. When a Java Virtual Machine starts up, there is usually a single non-daemon thread (which typically calls the method named main of some designated class). The Java Virtual Machine continues to execute threads until either of the following occurs:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.

- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagates beyond the run method.

The Thread Class

- Thread() Allocates a new Thread object.
- Thread(Runnable target) Allocates a new Thread object.
- Thread(Runnable target,String name) Allocates a new Thread object.
- Thread(String name) Allocates a new Thread object.

static Thread currentThread()	Returns a reference to the currently executing thread object.
String getName()	Returns this thread's name

int getpriority()	Returns this thread's priority.
ThreadGroup getThreadGroup()	Returns the thread group to which this thread belongs.
void destroy()	Destroys this thread, without any cleanup.
void start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
static void sleep(long millis)	Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds
void run()	If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Subclassing Thread

```
class SimpleThread extends Thread {
        public SimpleThread(String str) {
           super(str);
        }
        public void run() {
           for (int i = 0; i < 10; i++) {
              System.out.println(i + " " + getName());
              try {
                 sleep((long)(Math.random() * 1000));
              } catch (InterruptedException e) {}
           }
           System.out.println("DONE! " + getName());
        }
```

```
}
```

The first way to customize what a thread does when it is running is to subclass Thread (itself a Runnable object) and override its empty run method so that it does something.

The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the SimpleThread class is the run method. The run method is the heart of any Thread and where the action of the Thread takes place. The run method of the SimpleThread class contains a for loop that iterates ten times. In each iteration the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second. After the loop has finished, the run method prints DONE! along with the name of the thread. Following program will test this class.

```
public class ThreadTest {
 public static void main String[] args)          {
SimpleThread st1 = new SimpleThread(“Thread A”);
SimpleThread st2 = new SimpleThread(“Thread B”)
          st1.start();
            st2.start();
                  // OR
                        new SimpleThread(“Thread A”).start();
                        new SimpleThread(“Thread B”).start(); }        }
```

Implementing the Runnable Interface

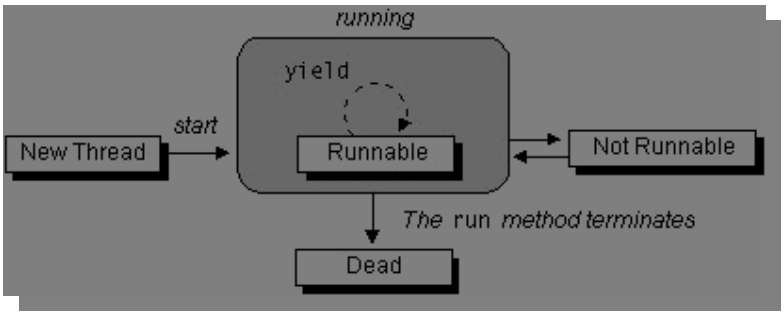
```
class SimpleThread implements Runnable {
        public void run() {
                Thread t = Thread.currentThread();
                for (int i = 0; i < 10; i++) {
                        System.out.println(i + " " + t.getName() );
```

```
                    try {
                        Thread.sleep((long)(Math.random() * 1000));
                    } catch (InterruptedException e) {}
                }
                System.out.println("DONE! " + t.getName());
        }
}
```

If your class must subclass some other class,you should use Runnable. Following class tests the above example.

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

Runnable r1 = new SimpleThread();
Runnable r2 = new SimpleThread();
Thread t1 = new Thread(r1,"Thread A");
Thread t2 = new Thread(r2,"Thread B");

t1.start();
t2.start();
}
}
```

The Life cycle of thread



This diagram shows the states that a Java thread can be in during its life. It also illustrates which method calls cause a transition to another state. A thread life cycle has four distinct phases like creating a thread, starting a thread, making a thread not Runnable and stopping a thread. Java provides constructs to control the threads in these phases. At thread becomes not runnable either when the sleep method is invoked or the thread calls the wait method to wait for a specific condition to be satisfied or the thread is in a blocking i/o.

The API for the Thread class includes a method called isAlive. The isAlive method returns true if the thread has been started and not stopped. If the isAlive method returns false, you know that the thread either is a New Thread or is Dead. If the isAlive method returns true, you know that the thread is either Runnable or Not Runnable. You cannot differentiate between a New Thread or a Dead thread. Nor can you differentiate between a Runnable thread and a Not Runnable thread.

Thread Priority

Usually we say that threads run concurrently. While conceptually this is true, in practice it usually isn't. Most computer configurations have a single CPU, so threads actually run one at a time in such a way as to provide an illusion of concurrency. Execution of multiple threads on a single CPU, in some order, is called scheduling. The Java runtime supports a very simple, deterministic scheduling algorithm known as fixed priority scheduling. This algorithm schedules threads based on their priority relative to other runnable threads.

When a Java thread is created, it inherits its priority from the thread that created it. You can also modify a thread's priority at any time after its creation using the setPriority method. Thread priorities are integers ranging between MIN_PRIORITY and MAX_PRIORITY (constants defined in the Thread class). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some reason will a lower priority thread

start executing. If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion.

The chosen thread will run until one of the following conditions is true:

- A higher priority thread becomes runnable.
- It yields, or its run method exits.
- On systems that support time-slicing, its time allotment has expired

Then the second thread is given a chance to run, and so on, until the interpreter exits. The Java runtime system's thread scheduling algorithm is also preemptive. If at any time a thread with a higher priority than all other runnable threads becomes runnable, the runtime system chooses the new higher priority thread for execution. The new higher priority thread is said to preempt the other threads.

Controlling Threads

It's an art of moving threads from one state to another state. It is achieved by triggering state transitions. The various pathways out of the running state can be:

- Yielding
- Suspending and then resuming
- Sleeping and then waking up
- Blocking and then continuing
- Waiting and then being notified

Yielding

A call to yield method causes the currently executing thread to move to the ready state, if the scheduler is willing to run any other thread in place of the yielding thread. A thread that has yielded goes into ready state. The yield method is a static method of the Thread class.

Suspending

It may sometimes be useful to temporarily stop a thread from processing in your application. When you suspend a thread, the state of the thread, including all the attributes and locks held by the thread, is maintained until that thread is resumed. Use thread suspension carefully. Suspending threads can easily cause application deadlocks and timeout conditions. You can solve most problems that involve thread suspension by using other safer mechanisms (such as synchronization). Thread.suspend is inherently deadlock-prone so it is also being deprecated, thereby necessitating the deprecation of Thread.resume. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling resume, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.

Sleeping

A sleeping thread passes time without doing anything and without using the CPU.
A call to the sleep method requests the currently executing thread to cease it’s execution for a specified period of time.

•public static void sleep(long ms) throws InterruptedException

•public static void sleep(long ms, int ns) throws InterruptedException

Both sleep and yield work on currently executing thread.

Blocking

Methods that perform input or output have to wait for some occurrence in the outside world before they can proceed, this behavior is known as blocking.

```
Try     {
        Socket s = new Socket(“Daemon”,6786);
        InputStream is = s.getInputStream();
        int b = is.read();
}
catch(IOException e)  {
        // Handle the exception
}
```

A thread can also become blocked if it fails to acquire the lock for a monitor if it issues a wait call. The wait method puts an executing thread into the waiting state and the notify and notifyAll put them back to ready state. When using threads, thread synchronization is one of important issues.

1.9 SUMMARY

Java's strengths lie in its portability - both at the source and at the binary level, in its object-oriented design and in its simplicity. A Java program is made up primarily of classes and objects. Classes and objects, in turn, are made up of methods and variables, and methods are made up of statements and expressions. Instance and class variables, hold the attributes of the class and its instances. Instance and class methods, define a class's behavior.

1.10 Self Test

1. Which of the following is a valid JAVA identifier?

   A. int
   B. anInt
   C. 1thing
   D. ONE-HUNDRED
   E. Cop2253.01

2. Which is not a JAVA reserved word

   A. final
   B. null
   C. break
   D. String
   E. true

3. Which is machine/platform independent?

   A. JAVA interpreter
   B. JAVA bytecode
   C. JAVA compiler
   D. Machine code
   E. Bytecode compiler

4. **for (int n=1; n<100;n++) stmt;**

   How many times the statement will be executed?

   A. 100
   B. 0

C. 99

D. 98

E. 1

5. **while (true) {**

stmt;

break;

}

stmt will be executed

A. once

B. never

C. infinite loop

D. 2 times

6. Consider the following class definition:

```
1. public class Test extends Base {
2.   public Test(int j) {
3.   }
4.   public Test(int j, int k) {
5.     super(j, k);
6.   }
7. }
```

Which of the following are legitimate calls to construct instances of the Test class?

A. `Test t = new Test();`

B. `Test t = new Test(1);`

C. `Test t = new Test(1, 2);`

D. `Test t = new Test(1, 2, 3);`

E. `Test t = (new Base()).new Test(1);`

7. Consider this class:

```
1. public class Test1 {
2.   public float aMethod(float a, float b) {
3.   }
4.
5. }
```

Which of the following methods would be legal if added (individually) at line 4?

A. public int aMethod(int a, int b) { }
B. public float aMethod(float a, float b) { }
C. public float aMethod(float a, float b, int c) throws _Exception { }
D. public float aMethod(float c, float d) { }
E. private float aMethod(int a, int b, int c) { }

8. The Java thread is a lightweight process

A. True
B. False

9. Character streams are easy to internationalize

A. True
B. False

10. A Throwable class contains a snapshot of the execution stack of its thread at the time it was created.

A. True
B. False

Answers

1. B
2. D
3. B
4. C
5. A
6. B, C
7. A, C, E
8. A
9. A
10. A

1.11 Questions

1. Why is java called platform independent? Explain
2. Explain the concepts of inheritance, method overriding and polymorphism in java
3. How is multiple inheritance achieved in java.
4. What are exceptions? Explain how they are handled in java.
5. How is garbage collection handled in java?
6. What is a thread? Explain the thread concept of java.

UNIT 2

JAVA GUI PROGRAMMING

2.1 INTRODUCTION

Applets are compiled java class files that are displayed in side a web browser. Applets add on to the functionality provided by the HTML. Java applets can be used to build full-featured graphical user interfaces, communicate over the Internet to a host server, and even communicate with other applets on a form.

The Java programming language provides a class library called the Abstract Window Toolkit (AWT) that contains a number of common graphical widgets. You can add these widgets to your display area and position them with a layout manager. AWT had some inherent shortcomings like dependency on their native peers for displaying userinterface components. JFC is a set of classes for bulding graphical user interfaces in java. JFC is composed of five APIs: AWT, Java 2D, Accessibility, Drag and Drop, and Swing.

Swing includes a component set that is targeted at forms-based applications. Loosely based on Netscape's acclaimed Internet Foundation Classes (IFC), the Swing components have had the most immediate impact on Java development. They provide a set of well-groomed widgets and a framework to specify how GUIs are visually presented, independent of platform.

2.2 OBJECTIVES

In this unit you will learn

- What is an Applet?
- Security aspects of Applet
- How to write and run an applet
- What is AWT?
- AWT basics

- Event handling
- What is JFC and Swing?
- Basics of Swing
- Layout Management

2.3 JAVA APPLETS

What is an Applet?

Java applets are compiled java class files that are run on a page within a Web browser. Thus applets are the client side components. Because the Web browser supplies the framework for retrieving and loading an applet and also supplies the main window holding the applet, java applets are somewhat simpler than a true java application. A Java application does not require any additional software, such as a Web browser, other than the typical java runtime files included in the Java Developer's Kit. Java applications are designed to be run in stand-alone mode on a client machine and therefore can be thought of as a known entity. Java applets, meanwhile, can be loaded from anywhere on the Internet, and therefore are subject to severe security restrictions.

Java applets are typically run within a Web browser like Netscape Navigator or Microsoft Internet Explorer. Of course, the AppletViewer tool included with the Java Developer's Kit can be used to test and run java applets as well. However, it will not be included on all client machines; therefore, it is not really an option to applet developers. Many beginning Java developers wonder how an applet can run within a browser if the Java runtime libraries are not installed on the client system. Actually web browsers have inbuilt java environment in them to run applets. They basically install a customized version of the JDK in their runtime directories so that they can load java applets and their own set of customized java classes.

Capabilities and Limitations of Applets

Java applets can be used to build full-featured graphical user interfaces, communicate over the Internet to a host server, and even communicate with other applets on a form. All of this can be done in an operating-environment-neutral manner, which is what makes this such a great technology.

For Java to be truly successful, however, the client security has to be completely assured. Because of this, security measures place some limitations on java applets. By default, applets cannot communicate with any server other than the originating server. Applets also cannot read or write files to the local file system. The basic security model for java treats all java applets and applications as unknown, unsecured objects running within a secure environment.

The Java designers have handled security at three levels:

- The elimination of pointers from the language eliminates an entire class of security problems. Programmers in C, for instance, can fake objects in memory because it is loosely typed and allows pointers.
- The bytecode verification process forces uploaded Java applets to undergo a rigorous set of checks in order to run on the local system. In other words, this will foil "bad" users who decided to write a hostile compiler.

- Client-side precautions add another layer of security. Most Web browsers preclude Java applets from doing file access or communicating with any computer on the Internet other than the computer that the applet was uploaded from. The Java class loader assists in this process.

Bytecode Verification

Language security features are simply not enough to prevent an applet from reformatting your hard drive or some other unspeakable act. Features needed to be built into the entire runtime system to prevent specially compiled applets from invading remote systems. Java is an interpreted language. This means that actual memory management for the application is put off until runtime (it is not built into the compiled Java classes). This feature allows Java to run on many different platforms thanks to the installed Java Virtual Machine. However, it also allows the Java runtime engine to verify that the bytecodes being loaded are, in fact, good bytecodes. This is done using a part of the Virtual Machine known as the verifier. The verifier has the unenviable task of checking each bytecode before it is executed (interpreted) to make sure that it is not going to perform an illegal operation. After the bytecode has been verified, the applet is guaranteed to do the following:

- Obey access restrictions such as public, protected, private, and friendly. No class will be allowed to access data that goes against these restrictions.
- Never perform illegal data conversions. Because Java is a strongly typed language, automatic conversions from arrays to pointers, for instance, are not allowed.
- Conform to all return, parameter, and argument types when calling methods.
- Live within its allocated stack. An applet that overruns its memory will not be loaded.

Client-Side Precautions

The set of precautions enforced by the client Web browser (or other applet loader) is done by a part of the Java runtime engine known as the class loader. The class loader does what its name says: it loads classes.

Three possible worlds are recognized by the class loader:

- The local system (highest level)
-The local network within a firewall (middle level)
-The Internet at large (lowest level)

The class loader implements defined rules that allow it to intelligently prevent an applet from wreaking havoc on your system. It does this by never allowing a class loaded from a lower level to replace a class existing on a higher level. The following example illustrates what this means.

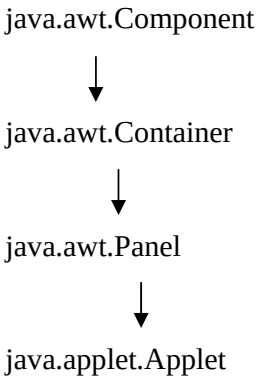
An applet located on a Web server across the Internet imports the java.awt.Button class so that it can display a button on the screen. The developer on the remote machine changed some of the button's internal functionality but kept the class interface without changing anything. Fortunately for you, the java.awt.Button class is included with the Java Virtual Machine installed on your system. Therefore, when the applet is uploaded to your machine, the class loader will always retrieve your local Button.class file. In addition to this, classes cannot call methods from other classes in other security levels unless those methods are explicitly declared to be public. This means that Java applets loaded from a remote machine cannot call file system I/O methods. If those methods were called, the class loader would catch the error, and the applet load would fail.

Java handles security at several different levels. The language is designed in a manner that removes many security holes because it does not allow pointer manipulation. The bytecode verifier is used to verify each uploaded Java class to ensure that it obeys all Java language rules. The class loader enforces security on another level by controlling applet operations at runtime. It is important to realize that the bytecode verifier and class loader both exist on the local system and are part of the Java Virtual Machine. Because these two components are critical to the success of the Java security model, the user must rely on these components to ensure that Java applets are secure.

Building a Java Applet

Java applets are subclassed from the Applet class in the java.applet package.

java.lang.Object



Each applet has four major events in its lifetime:

- Initialization
- Starting
- Stopping
- Destroying

These four events correspond directly to four methods within the Applet class: init(), start(), stop(), and destroy().

public void init()

The init() method is called when the applet is initially loaded. This method is used to do one-time setup features such as add components to the layout manager, set screen colors, and connect to a host database.

public void start()

The start() method is called after the applet has been initialized, and also each time the applet is restarted after being stopped. Applets can be stopped when the user changes to another Web page. If the user returns at a later time to the page with the applet on it, the applet will be restarted by the browser. Therefore, the start() method can be called many

times during an applet's life cycle. Common operations that occur during an applet's start() method are the initialization of threads within the applet and the updating of the screen display.

public void stop()

The stop() method is called whenever the user leaves the current page. Note that by default when the user leaves a page, the applet will continue to run. This can result in an enormous consumption of system resources if many applets have been loaded and these applets are performing some resource-intensive task such as animation. The stop() method is used to temporarily suspend the execution of the applet until the start() method is called again.

public void destroy()

The destroy() method is called whenever it is time to completely finish the applet's execution. This method is generally called when the browser is exiting or the applet is being reloaded from the host server. The destroy() method is used to free up allocated resources such as threads or database connections.

Program below shows a simple applet that implements all four life cycle methods: init(), start(), stop(), and destroy(). This applet updates the screen as well as the browser status bar with some information indicating which method is being called.

```
import java.awt.Graphics;
import java.awt.Font;
import java.awt.Color;
public class LifeCycleApplet extends java.applet.Applet
{
  Font theFont = new Font("Helvetica", Font.BOLD, 20);
```

```
int  i;
String String1, String2;

public void paint(Graphics g)
{
  g.setFont(theFont);
  g.setColor(Color.blue);
  g.drawString(String1 + String2, 5, 30);
}
public void init()
{
  i = 0;
  String1 = "";
  String2 = "The applet is initializing!";
  repaint();
  showStatus("The applet is initializing!");
}

public void start()
{
  i = 1;
  String1 = String2;
  String2 = "The applet is starting!";
  repaint();
  showStatus("The applet is starting!");
}

public void stop()
{
  i = 2;
  String1 = String2;
  String2 = "The applet is stopping!";
  repaint();
```

```
    showStatus("The applet is stopping!");
  }

  public void destroy()
  {
    i = 3;
    String1 = String2;
    String2 = "The applet is being destroyed!";
    repaint();
    showStatus("The applet is being destroyed!");
  }
}
```

The paint method is one of two display methods an applet can override:

Paint : The basic display method. Many applets implement the paint method to draw the applet's representation within a browser page.

update : A method you can use along with paint to improve drawing performance.

Applets inherit their paint and update methods from the Applet class, which inherits them from the Abstract Window Toolkit (AWT) Component class. The applet's coordinate system starts at (0,0), which is at the upper left corner of the applet's display area.

The repaint method causes a call to this component's update method as soon as possible.

The update method of Component does the following:

Clears this component by filling it with the background color.

Sets the color of the graphics context to be the foreground color of this component.

Calls this component's paint method to completely redraw this component.

The previous example will always show the last two life cycle events on the screen. Because of the speed at which Java executes, you will probably not see the init() method's results all by themselves.

HTML and Java Applets

The last example took advantage of a file named Example.html to display the applet within a Web browser. Java applets can be displayed within a Web browser by being embedded in a standard HTML page. This does not mean that the actual bytecode or source code of the applet is included within the HTML file. Instead, the HTML text contains a reference to the Java applet known as a tag. Each element within an HTML file to be displayed by a Web browser is referenced using different types of these tags. Program below shows the contents of the HTML file used to load the LifeCycleApplet applet

```
   <HTML>
<HEAD>
<TITLE>This is the LifeCycle applet!</TITLE>
</HEAD>
<BODY>
<H1>Prepare to be amazed!</H1>
<BR>
<APPLET CODE="LifeCycleApplet.class" WIDTH=600 HEIGHT=50>
If you can see this, your browser does not support Java applets.
</APPLET>
</BODY>
</HTML>
```

A quick examination of HTML program shows three primary elements:

The <TITLE> tag-Used to display the title caption for this page.

The <H1> tag-Used to represent the main heading for the page.

The <APPLET> tag-Used to represent a Java applet to be loaded.

If you are new to HTML, the most important point to realize is that nothing in this file specifies actual screen layout. The TITLE tag, for instance, does not specify that the title appear at (x,y) coordinates (150, 200) and that it should be set apart from the heading by ¨". HTML only specifies the markup that tells browsers what information to display. The actual screen layout within the applet is completely controllable down to the pixel level, by a java developer.

The <APPLET> Tag

The syntax for using the <APPLET> tag is the following:

<APPLET attributes>
applet_parameters
alternate_content
</APPLET>

The APPLET attributes are standard values that all applets accept and are a standard part of HTML. The applet_parameters are applet-specific parameters that are read by the applet at runtime. This is a handy way of passing arguments to an applet to allow the applet to be more generic.

<APPLET> Tag Attributes

ALT-Alternate text that can be displayed by text-only browsers.

ALIGN-The ALIGN attribute designates the alignment of the applet within the browser page.

CODE-(Required) The CODE attribute is used to indicate the .class file that loads the applet.

CODEBASE-The CODEBASE attribute is used to indicate the location of the .class file that loads the applet.

HEIGHT-(Required) The HEIGHT attribute is used to set the applet's bounding rectangle height.

HSPACE-The HSPACE attribute sets the amount of horizontal space to set off around the applet.

NAME-The NAME attribute sets the symbolic name of the applet.

VSPACE-The VSPACE attribute sets the amount of vertical space to set off around the applet.

WIDTH-(Required) The WIDTH attribute is used to set the applet's box width.

Passing Parameters to Java Applets

Parameters are an easy way to configure Java applets without actually changing the source file. In the previous applet example, the text drawn on the screen was drawn using the blue color. This was "hardwired" into the applet's code. However, just as easily, we could have passed a parameter to the applet specifying that it use the blue tag. Program below shows how to pass parameters to the applet using HTML parameters

```
.       <HTML>
  <HEAD>
  <TITLE>This is the LifeCycle applet!</TITLE>
  </HEAD>
  <BODY>
  <H1>Prepare to be amazed!</H1>
  <BR>
  <APPLET CODE="LifeCycleApplet.class" WIDTH=600 HEIGHT=50>
  <PARAM NAME=color VALUE="blue">
  If you can see this, your browser does not support Java applets.
  </APPLET>
  </BODY>
  </HTML>
```

How does the Java applet determine the value of the parameters? The answer is that the applet has to call the getParameter() method supplied by the java.applet.Applet parent class. Calling getParameter("color") using the previous Java applet example would return a String value containing the text "blue". It is then left up to the applet to take advantage of this information and actually paint the text blue on the screen.

Here are three methods commonly used by applets:

String getParameter(String name)-Returns the value for the specified parameter string

URL getCodeBase()-Returns the URL of the applet

URL getDocumentBase()-Returns the URL of the document containing the applet

Inter-Applet Communication

Imagine a case where a button-click in one applet could update a database and trigger update messages to another applet running within the same Web page or even running on a remote computer. This type of communication is known as inter-applet communication.

Possibilities of Inter-Applet Communication

There are several interesting possibilities that will be available when applets are truly enabled to communicate among themselves. Here are possible types of communication:

- Applets that share the same Web page within the same browser
- Applets loaded in completely different browser windows
- Applets loaded on different client's browsers
- Applets loaded from different servers

Using a technology such as Java Beans or ActiveX will also allow developers to "wrap" their applets with code that will expose an applet's methods and properties in some standardized way. Using the component model, the applet could potentially be dropped onto a Visual Basic form or used to communicate with applications written in other programming languages.

Some Inter-Applet Communication Methods

Once again, keep in mind that currently, the only way to implement this type of communication is to take advantage of specific browser features. The following examples focus on capabilities using the Netscape Navigator browser.

Using the AppletContext

Calling the getApplets() or getApplet() method is the easiest way to reach applets running within the same Web page. This method returns an Enumeration containing all applets located within the current AppletContext. This works fine on browsers that treat a single Web page as an AppletContext; however, some browsers break each applet up into a separate AppletContext.

Using JavaScript

The Netscape Navigator 3.0 browser implements a technology known as LiveConnect to allow an applet to expose its methods and data to the outside world. Using the scripting language known as JavaScript, the Web page developer can access and change an applet's internal data.

Netscape Navigator 3.0 will also allow the use of cookies. Cookies are basically text files that can contain messages. At the current time, only JavaScript can actually access cookies, but this feature could be useful to script together applets that were loaded at different times. These features are truly browser-dependent. HTML files containing these features can only be displayed correctly using the Netscape Navigator 3.0 browser.

Using Static Variables

If both applets doing the communication share the same class, a common inter-applet communication mechanism is the use of static variables. Static variables can be thought of as global variables that apply to all instances of the class within an applet.

2.4 AWT FUNDAMENTALS

Graphical User Interfaces

The Java programming language provides a class library called the Abstract Window Toolkit (AWT) that contains a number of common graphical widgets. You can add these widgets to your display area and position them with a layout manager.

AWT Basics

All graphical user interface objects stem from a common superclass, Component. To create a Graphical User Interface (GUI), you add components to a Container object. Because a Container is also a Component, containers may be nested arbitrarily. Most often, you will use a Panel when creating nested GUIs.

Each AWT component uses native code to display itself on your screen. When you run a Java application under Microsoft Windows, buttons are really Microsoft Windows buttons. When you run the same application on a Macintosh, buttons are really Macintosh buttons. When you run on a UNIX machine that uses Motif, buttons are really Motif buttons.

Applications versus Applets

Recall that an Applet is a Java program that runs in a web page, while an application is one that runs from the command line. An Applet is a Panel that is automatically inserted into

a web page. The browser displaying the web page instantiates and adds the Applet to the proper part of the web page. The browser tells the Applet when to create its GUI (by calling the init() method of Applet) and when to start() and stop() any special processing.

Applications run from a command prompt. When you execute an application from the command prompt, the interpreter starts by calling the application's main() method.

Basic GUI Logic

There are three steps you take to create any GUI application or applet:

1. Compose your GUI by adding components to Container objects.
2. Setup event handlers to respond to user interaction with the GUI.
3. Display the GUI (automatically done for applets, you must explicitly do this for applications).

When you display an AWT GUI, the interpreter starts a new thread to watch for user interaction with the GUI. This new thread sits and waits until a user presses a key, clicks or moves the mouse, or any other system-level event that affects the GUI. When it receives such an event, it calls one of the event handlers you set up for the GUI. Note that the event handler code is executed **within** the thread that watches the GUI!

Because this extra thread exists, your main method can simply end after it displays the GUI. This makes GUI code very simple to write in AWT. Compose the GUI, setup event handlers, then display.

A Simple Example

The following simple example shows some GUI code. This example creates an Applet that contains just an Applet.

```
import java.awt.Button;
```

```
import java.applet.Applet;

public class AButton extends Applet {
  public void init() {
    // STEP 1: Compose the GUI
    Button beepButton = new Button("Beep");
    add(beepButton);
    // STEP 2: Setup Event handlers
    beepButton.addActionListener(new Beeper());

    // STEP 3: Display the GUI (automatic -- this is an applet)
  }
}
```

In step 2 from above, event handling is set up by adding an instance of a listener class to the button. When the button is pressed, a certain method in the listener class is called. In this example, the listener class implements ActionListener (because Button requires it). When the button is pressed, the button calls the actionPerformed() method of the listener class.

Suppose you want to produce a "beep" sound when the button is pressed. You can define your event handler as follows:

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
import java.awt.Component;
 public class Beeper implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    Component c = (Component)event.getSource();
    c.getToolkit().beep();
  }
}
```

When actionPerformed() is called, it produces a beep sound on the computer.

To try this applet, create a simple HTML page as follows.

```
<html>
  <applet code=AButton.class width=100 height=100>
```

```
 </applet>
</html>
```

Then test the HTML page by running appletviewer or by loading the HTML file in a browser that supports the Java Runtime Environment (JRE). Note that in this case, the browser must support at least version 1.1 of the JRE, as the example uses the event handling capabilities introduced with that release.

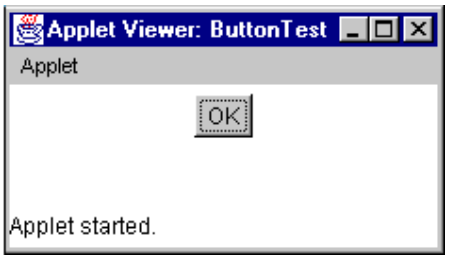
AWT Components

All AWT components extend class Component. Having this single class is rather useful, as the library designers can put a lot of common code into it. Lets examine each of the AWT components. Most, but not all, directly extend Component.

Buttons

A Button has a single line label and may be "pushed" with a mouse click.

```
import java.awt.*;
import java.applet.Applet;

public class ButtonTest extends Applet {
  public void init() {
    Button button = new Button("OK");
    add(button);
  }
}
```

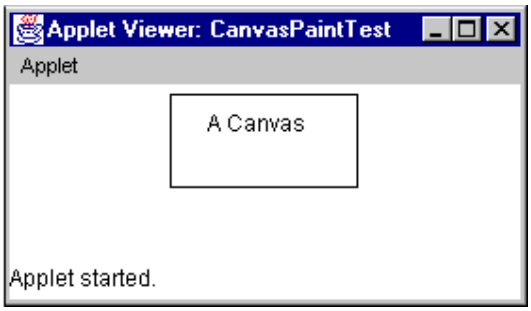


Note that in the above example there is no event handling added; pressing the button will not do anything. The AWT button has no direct support for images as labels.

Canvas

A Canvas is a graphical component representing a region where you can draw things such as rectangles, circles, and text strings. The name comes from a painter's canvas. You

subclass Canvas to override its default paint() method to define your own components. You can subclass Canvas to provide a custom graphic in an applet.




```
import java.awt.Canvas;
import java.awt.Graphics;

class DrawingRegion extends Canvas {
  public DrawingRegion() {
    setSize(100, 50);
  }
  public void paint(Graphics g) {
    g.drawRect(0, 0, 99, 49); // draw border
    g.drawString("A Canvas", 20,20);
  }
}
```

Then you use it like any other component, adding it to a parent container, for example in an Applet subclass.

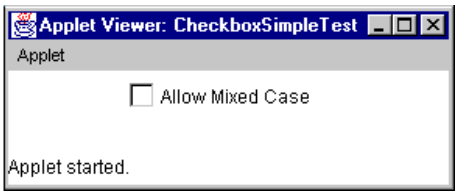
```
import java.applet.Applet;
public class CanvasPaintTest extends Applet {
  public void init() {
    DrawingRegion region = new DrawingRegion();
    add(region);
  }
```

The Canvas class is frequently extended to create new component types, for example image buttons. However, starting with the JRE 1.1, you can now directly subclass Component directly to create lightweight, transparent widgets.

Checkbox

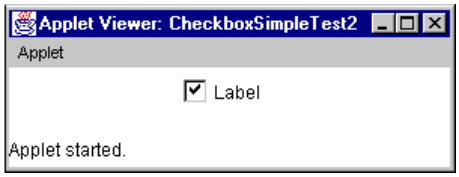
A Checkbox is a label with a small pushbutton. The state of a Checkbox is either true (button is checked) or false (button not checked). The default initial state is false. Clicking a Checkbox toggles its state. For example:

```
import java.awt.*;
import java.applet.Applet;

public class CheckboxSimpleTest extends Applet {
  public void init() {
    Checkbox m = new Checkbox("Allow Mixed Case");
    add(m);
  }
}
```



To set a Checkbox initially true use an alternate constructor:

```
import java.awt.*;
import java.applet.Applet;

public class CheckboxSimpleTest2 extends Applet {
  public void init() {
    Checkbox m = new Checkbox("Label", true);
    add(m);
  }
}
```

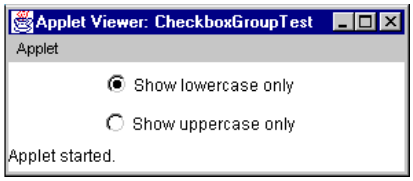


CheckboxGroup

A CheckboxGroup is used to control the behavior of a group of Checkbox objects (each of which has a true or false state). Exactly one of the Checkbox objects is allowed to be true at one time. Checkbox objects controlled with a CheckboxGroup are usually referred to as "radio buttons".

The following example illustrates the basic idea behind radio buttons.

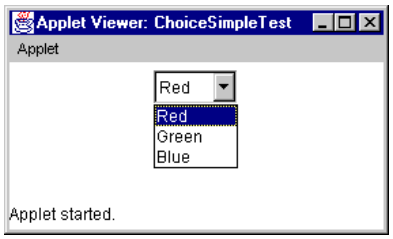
```
import java.awt.*;
import java.applet.Applet;


public class CheckboxGroupTest extends Applet {
  public void init() {
    // create button controller
    CheckboxGroup cbg = new CheckboxGroup();
     Checkbox cb1 =
      new Checkbox("Show lowercase only", cbg, true);
    Checkbox cb2 =
      new Checkbox("Show uppercase only", cbg, false);


    add(cb1);
    add(cb2);
  }
}
```



Choice

Choice objects are drop-down lists. The visible label of the Choice object is the currently selected entry of the Choice.

```
import java.awt.*;
import java.applet.Applet;
 public class ChoiceSimpleTest extends Applet {
  public void init() {
    Choice rgb = new Choice();
   rgb.add("Red");
    rgb.add("Green");
    rgb.add("Blue");


    add(rgb);
  }
}
```

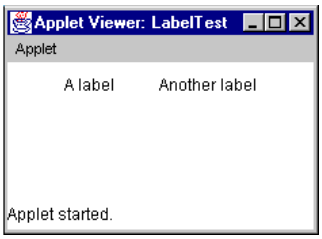


The first item added is the initial selection.

Label

A Label is a displayed Label object. It is usually used to help indicate what other parts of the GUI do, such as the purpose of a neighboring text field.

```
import java.awt.*;
import java.applet.Applet;


public class LabelTest extends Applet {
  public void init() {
    add(new Label("A label"));
    // right justify next label
    add(new Label("Another label", Label.RIGHT));
  }
}
```

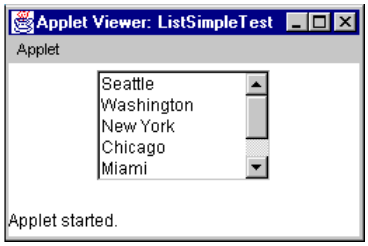


Like the Button component, a Label is restricted to a single line of text.

List

A List is a scrolling list box that allows you to select one or more items. Multiple selections may be used by passing true as the second argument to the constructor.

```
 import java.awt.*;
import java.applet.Applet;


public class ListSimpleTest extends Applet {
  public void init() {
    List list = new List(5, false);
    list.add("Seattle");
    list.add("Washington");
    list.add("New York");
    list.add("Chicago");
    list.add("Miami");
```



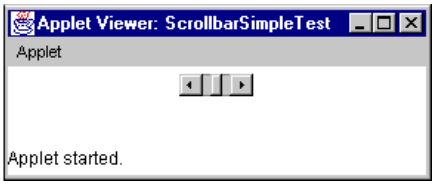
```
    list.add("San Jose");
    list.add("Denver");
     add(list);
  }
}
```

The constructor may contain a preferred number of lines to display. The current LayoutManager may choose to respect or ignore this request.

Scrollbar

A Scrollbar is a "slider" widget with characteristics specified by integer values that are set during Scrollbar construction. Both horizontal and vertical sliders are available.

```
import java.awt.*;
import java.applet.Applet;

// A simple example that makes a Scrollbar appear
public class ScrollbarSimpleTest extends Applet {
  public void init() {
    Scrollbar sb =
      new Scrollbar(Scrollbar.HORIZONTAL,
                    0, // initial value is 0
                    5, // width of slider
                  -100, 105); // range -100 to 100
    add(sb);
  }
}
```



The maximum value of the Scrollbar is determined by subtracting the Scrollbar width from the maximum setting (last parameter).

TextField

A TextField is a scrollable text display object with one row of characters. The preferred width of the field may be specified during construction and an initial string may be specified.


```
import java.awt.*;
import java.applet.Applet;


public class TextFieldSimpleTest extends Applet {
  public void init() {
    TextField f1 =
      new TextField("type something");
    add(f1);
  }
}
```

Tips:

Call setEditable(false) to make the field read-only.

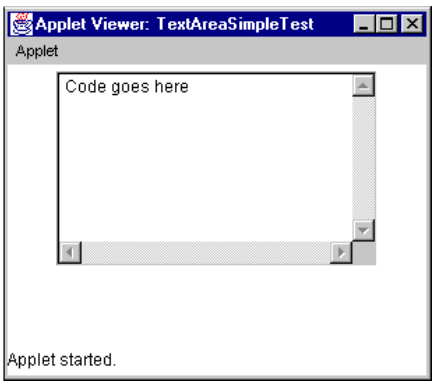
For password fields: field.setEchoChar('?');

To clear/reset: field.setEchoChar((char)0);

TextArea

A TextArea is a multi-row text field that displays a single string of characters, where newline ('\n' or '\n\r' or '\r', depending on platform) ends each row. The width and height of the field is set at construction, but the text can be scrolled up/down and left/right.

```
import java.awt.*;
import java.applet.Applet;
```

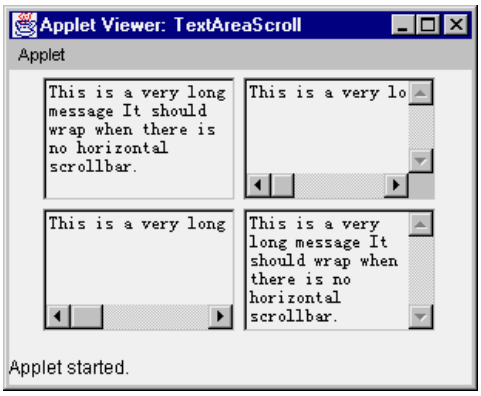


```
public class TextAreaSimpleTest extends Applet {
  TextArea disp;
  public void init() {
    disp = new TextArea("Code goes here", 10, 30);
    add(disp);
  }
}
```

There is no way, for example, to put the cursor at beginning of row five, only to put the cursor at single dimension position 50.

There is a four-argument constructor that accepts a fourth parameter of a scrollbar policy. The different settings are the class constants: SCROLLBARS_BOTH, SCROLLBARS_HORIZONTAL_ONLY, SCROLLBARS_NONE, and SCROLLBARS_VERTICAL_ONLY. When the horizontal (bottom) scrollbar is not present, the text will wrap.

```
import java.awt.*;
import java.applet.Applet;

public class TextAreaScroll extends Applet {
  String s =
    "This is a very long message " +
    "It should wrap when there is " +
    "no horizontal scrollbar.";
  public void init() {
    add(new TextArea (s, 4, 15,
      TextArea.SCROLLBARS_NONE));
    add(new TextArea (s, 4, 15,
      TextArea.SCROLLBARS_BOTH));
    add(new TextArea (s, 4, 15,
      TextArea.SCROLLBARS_HORIZONTAL_ONLY));
    add(new TextArea (s, 4, 15,
      TextArea.SCROLLBARS_VERTICAL_ONLY));
  }
}
```



Common Component Methods

All AWT components share the 100-plus methods inherited from the Component class. Some of the most useful and commonly-used methods are listed below:

- getSize() - Gets current size of component, as a Dimension.< BR>
- Dimension d = someComponent.getSize();
- int height = d.height;
- int width = d.width;

Note: With the Java 2 Platform, you can directly access the width and height using the getWidth() and getHeight() methods. This is more efficient, as the component doesn't need to create a new Dimension object. For example:

```
int height = someComponent.getHeight();  // Java 2 Platform only!
int width  = someComponent.getWidth();   // Java 2 Platform only!
```

If you're using the Java 2 platform, you should only use getSize() if you really need a Dimension object!

- getLocation() - Gets position of component, relative to containing component, as a Point.
- Point p = someComponent.getLocation();
- int x = p.x;
- int y = p.y;

Note: With the Java 2 Platform, you can directly access the x and y parts of the location using getX() and getY(). This is more efficient, as the component doesn't have to create a new Point object. For example:

```
int x = someComponent.getX();  // Java 2 Platform only!
int y = someComponent.getY();  // Java 2 Platform only!
```

If you're using the Java 2 platform, you should only use getLocation() if you really need a Point object!

- getLocationOnScreen() - Gets the position of the component relative to the upper-left corner of the computer screen, as a Point.

- Point p = someComponent.getLocationOnScreen();
- int x = p.x;
- int y = p.y;
- getBounds() - Gets current bounding Rectangle of component.
- Rectangle r = someComponent.getBounds();
- int height = r.height;
- int width = r.width;
- int x = r.x;
- int y = r.y;

This is like a combination of calling getLocation() and getSize(). Note: If you're using the Java 2 Platform and don't really need a Rectangle object, you should use getX(), getY(), getWidth(), and getHeight() instead.

- setEnabled(boolean) - Toggles the state of the component. If set to true, the component will react to user input and appear normal. If set to false, the component will ignore user interaction, and usually appear ghosted or grayed-out.

- setVisible(boolean) - Toggles the visibility state of the component. If set to true, the component will appear on the screen if it is contained in a visible container. If false, the component will not appear on the screen. Note that if a component is marked as not visible, any layout manager that is responsible for that component will usually proceed with the layout algorithm as though the component were not in the parent container! This means that making a component invisible will not simply make it disappear while reserving its space in the GUI. Making the component invisible will cause the layout of its sibling components to readjust.

- setBackground(Color)/setForeground(Color) - Changes component background/foreground colors.

- setFont(Font) - Changes font of text within a component.

Containers

A Container is a Component, so may be nested. Class Panel is the most commonly-used Panel and can be extended to partition GUIs. Class Applet is a specialized Panel for running programs within a browser.

Common Container Methods

Besides the 100-plus methods inherited from the Component class, all Container subclasses inherit the behavior of about 50 common methods of Container (most of which just override a method of Component). While the most common method of Container used add(), has already been briefly discussed, if you need to access the list of components within a container, you may find the getComponentCount(), getComponents(), and getComponent(int) methods helpful.

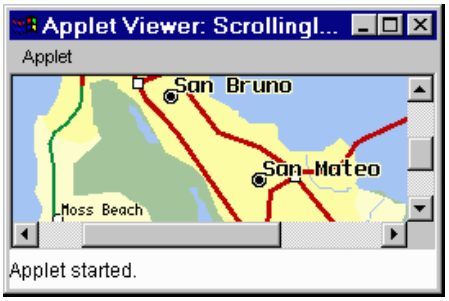
ScrollPane

The ScrollPane container was introduced with the 1.1 release of the Java Runtime Environment (JRE) to provide a new Container with automatic scrolling of any one large Component. That large object could be anything from an image that is too big for the display area to a bunch of spreadsheet cells. All the event handling mechanisms for scrolling are managed for you. Also, there is no LayoutManager for a ScrollPane since there is only a single object within it.

The following example demonstrates the scrolling of a large image. Since an Image object is not a Component, the image must be drawn by a component such as a Canvas .

```
import java.awt.*;
```

```
import java.applet.*;
class ImageCanvas extends Component {
  private Image image;
  public ImageCanvas(Image i) {
    image = i;
  }
  public void paint(Graphics g) {
    if (image != null)
      g.drawImage(image, 0, 0, this);
  }
}

public class ScrollingImage extends Applet {
  public void init() {
    setLayout(new BorderLayout());
    ScrollPane sp =
      new ScrollPane(ScrollPane.SCROLLBARS_ALWAYS);
    Image im =
      getImage(getCodeBase(), "./images/SMarea.gif"); // give any local image url
    sp.add(new ImageCanvas(im));
    add(sp, BorderLayout.CENTER);
  }
}
```



Event Handling

Events

Beginning with the 1.1 version of the JRE, objects register as listeners for events. If there are no listeners when an event happens, nothing happens. If there are twenty listeners registered, each is given an opportunity to process the event, in an undefined order. With a Button, for example, activating the button notifies any registered ActionListener objects.

Consider SimpleButtonEvent applet which creates a Button instance and registers itself as the listener for the button's action events:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class SimpleButtonEvent extends Applet
      implements ActionListener {
  private Button b;

  public void init() {
    b = new Button("Press me");
    b.addActionListener(this);
    add(b);
  }

  public void actionPerformed(ActionEvent e) {
    // If the target of the event was our Button
    // In this example, the check is not
    //   truly necessary as we only listen to
    //   a single button
    if ( e.getSource() == b ) {
      getGraphics().drawString("OUCH",20,20);
    }
  }
}
```

Notice that any class can implement ActionListener, including, in this case, the applet itself. All listeners are always notified. If you don't want an event to be processed further, you can call the AWTEvent.consume() method. However each listener would then need to check for consumption using the isConsumed() method. Consuming events primarily stops events from

being processed by the system, after every listener is notified. So, if you want to reject keyboard input from the user, you can consume() the KeyEvent. All the KeyListener implementers will still be notified, but the character will not be displayed. (Consumption only works for InputEvent and its subclasses.)

So, here is how everything works:

- Components generate subclasses of AWTEvent when something interesting happens.

- Event sources permit any class to be a listener using the addXXXListener() method, where XXX is the event type you can listen for, for example addActionListener(). You can also remove listeners using the removeXXXListener() methods. If there is an add/removeXXXListener() pair, then the component is a source for the event when the appropriate action happens.

- In order to be an event handler you have to implement the listener type, otherwise, you cannot be added, ActionListener being one such type.

- Some listener types are special and require you to implement multiple methods. For instance, if you are interested in key events, and register a KeyListener, you have to implement three methods, one for key press, one for key release, and one for both, key typed. If you only care about key typed events, it doesn't make sense to have to stub out the other two methods. There are special classes out there called adapters that implement the listener interfaces and stub out all the methods. Then, you only need to subclass the adapter and override the necessary method(s).

AWTEvent

Events subclass the AWTEvent class. And nearly every event-type has an associated Listener interface, PaintEvent and InputEvent do not. (With PaintEvent, you just override paint() and update(), for InputEvent, you listen for subclass events, since it is abstract.

Low-level Events

Low-level events represent a low-level input or window operation, like a key press, mouse movement, or window opening. The following table displays the different low-level events, and the operations that generate each event (each operation corresponds to a method of the listener interface):

ComponentEvent	Hiding, moving, resizing, showing
ContainerEvent	Adding/removing component
FocusEvent	Getting/losing focus
KeyEvent	Pressing, releasing, or typing (both) a key
MouseEvent	Clicking, dragging, entering, exiting, moving, pressing, or releasing
WindowEvent	Iconifying, deiconifying, opening, closing, really closed, activating, deactivating

For instance, typing the letter 'A' on the keyboard generates three events, one for pressing, one for releasing, and one for typing. Depending upon your interests, you can do something for any of the three events.

Semantic Events

Semantic events represent interaction with a GUI component; for instance selecting a button, or changing the text of a text field. Which components generate which events is shown in the next section.

ActionEvent	Do the command
AdjustmentEvent	Value adjusted
ItemEvent	State changed
TextEvent	Text changed

Event Sources

The following table represents the different event sources. Keep in mind the object hierarchy. For instance, when Component is an event source for something, so are all its subclasses:

Low-Level Events	
Component	ComponentListener FocusListener KeyListener MouseListener MouseMotionListener
Container	ContainerListener
Window	WindowListener

Semantic Events	
Button List MenuItem TextField	ActionListener
Choice Checkbox Checkbox CheckboxMenuItem List	ItemListener
Scrollbar	AdjustmentListener
TextArea TextField	TextListener

Notice that although there is only one MouseEvent class, the listeners are spread across two interfaces. This is for performance issues. Since motion mouse events are generated more frequently, if you have no interest in them, you can ignore them more easily, without the performance hit.

Event Listeners

Each listener interface is paired with one event type and contains a method for each type of event the event class embodies. For instance, the KeyListener contains three methods, one for each type of event that the KeyEvent has: keyPressed(), keyReleased(), and keyTyped().

Summary of Listener interfaces and their methods

Interface	Method(s)
ActionListener	actionPerformed(ActionEvent e)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
ComponentListener	componentHidden(ComponentEvent e)
	componentMoved(ComponentEvent e)
	componentResized(ComponentEvent e)
	componentShown(ComponentEvent e)
ContainerListen er	componentAdded(ContainerEvent e)
	componentRemoved(ContainerEvent e)
FocusListener	focusGained(FocusEvent e)
	focusLost(FocusEvent e)
ItemListener	itemStateChanged(ItemEvent e)
KeyListener	keyPressed(KeyEvent e)
	keyReleased(KeyEvent e)
	keyTyped(KeyEvent e)
MouseListener	mouseClicked(MouseEvent e)

	mouseEntered(MouseEvent e)
	mouseExited(MouseEvent e)
	mousePressed(MouseEvent e)
	mouseReleased(MouseEvent e)
MouseMotionLi stener	mouseDragged(MouseEvent e)
	mouseMoved(MouseEvent e)
TextListener	textValueChanged(TextEvent e)
WindowListener	windowActivated(WindowEvent e)
	windowClosed(WindowEvent e)
	windowClosing(WindowEvent e)
	windowDeactivated(WindowEvent e)
	windowDeiconified(WindowEvent e)
	windowIconified(WindowEvent e)
	windowOpened(WindowEvent e)

Event Adapters

Since the low-level event listeners have multiple methods to implement, there are event adapter classes to ease the pain. Instead of implementing the interface and stubbing out the methods you do not care about, you can subclass the appropriate adapter class and just override the one or two methods you are interested in. Since the semantic listeners only contain one method to implement, there is no need for adapter classes.

```
public class MyKeyAdapter extends KeyAdapter {
 public void keyTyped(KeyEvent e) {
   System.out.println("User typed: " +
     KeyEvent.getKeyText(e.getKeyCode()));
 }
```

```
}
```

Button Pressing Example

The following code demonstrates the basic concept a little more beyond the earlier example. There are three buttons within a Frame, their displayed labels may be internationalized so you need to preserve their purpose within a command associated with the button. Based upon which button is pressed, a different action occurs.


```
import java.awt.*;
import java.awt.event.*;


public class Activator {
  public static void main(String[] args) {
    Button b;
    ActionListener al = new MyActionListener();
    Frame f = new Frame("Hello Java");
    f.add(b = new Button("Hola"),
                BorderLayout.NORTH);
    b.setActionCommand("Hello");
    b.addActionListener(al);
    f.add(b = new Button("Aloha"),
               BorderLayout.CENTER);
    b.addActionListener(al);
    f.add(b = new Button("Adios"),
                BorderLayout.SOUTH);
    b.setActionCommand("Quit");
    b.addActionListener(al);
    f.pack();
    f.show();
  }
}
```

```
class MyActionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    // Action Command is not necessarily label
    String s = e.getActionCommand();
    if (s.equals("Quit")) {
      System.exit(0);
    }
    else if (s.equals("Hello")) {
      System.out.println("Bon Jour");
    }
    else {
      System.out.println(s + " selected");
    }
  }
}
```

Since this is an application, you need to save the source (as Activator.java), compile it, and run it outside the browser. Also, if you wanted to avoid checking which button was selected, you can associate a different ActionListener to each button, instead of one to all. This is actually how many Integrated Development Environments (IDEs) generate their code.

Adapters Example

The following code demonstrates using an adapter as an anonymous inner class to draw a rectangle within an applet. The mouse press signifies the top left corner to draw, with the mouse release the bottom right.

```
 import java.awt.*;
import java.awt.event.*;

public class Draw extends java.applet.Applet {
  public void init() {
    addMouseListener(
       new MouseAdapter() {
```

```
        int savedX, savedY;
        public void mousePressed(MouseEvent e) {
          savedX = e.getX();
          savedY = e.getY();
        }
        public void mouseReleased(MouseEvent e) {
          Graphics g = Draw.this.getGraphics();
          g.drawRect(savedX, savedY,
                  e.getX()-savedX,
                  e.getY()-savedY);
        }
      }
    );
  }
}
```

Applications and Menus

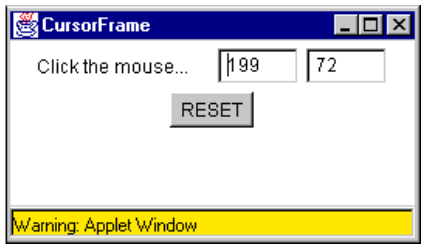
GUI-based Applications

To create a window for your application, define a subclass of Frame (a Window with a title, menubar, and border) and have the main method construct an instance of that class. Applications respond to events in the same way as applets do. The following example, BasicApplication, responds to the native window toolkit quit, or closing, operation:


```
 import java.awt.*;
import java.awt.event.*;


public class BasicApplication extends Frame {
  public BasicApplication() {
    super("BasicApplication Title");
    setSize(200, 200);
    // add a demo component to this frame
    add(new Label("Application Template...",
```

```
                    Label.CENTER),
    BorderLayout.CENTER);
   addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
     setVisible(false); dispose();
     System.exit(0);
    }
   });
  }

  public static void main(String[] args) {
   BasicApplication app =
            new BasicApplication();
   app.setVisible(true);
  }
}
```

Consider an application that displays the x,y location of the last mouse click and provides a button to reset the displayed x,y coordinates to 0,0:




```
 import java.awt.*;
import java.awt.event.*;
public class CursorFrame extends Frame {
  TextField a, b;
  Button btn;
  public CursorFrame() {
   super("CursorFrame");
   setSize(400, 200);
   setLayout(new FlowLayout());
   add(new Label("Click the mouse..."));
   a = new TextField("0", 4);
   b = new TextField("0", 4);
   btn = new Button("RESET");
   add(a); add(b); add(btn);
   addMouseListener(new MouseAdapter() {
```

```
    public void mousePressed(MouseEvent e) {
      a.setText(String.valueOf(e.getX()));
      b.setText(String.valueOf(e.getY()));
    }
   });
   addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      setVisible(false);
      dispose();
      System.exit(0);
    }
   });
   btn.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      a.setText("0");
      b.setText("0");
    }
   });
  }
  public static void main(String[] args) {
   CursorFrame app = new CursorFrame();
   app.setVisible(true);
  }
}
```

This application provides anonymous classes to handle mouse events, application window closing events, and the action event for resetting the text fields that report mouse coordinates.

When you have a very common operation, such as handling application window closing events, it often makes sense to abstract out this behavior and handle it elsewhere. In this case, it's logical to do this by extending the existing Frame class, creating the specialization AppFrame:

```
import java.awt.*;
import java.awt.event.*;

public class AppFrame extends Frame
    implements WindowListener {
  public AppFrame(String title) {
    super(title);
    addWindowListener(this);
  }
  public void windowClosing(WindowEvent e) {
    setVisible(false);
    dispose();
    System.exit(0);
  }
  public void windowClosed(WindowEvent e) {}
  public void windowDeactivated(WindowEvent e) {}
  public void windowActivated(WindowEvent e) {}
  public void windowDeiconified(WindowEvent e) {}
  public void windowIconified(WindowEvent e) {}
  public void windowOpened(WindowEvent e) {}
}
```

AppFrame directly implements WindowListener, providing empty methods for all but one window event, namely, the window closing operation. With this definition, applications such as CursorFrame can extend AppFrame instead of Frame and avoid having to provide the anonymous class for window closing operations:

Applications: Dialog Boxes

A Dialog is a window that requires input from the user. Components may be added to the Dialog like any other container. Like a Frame, a Dialog is initially invisible. You must call the method setVisible() to activate the dialog box.


```
 import java.awt.*;
import java.awt.event.*;

public class DialogFrame extends AppFrame {
 Dialog d;

 public DialogFrame() {
  super("DialogFrame");
  setSize(200, 100);
  Button btn, dbtn;
  add(btn = new Button("Press for Dialog Box"),
   BorderLayout.SOUTH);
  d = new Dialog(this, "Dialog Box", false);
  d.setSize(150, 150);
  d.add(new Label("This is the dialog box."),
   BorderLayout.CENTER);
  d.add(dbtn = new Button("OK"),
   BorderLayout.SOUTH);
  btn.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    d.setVisible(true);
   }
  });
  dbtn.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
    d.setVisible(false);
   }
  });
```

```
    d.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        d.setVisible(false);
      }
    });
  }
  public static void main(String[] args) {
    DialogFrame app = new DialogFrame();
    app.setVisible(true);
  }
}
```

Again, you can define anonymous classes on the fly for:

1. Activating the dialog window from the main application's command button.
2. Deactivating the dialog window from the dialog's command button.
3. Deactivating the dialog window in response to a native window system's closing operation.

Although the anonymous class functionality is quite elegant, it is inconvenient to have to repeatedly include the window-closing functionality for every dialog instance that your applications instantiate by coding and registering the anonymous window adapter class. As with AppFrame, you can define a specialization of Dialog that adds this functionality and thereafter simply use the enhanced class. For example, WMDialog provides this functionality:

```
 import java.awt.*;
import java.awt.event.*;
public class WMDialog extends Dialog
    implements WindowListener {
  public WMDialog(Frame ref, String title, boolean modal) {
    super(ref, title, modal);
    addWindowListener(this);
  }
```

```
 public void windowClosing(WindowEvent e) {
   setVisible(false);
 }
 public void windowClosed(WindowEvent e) {}
 public void windowDeactivated(WindowEvent e) {}
 public void windowActivated(WindowEvent e) {}
 public void windowDeiconified(WindowEvent e) {}
 public void windowIconified(WindowEvent e) {}
 public void windowOpened(WindowEvent e) {}
}
```

Applications: Menus

An application can have a MenuBar object containing Menu objects that are comprised of MenuItem objects. Each MenuItem can be a string, menu, checkbox, or separator (a line across the menu).

To add menus to any Frame or subclass of Frame:

- Create a MenuBar

  ```
  MenuBar mb = new MenuBar();
  ```

- Create a Menu

  ```
  Menu m = new Menu("File");
  ```

- Create your MenuItem choices and add each to the Menu , in the order you want them to appear, from top to bottom.

  ```
  m.add(new MenuItem("Open"));
  m.addSeparator();     // add a separator
  m.add(new CheckboxMenuItem("Allow writing"));
  // Create submenu
  Menu sub = new Menu("Options...");
  ```

```
sub.add(new MenuItem("Option 1"));
m.add(sub);          // add sub to File menu
```

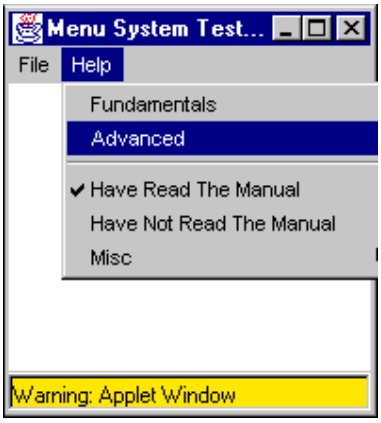
- Add each Menu to the MenuBar in the order you want them to appear, from left to right.

```
mb.add(m);          // add File menu to bar
```

- Add the MenuBar to the Frame by calling the setMenuBar() method.

```
setMenuBar(mb); // set menu bar of your Frame
```

The following program, MainWindow, creates an application window with a menu bar and several menus using the strategy outlined above:




```
import java.awt.*;
import java.awt.event.*;

// Make a main window with two top-level menus: File and Help.
// Help has a submenu and demonstrates a few interesting menu
items.
public class MainWindow extends Frame {
  public MainWindow() {
    super("Menu System Test Window");
    setSize(200, 200);

    // make a top level File menu
    FileMenu fileMenu = new FileMenu(this);
     // make a top level Help menu
    HelpMenu helpMenu = new HelpMenu(this);

    // make a menu bar for this frame
    // and add top level menus File and Menu
    MenuBar mb = new MenuBar();
    mb.add(fileMenu);
    mb.add(helpMenu);
    setMenuBar(mb);
```

```
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        exit();
      }
    });
  }
  public void exit() {
    setVisible(false); // hide the Frame
    dispose(); // tell windowing system to free resources
    System.exit(0); // exit
  }
  public static void main(String args[]) {
    MainWindow w = new MainWindow();
    w.setVisible(true);
  }
}
// Encapsulate the look and behavior of the File menu
class FileMenu extends Menu implements ActionListener {
  MainWindow mw;  // who owns us?
  public FileMenu(MainWindow m) {
    super("File");
    mw = m;
    MenuItem mi;
    add(mi = new MenuItem("Open"));
    mi.addActionListener(this);
    add(mi = new MenuItem("Close"));
    mi.addActionListener(this);
    add(mi = new MenuItem("Exit"));
    mi.addActionListener(this);
  }
  // respond to the Exit menu choice
  public void actionPerformed(ActionEvent e) {
    String item = e.getActionCommand();
```

```
    if (item.equals("Exit"))
      mw.exit();
    else
      System.out.println("Selected FileMenu " + item);
  }
}
 // Encapsulate the look and behavior of the Help menu
class HelpMenu extends Menu implements ActionListener {
  MainWindow mw;  // who owns us?
  public HelpMenu(MainWindow m) {
    super("Help");
    mw = m;
    MenuItem mi;
    add(mi = new MenuItem("Fundamentals"));
    mi.addActionListener(this);
    add(mi = new MenuItem("Advanced"));
    mi.addActionListener(this);
    addSeparator();
    add(mi = new CheckboxMenuItem("Have Read The Manual"));
    mi.addActionListener(this);
    add(mi = new CheckboxMenuItem("Have Not Read The Manual"));
    mi.addActionListener(this);

    // make a Misc sub menu of Help menu
    Menu subMenu = new Menu("Misc");
    subMenu.add(mi = new MenuItem("Help!!!"));
    mi.addActionListener(this);
    subMenu.add(mi = new MenuItem("Why did that happen?"));
    mi.addActionListener(this);
    add(subMenu);
  }
  // respond to a few menu items
  public void actionPerformed(ActionEvent e) {
```

```
    String item = e.getActionCommand();
    if (item.equals("Fundamentals"))
      System.out.println("Fundamentals");
    else if (item.equals("Help!!!"))
      System.out.println("Help!!!");
    // etc...
  }
}
```

2.5 JFC AND SWING

Introduction

The GUI programming in Java 1.0 was basically done using AWT. The AWT provides a set of primitive user interface components. These include elements such as buttons, labels and text fields. AWT also provided higher user level interface components such as lists and choice boxes, classes for creating frames and dialog boxes and classes to support menus that may be attached to a frame.

The original AWT was a peer-based toolkit, meaning that each java AWT component created a peer that was implemented in native code for the platform on which Java Virtual Machine (JVM) was executing. The peer in turn was created in the window in which the component was displayed. The process of requiring each interface component to create its own window made the AWT classes difficult to port to different platforms. This puts a constraint on the set of components that can be displayed on different platforms. For e.g. if any component is not supported by a particular platform, then this component cannot be a sub set of AWT. AWT also contained support classes like AWT Layout Manager classes and geometry classes, which were not directly user interface components.

With Java 1.1 lightweight components were introduced in the AWT. A lightweight component is created by extending the AWT Component or Container class directly. The

lightweight components are not dependent on their peers in the target platforms. The earlier AWT mainly consisted of heavyweight components, which are dependent on the peers in the target platforms. With the introduction of lightweight components, pure Java lightweight component toolkits was now possible. Eliminating the requirement for native peers provided a way for the single Java implementation of a component that can be used for all targeted platforms.

Java Foundation Classes

Sun Microsystems is leveraging the technology of Netscape Communications, IBM, and Lighthouse Design (now owned by Sun) to create a set of Graphical User Interface (GUI) classes that integrate with JDK 1.1.5+, are standard with the Java 2 platform and provide a more polished look and feel than the standard AWT component set. The collection of APIs coming out of this effort, called the Java Foundation Classes (JFC), allows developers to build full-featured enterprise-ready applications. So, JFC is set of APIs for building the java GUI components. JDK 1.2 integrates JFC 1.1 as core API and adds the Java 2D and Drag and Drop APIs.

JFC is composed of five APIs: AWT, Java 2D, Accessibility, Drag and Drop, and Swing. The AWT components refer to the AWT, as it exists in JDK versions 1.1.2 and later. Java 2D is a graphics API designed to provide a set of classes for two dimensional graphics and imaging. The Accessibility API provides assistive technologies, like screen magnifiers, speech recognition for disabled users. Drag and Drop provides interoperability between non java applications and java applications.

Swing is a new feature provided by the JFC 1.1. Actually Swing was the code name used by the JavaSoft project team (that was involved in developing swings) for the next generation of AWT. Swing extends AWT by supplying many more types of GUI components, providing 100% pure Java implementations of these components, and gives the

capability to change the appearance and behavior of these components on different platforms. The Swing components are 100% pure Java. This means that they don't depend on the native window implementation to support them. Swing components are available and consistent across all platforms. Although Swing components are implemented in terms of the underlying AWT, these components do not use AWT components. In fact, all the traditional AWT components are re-implemented as Swing components.

With the use of Model View Architecture (MVC) which is implemented in Swing, you can change the look and Feel of your application. For example, you can have your application designed or running on Microsoft windows to have a look as if its running on the Unix windows platform. This feature of Swing is known as Pluggable Look and Feel (PLAF). Swing PLAF architecture makes it easy to customize both the appearance and the behavior of any Swing control. It also comes with several predefined Look And Feel. eg. MotifPluggableLookAndFeel, windowsPluggableLookAndFeel, MetalPluggableLookAndFeel and MacPluggableLookAndFeel.

Swing Package Overview

javax.swing	The high level swing package primarily consists of components, adapters, default component models, and interfaces for all the delegates and models.
javax.swing.border	The border package declares the Border interface and classes, which define specific border rendering styles.
javax.swing.colorchooser	The colorchooser package contains support classes for the color chooser component.
javax.swing.event .	The event package is for the Swing-specific event types and listeners. In addition to the java.awt.event types, Swing

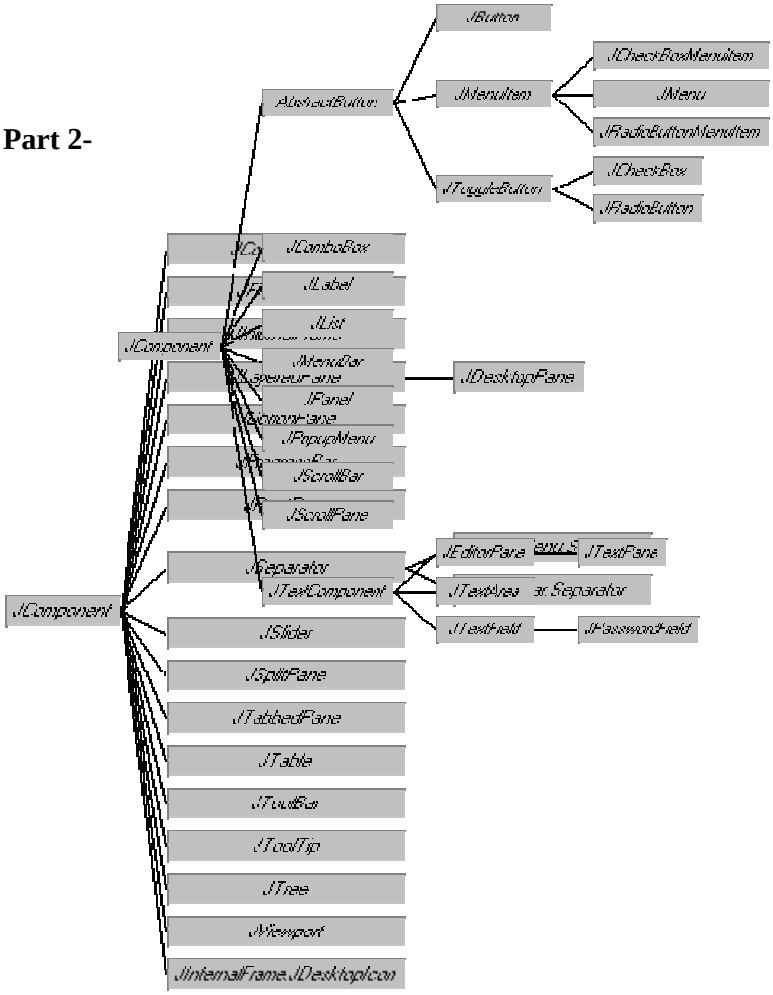
	components can generate their own event types.
javax.swing.filechooser .	The filechooser package contains support classes for the file chooser component
javax.swing.plaf.*	The pluggable look-and-feel (PLAF) packages contain the User Interface (UI) classes (delegates) which implement the different look-and-feel aspects for Swing components. There are also PLAF packages under the javax.swing.plaf hierarchy.
javax.swing.table	The table package contains the support interfaces and classes the Swing table component.
javax.swing.text .	The text package contains the support classes for the Swing document framework
javax.swing.text.html.*	The text.html package contains the support classes for an HTML version 3.2 renderer and parser.
javax.swing.text.rtf	The text.rtf package contains the support classes for a basic Rich Text Format (RTF) renderer.
javax.swing.tree	The tree package contains the interfaces and classes which support the Swing tree component.
javax.swing.undo .	The undo package provides the support classes for implementing undo/redo capabilities in a GUI
javax.accessibility	The JFC Accessibility package is included with the Swing classes.

Component Hierarchy: Part 1--AWT Similar

Part 2-

Component Hierarchy:

-New And Expanded Components



Jcomponent

Swing components are implemented as subclasses of the JComponent class, which inherits from the **Containe**r class. Swing components inherit the following functionality from JComponent:

- 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 small window that appears near the component.

- Look and Feel -Subject to the security restrictions, you can choose the look and feel used by all Swing components by invoking the UIManager.setLookAndFeel( ) method.

- Borders -Using the setBorder() method, you can specify the border that a component displays around its edges.

JFrame

The JFrame class is an extension to the AWT Frame class. An instance of the JFrame class is a heavyweight component. It creates a top-level window that can be positioned and sized independently of other windows. The JFrame instance is managed by the system window manager.

To create an instance of a JFrame class, you can write:

```
JFrame frame=new JFrame ("My Frame");
frame.setSize(300,300); // to give size to a frame
frame.setVisible(true); // to make frame visible
```

It contains the default Java icon on the far left of the titlebar, title in the center, the minimize and maximize buttons, as well as a close button to the far right of the titlebar, The icon can be changed to any image by using the setIconImage method. The current image can be queried with the getIconImage method.

Using JFrame, you can to add the child to the JFrames contentPane as:

```
frame.getContentpane().add(child);
```

Note: Content Pane is a layer on some of the swing components eg. JFrame and Japplet and acts like a container for other components, when it is necessary to have a different look and feel for the same components.

Jinternal Frame

The JInternalFrame class provides the lightweight container that provides many of the features of a native frame, including dragging, closing, becoming an icon, resizing, title display, and support for a menu bar. The fundamental difference between the two classes is that you can add JInternalFrame instance to other frames, and a JFrame instance is a top-level window. The JInternalFrame class extends the JComponent class, while the JFrame class extends the AWT, Frame class.

Jpanel

It is a lightweight Panel object offering built-in support for double buffering. When buffering is enabled, through the constructor, all the drawing operations of components within the panel will be drawn to an off-screen drawing area prior to being drawn to the screen. The JPanel class is used in most of the examples in this section.

Icons

The second component, Icon, isn't really a component at all. However, you can use it with almost all Swing components.An Icon is used to describe fixed-size pictures, or glyphs. Typically, you embed icons in a JButton or other JComponent. Objects that can act as icons implement the Icon interface, shown below. It contains a paintIcon() method that specifies a drawing origin. You render the picture specified in the paintIcon() method in a rectangle whose size cannot exceed a rectangle with an origin at (x, y), a width of getIconWidth(), and a height of getIconHeight(). The Component parameter to paintIcon() is not usually used, unless you need to specify additional information, such as a font or color.

```
public interface Icon {
  void paintIcon( Component c, Graphics g, int x, int y);
  int getIconWidth();
  int getIconHeight();
}
```

The ImageIcon class is an implementation of Icon that creates an Icon from an Image.

```
Icon tinyPicture = new ImageIcon("TinyPicture.gif");
```

Alternatively, the ImageIcon constructor can take an Image or URL object or byte array as its parameter, with an optional String description parameter. One nice thing about ImageIcon is it checks a cache before retrieving the image file.

Swing uses ImageIcon rather than Image for two reasons:

- An Image loads asynchronously, creating the need to monitor the loading process (with MediaTracker).
- An Image is not serializable.

In addition to using ImageIcon, you can implement the interface yourself to create your own icons:

```
public class RedOval implements Icon {
  public void paintIcon (Component c, Graphics g, int x, int y) {
    g.setColor(Color.red);
    g.drawOval (x, y, getIconWidth(), getIconHeight());
  }
  public int getIconWidth() {
    return 10;
  }
  public int getIconHeight() {
    return 10;
  }
}
```

JLabel

A JLabel is a single line label similar to java.awt.Label. Additional functionality that a JLabel has is the ability to:

Add an Icon

Set the vertical and horizontal position of text relative to the Icon

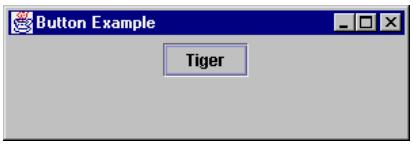
Set the relative position of contents within component

```
public class LabelPanel extends JPanel {
  public LabelPanel() {
    // Create and add a JLabel
    JLabel plainLabel = new JLabel("Plain Small Label");
    add(plainLabel);
    // Create a 2nd JLabel
    JLabel fancyLabel = new JLabel("Fancy Big Label");
    // Instantiate a Font object to use for the label
    Font fancyFont =
      new Font("Serif", Font.BOLD | Font.ITALIC, 32);
    // Associate the font with the label
    fancyLabel.setFont(fancyFont);
    // Create an Icon
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    // Place the Icon in the label
    fancyLabel.setIcon(tigerIcon);
    // Align the text to the right of the Icon
    fancyLabel.setHorizontalAlignment(JLabel.RIGHT);
    // Add to panel
    add(fancyLabel);
  }
}
```

JButton

A JButton can be instantiated and used in a GUI just like a java.awt.Button. It behaves like an AWT 1.1 Button, notifying ActionListener list elements when pushed.

```
public class ButtonPanel extends JPanel {
  public ButtonPanel () {
    JButton myButton = new JButton("Tiger");
    add(myButton);
  }
}
```

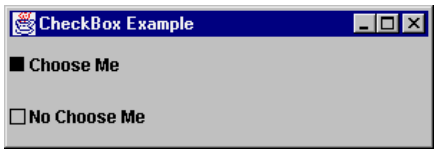


Also, the JButton has support for an embedded Icon, specified in the constructor, or via the setIcon() method. This creates an image button; here, with the label Tiger:

```
public class ButtonPanel extends JPanel {
  public ButtonPanel() {
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    JButton myButton = new JButton("Tiger", tigerIcon);
    add(myButton);
  }
}
```

JCheckBox

A JCheckBox is similar to an AWT Checkbox that is not in a CheckboxGroup. Although Swing provides a default graphic to signify JCheckBox selection, you also can specify your own Icon objects for both the checked and unchecked state.

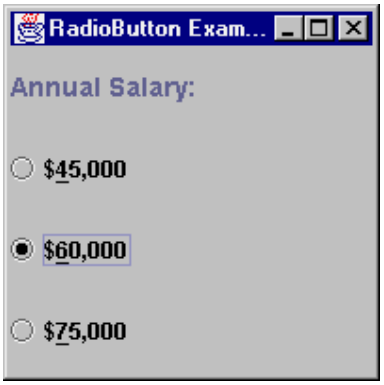



```
public class CheckboxPanel extends JPanel {

  Icon unchecked = new ToggleIcon (false);
  Icon checked = new ToggleIcon (true);

  public CheckboxPanel() {

    // Set the layout for the JPanel
    setLayout(new GridLayout(2, 1));
    // Create checkbox with its state
```

```
  //  initialized to true
  JCheckBox cb1 = new JCheckBox("Choose Me", true);
  cb1.setIcon(unchecked);
  cb1.setSelectedIcon(checked);
  // Create checkbox with its state
  // initialized to false
  JCheckBox cb2 = new JCheckBox(
    "No Choose Me", false);
  cb2.setIcon(unchecked);
  cb2.setSelectedIcon(checked);
  add(cb1);
  add(cb2);
}
class ToggleIcon implements Icon {
  boolean state;
  public ToggleIcon (boolean s) {
    state = s;
  }
  public void paintIcon (Component c, Graphics g,
                         int x, int y) {
    int width = getIconWidth();
    int height = getIconHeight();
    g.setColor (Color.black);
    if (state)
      g.fillRect (x, y, width, height);
    else
      g.drawRect (x, y, width, height);
  }
  public int getIconWidth() {
    return 10;
  }
  public int getIconHeight() {
    return 10;
  }
}
```

JRadioButton

In AWT, radio buttons are checkboxes that belong to the same CheckboxGroup; which ensures that only one checkbox is selected at a time. Swing has a separate widget called a JRadioButton. Each JRadioButton is added to a ButtonGroup so the group behaves as a set of radio buttons. Like CheckboxGroup, ButtonGroup is a functional object that has no visual representation.




```
public class RadioButtonPanel extends JPanel {

 public RadioButtonPanel() {
  // Set the layout to a GridLayout
  setLayout(new GridLayout(4,1));

  // Declare a radio button
  JRadioButton radioButton;

  // Instantiate a ButtonGroup for functional
  // association among radio buttons
  ButtonGroup rbg = new ButtonGroup();

  // Create a label for the group
  JLabel label = new JLabel("Annual Salary: ");
  label.setFont(new Font(
    "SansSerif", Font.BOLD, 14));
  add(label);

  // Add a new radio button to the pane
  radioButton = new JRadioButton("$45,000");
  add (radioButton);
  // set key accelerator
  radioButton.setMnemonic (KeyEvent.VK_4);

  // Add the button to the ButtonGroup
  rbg.add (radioButton);
```

```
    // Set this radio button to be the default
    radioButton.setSelected(true);

    // Set up two more radio buttons
    radioButton = new JRadioButton("$60,000");
    radioButton.setMnemonic (KeyEvent.VK_6);
    add (radioButton);
    rbg.add (radioButton);
    radioButton = new JRadioButton("$75,000");
    radioButton.setMnemonic (KeyEvent.VK_7);
    add (radioButton);
    rbg.add (radioButton);
  }
}
```

Technically speaking, you can add JCheckBox or JToggleButton (described next) components to a CheckboxGroup. At most, one will be selected while in the group.

JToggleButton

The JToggleButton class is the parent to both JCheckBox and JRadioButton. It doesn't have an AWT equivalent. The JToggleButton works like a Button that stays pressed in when toggled on. When a JToggleButton is toggled off, you cannot tell it from a regular Button or JButton class.


```
public class ToggleButtonPanel extends JPanel {
  public ToggleButtonPanel() {
    // Set the layout to a GridLayout
    setLayout(new GridLayout(4,1, 10, 10));
    add (new JToggleButton ("Fe"));
    add (new JToggleButton ("Fi"));
    add (new JToggleButton ("Fo"));
    add (new JToggleButton ("Fum"));
  }
```

```
}
```

JScrollPane

Like the AWT 1.1 ScrollPane, JScrollPane handles automatic horizontal and vertical scrolling of content. It lays out components using a ScrollPaneLayout, described in more detail under Swing Layouts. The key thing to know when using a JScrollPane is that Swing provides a JViewport for adding the object to scroll.

To get a handle to the viewport, JScrollPane has a getViewport() method. Then, to add a component to the viewport, the JViewport class has an add method.

```
JViewport vport = someScrollPane.getViewport();
vport.add(someComponent);
```

Or, more commonly, the two lines are combined:

```
someScrollPane.getViewport().add(someComponent);
```

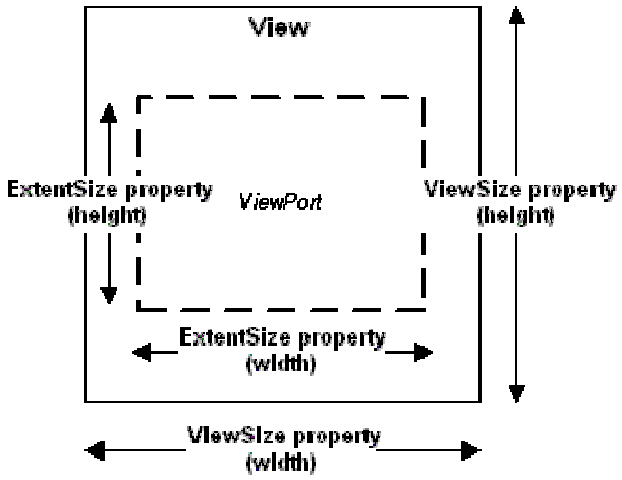
Another option is to provide the component to scroll to the constructor:

```
JScrollPane pane = new JScrollPane(someComponent);
```

```
public class ScrollPanel extends JPanel {

  public ScrollPanel() {
    setLayout(new BorderLayout());
    Icon bigTiger = new ImageIcon("BigTiger.gif");
    JLabel tigerLabel = new JLabel(bigTiger);
    JScrollPane scrollPane =
      new JScrollPane(tigerLabel);
    add(scrollPane, BorderLayout.CENTER);
  }
}
```

Viewports

The JViewport offers a view into a much larger area then can be seen without it. It can be either used within the JScrollPane component or as a standalone widget, where you control all the scrolling functionality yourself. Normally, you wouldn't want to do all the scrolling functionality yourself, but the capability is available.



(Besides JScrollPane, JViewport is used internally within the Swing text components to handle scrolling of text.)

JTextComponents

JTextComponent is a generalized text class that contains all the features you would expect from a simple editor. Some of its methods include:

copy()

cut()

paste()

getSelectedText()

setSelectionStart()

setSelectionEnd()

selectAll()

replaceSelection()

getText()

setText()

setEditable()

setCaretPosition()

Although you won't instantiate a JTextComponent object directly, you will often use these methods, many of which are not available in AWT text widgets. JTextComponent objects in Swing can be placed in a panel in a fashion nearly identical to AWT text widgets.

There are three basic subclasses of JTextComponent: JTextField, JTextArea, and JEditorPane. JPasswordField and JTextPane are sub-subclasses that are also of interest. If you want your users to be able to see content that exceeds the screen display area, you must place the component inside of a JScrollPane to support scrolling to the extra content.

JTextField & JTextArea

Other than having to add a JTextArea to a JScrollPane for scrolling, JTextField and JTextArea behave very similarly to their AWT counterparts: java.awt.TextField and java.awt.TextArea:

```
// Instantiate a new TextField
JTextField tf = new JTextField();
// Instantiate a new TextArea
JTextArea ta = new JTextArea();
// Initialize the text of each
tf.setText("TextField");
ta.setText("JTextArea\n Allows Multiple Lines");
add(tf);
add(new JScrollPane(ta));
```

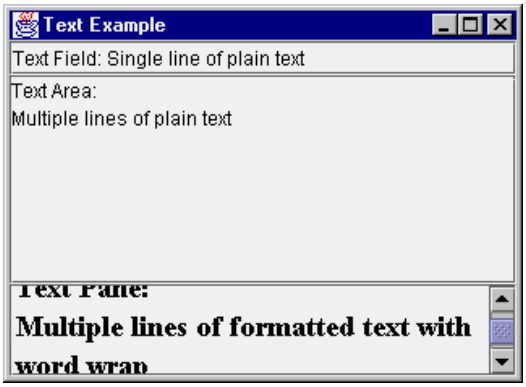
The JTextField also supports setting of text justification with setHorizontalAlignment(). The three available settings are LEFT, CENTER, and RIGHT, where LEFT is the default.

JTextPane

JTextPane is a full-featured text editor that supports formatted text, word wrap, and image display. It uses a linked list of objects that implement the Style interface to specify formatting and supplies some convenience methods for formatting text.

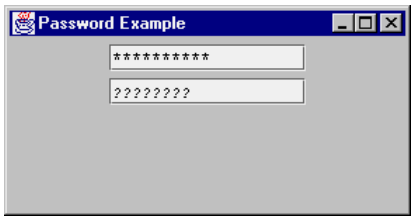
```
JTextPane tp = new JTextPane();
MutableAttributeSet attr = new SimpleAttributeSet();
```

```
StyleConstants.setFontFamily(attr, "Serif");
StyleConstants.setFontSize(attr, 18);
StyleConstants.setBold(attr, true);
tp.setCharacterAttributes(attr, false);
add(new JScrollPane(tp));


public class TextPanel extends JPanel {
  public TextPanel() {
    // Set the layout to a BorderLayout
    setLayout(new BorderLayout());

    // Create the three basic text components
    JTextField textField = new JTextField();
    JTextArea textArea = new JTextArea();
    JTextPane textPane = new JTextPane();

    //Set the textpane's font properties
    MutableAttributeSet attr =
      new SimpleAttributeSet();
    StyleConstants.setFontFamily(attr, "Serif");
    StyleConstants.setFontSize(attr, 18);
    StyleConstants.setBold(attr, true);
    textPane.setCharacterAttributes(attr, false);

    add(textField, BorderLayout.NORTH);
    add(new JScrollPane(textArea),
      BorderLayout.CENTER);
    add(new JScrollPane(textPane), BorderLayout.SOUTH);
  }
}
```



JPasswordField

The JPasswordField is a JTextField that refuses to display its contents openly. By default, the mask character is the asterisk ('*'). However, you can change this with the setEchoChar() method. Unlike java.awt.TextField, an echo character of (char)0 does not unset the mask.



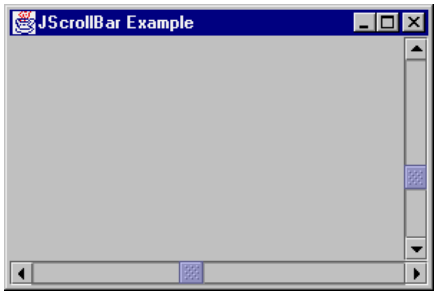

```
class PasswordPanel extends JPanel {
  PasswordPanel() {
    JPasswordField pass1 = new JPasswordField(20);
    JPasswordField pass2 = new JPasswordField(20);
    pass2.setEchoChar ('?');
    add(pass1);
    add(pass2);
  }
}
```

JEditorPane

The JEditorPane class is a specialized JTextComponent for displaying and editing HTML 3.2 tags or some other format like RTF (rich text format), as determined by the input. It is not meant to provide a full-fledged browser, but a lightweight HTML viewer, usually for the purpose of displaying help text. You either construct the pane with a URL parameter (via a String or URL), or change pages with the setPage() method. For HTML content, links within the HTML page are traversable with the help of a HyperlinkListener.

JScrollBar

JScrollBar offers a lightweight version of the java.awt.Scrollbar component.

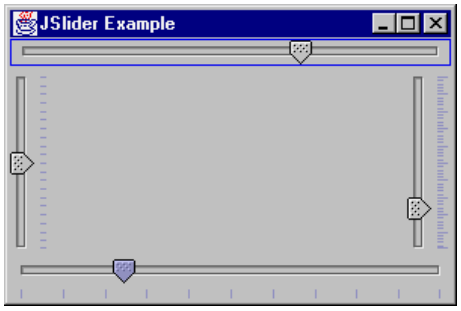



```
public class ScrollbarPanel extends JPanel {


  public ScrollbarPanel() {
    setLayout(new BorderLayout());
    JScrollBar scrollBar1 = new JScrollBar (
      JScrollBar.VERTICAL, 0, 5, 0, 100);
    add(scrollBar1, BorderLayout.EAST);
    JScrollBar scrollBar2 = new JScrollBar (
      JScrollBar.HORIZONTAL, 0, 5, 0, 100);
```

```
    add(scrollBar2, BorderLayout.SOUTH);
  }
}
```

JSlider

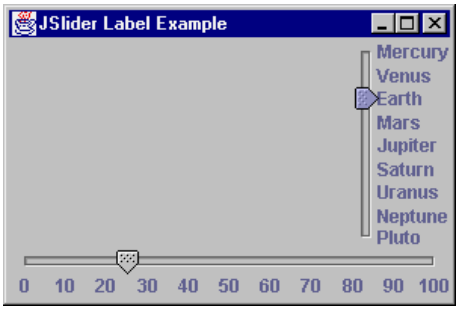
JSlider functions like a JScrollBar; however, it adds the ability to display major and minor tick marks, as well as display a Border around the slider.




```
public class SliderPanel extends JPanel {
  public SliderPanel() {
    setLayout(new BorderLayout());
    JSlider slider1 =
      new JSlider (JSlider.VERTICAL, 0, 100, 50);
    slider1.setPaintTicks(true);
    slider1.setMajorTickSpacing(10);
    slider1.setMinorTickSpacing(2);
    add(slider1, BorderLayout.EAST);
    JSlider slider2 =
      new JSlider (JSlider.VERTICAL, 0, 100, 50);
    slider2.setPaintTicks(true);
    slider2.setMinorTickSpacing(5);
    add(slider2, BorderLayout.WEST);
    JSlider slider3 =
      new JSlider (JSlider.HORIZONTAL, 0, 100, 50);
    slider3.setPaintTicks(true);
    slider3.setMajorTickSpacing(10);
    add(slider3, BorderLayout.SOUTH);
    JSlider slider4 =
      new JSlider (JSlider.HORIZONTAL, 0, 100, 50);
    slider4.setBorder(
      BorderFactory.createLineBorder(Color.blue));
    add(slider4, BorderLayout.NORTH);
  }
}
```

In addition to plain tick marks, with JSlider you can place labels along the axis as either a serious of numbers or components. For numeric labels, by just calling setPaintLabels (true), the slider will generate and use a series of labels based on the major tick spacing. So, if the slider range is 0 to 100 with tick spacing of 10, the slider would then have labels of 0, 10, 20, ... 100. On the other hand, if you want to generate the labels yourself, you can provide a Hashtable of labels. The hashtable key would be the Integer value of the position. The hashtable value would be a Component to use for display of the label. The following demonstrates both:

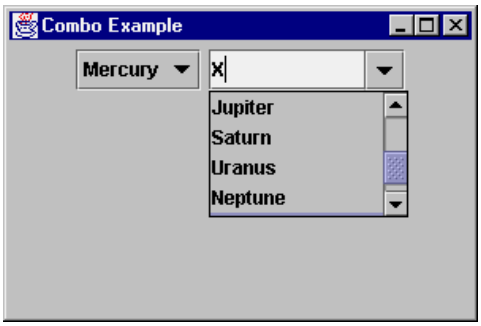
```
public class SliderPanel2 extends JPanel {
  public SliderPanel2() {
    setLayout(new BorderLayout());
    JSlider right, bottom;
    right = new JSlider(JSlider.VERTICAL, 1, 9, 3);
    Hashtable h = new Hashtable();
    h.put (new Integer (1), new JLabel("Mercury"));
    h.put (new Integer (2), new JLabel("Venus"));
    h.put (new Integer (3), new JLabel("Earth"));
    h.put (new Integer (4), new JLabel("Mars"));
    h.put (new Integer (5), new JLabel("Jupiter"));
    h.put (new Integer (6), new JLabel("Saturn"));
    h.put (new Integer (7), new JLabel("Uranus"));
    h.put (new Integer (8), new JLabel("Neptune"));
    h.put (new Integer (9), new JLabel("Pluto"));
    right.setLabelTable (h);
    right.setPaintLabels (true);
    right.setInverted (true);
    bottom =
      new JSlider(JSlider.HORIZONTAL, 0, 100, 25);
    bottom.setMajorTickSpacing (10);
    bottom.setPaintLabels (true);
    add(right, BorderLayout.EAST);
    add(bottom, BorderLayout.SOUTH);
  }
}
```



JComboBox

The JComboBox works like AWT's Choice component, but renames some methods and offers an editable option. For times when a fixed-list of choices isn't enough, you can offer a JComboBox with a list of default choices, but still permit the entry of another value. The nicest part about this control is that when the user presses the key for the first letter of an entry, it changes the highlighted selection. You can enhance this behavior by providing your own KeySelectionManager, a public inner class of JComboBox.

```
public class ComboPanel extends JPanel {
  String choices[] = {
    "Mercury", "Venus", "Earth",
    "Mars", "Jupiter", "Saturn",
    "Uranus","Neptune", "Pluto"};
  public ComboPanel() {
    JComboBox combo1 = new JComboBox();
    JComboBox combo2 = new JComboBox();
    for (int i=0;i<choices.length;i++) {
      combo1.addItem (choices[i]);
      combo2.addItem (choices[i]);
    }
    combo2.setEditable(true);
    combo2.setSelectedItem("X");
    combo2.setMaximumRowCount(4);
    add(combo1);
    add(combo2);
  }
}
```

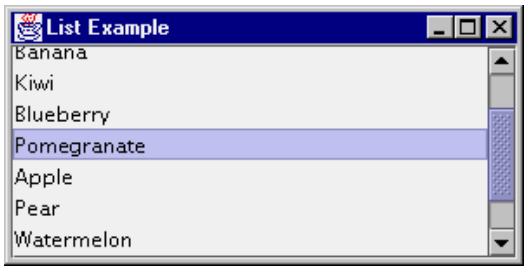


JList

The JList component has both an easy (non-MVC) implementation and a more complicated view. You'll see how to display a list of String objects, just like an AWT List component. To

add a String[] (or Vector) of elements to a JList, just tell the constructor or use the setListData() method. There is one major difference between List and JList. JList doesn't directly support scrolling. You need to place the JList within a JScrollPane object, and let it deal with the scrolling.

```
public class ListPanel extends JPanel {
  String label [] = {"Cranberry", "Orange",
    "Banana", "Kiwi", "Blueberry",
    "Pomegranate", "Apple", "Pear",
    "Watermelon", "Raspberry", "Snozberry"
  };
  public ListPanel() {
    setLayout (new BorderLayout());
    JList list = new JList(label);
    JScrollPane pane = new JScrollPane(list);
    add(pane, BorderLayout.CENTER);
  }
}
```



Borders

The javax.swing.border package consists of several objects to draw borders around components. They all implement the Border interface, which consists of three methods:

- public Insets getBorderInsets(Component c)
  Defines the drawable area necessary to draw the border
- public boolean isBorderOpaque()
  Defines if the border area is opaque or transparent
- public void paintBorder (Component c, Graphics g, int x, int y, int width, int height)
  Defines how to draw the border within the specified area. The routine should only draw into the area requested with getBorderInsets().

The border behavior is defined for JComponent, so all subclasses inherit the behavior. Swing provides nine borders, and you can create your own if none of them meets your needs:

AbstractBorder - An abstract class that implements the Border interface BevelBorder - A 3D border that may be raised or lowered

CompoundBorder - A border that can nest multiple borders

EmptyBorder - A border where you specify the reserved space for an undrawn border

EtchedBorder - A border that appears as a groove, instead of raised or lowered

LineBorder - A border for single color borders, with arbitrary thickness

MatteBorder - A border that permits tiling of an icon or color

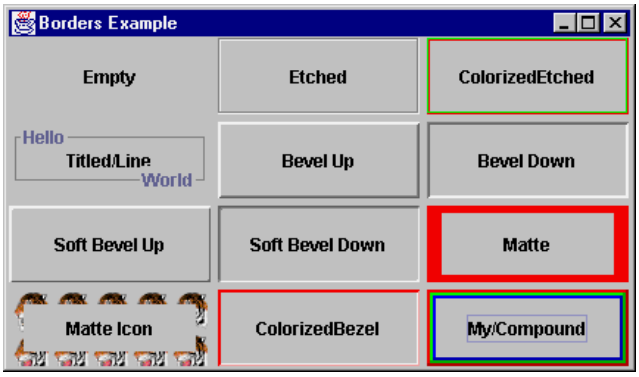
SoftBevelBorder - A 3D border with softened corners

TitledBorder - A border that permits title strings in arbitrary locations

You can create a border object directly from the appropriate class constructor or ask a BorderFactory to create the border for you, with methods like createBevelBorder(type) and createTitledBorder("TItle"). When using BorderFactory, multiple requests to create the same border return the same object.

```
public class BorderPanel extends JPanel {

 class MyBorder implements Border {
   Color color;
   public MyBorder (Color c) {
     color = c;
   }
   public void paintBorder (Component c, Graphics g,
       int x, int y, int width, int height) {
     Insets insets = getBorderInsets(c);
     g.setColor (color);
     g.fillRect (x, y, 2, height);
     g.fillRect (x, y, width, 2);
     g.setColor (color.darker());
     g.fillRect (x+width-insets.right, y, 2, height);
     g.fillRect (x, y+height-insets.bottom, width, 2);
   }
   public boolean isBorderOpaque() {
```

```
    return false;
  }
  public Insets getBorderInsets(Component c) {
    return new Insets (2, 2, 2, 2);
  }
}
public BorderPanel() {
  setLayout (new GridLayout (4, 3, 5, 5));
  JButton b = new JButton("Empty");
  b.setBorder (new EmptyBorder (1,1,1,1));
  add(b);
  b = new JButton ("Etched");
  b.setBorder (new EtchedBorder ());
  add(b);
  b = new JButton ("ColorizedEtched");
  b.setBorder (new EtchedBorder (Color.red,
                                 Color.green));
  add(b);
  b = new JButton ("Titled/Line");
  b.setBorder(new TitledBorder (
    new TitledBorder(
      LineBorder.createGrayLineBorder(),
      "Hello"),
    "World",
    TitledBorder.RIGHT,
    TitledBorder.BOTTOM));
  add(b);
  b = new JButton ("Bevel Up");
  b.setBorder(new BevelBorder(BevelBorder.RAISED));
  add(b);
  b = new JButton ("Bevel Down");
  b.setBorder(new BevelBorder(BevelBorder.LOWERED));
  add(b);
  b = new JButton ("Soft Bevel Up");
  b.setBorder(
    new SoftBevelBorder(SoftBevelBorder.RAISED));
```



```
    add(b);
    b = new JButton ("Soft Bevel Down");
    b.setBorder(
      new SoftBevelBorder(SoftBevelBorder.LOWERED));
    add(b);
    b = new JButton ("Matte");
    b.setBorder(
      new MatteBorder(5, 10, 5, 10, Color.red));
    add(b);
    b = new JButton ("Matte Icon");
    Icon icon = new ImageIcon ("SmallTiger.gif");
    b.setBorder(new MatteBorder(10, 10, 10, 10, icon));
    add(b);

    b = new JButton ("ColorizedBezel");
    b.setBorder(new BevelBorder(BevelBorder.RAISED,
                Color.red, Color.pink));
    add(b);
    b = new JButton ("My/Compound");
    b.setBorder(new CompoundBorder(
      new MyBorder(Color.red),
      new CompoundBorder (new MyBorder(Color.green),
                  new MyBorder(Color.blue))));
    add(b);
  }
}
```

You can change the border of any JComponent object with the setBorder() method.

Menus

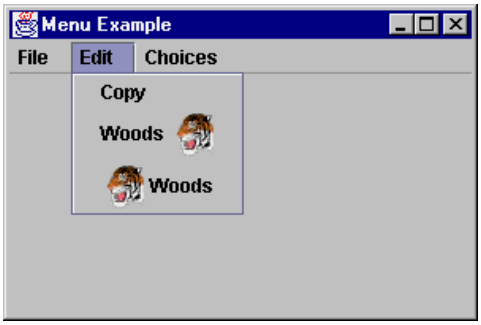
The menuing model used in Swing is nearly identical to that used in AWT. There are three key exceptions:

- The menu classes (JMenuItem, JCheckBoxMenuItem, JMenu, and JMenuBar) are all subclasses of JComponent. They are not off in their own independent class hierarchy.

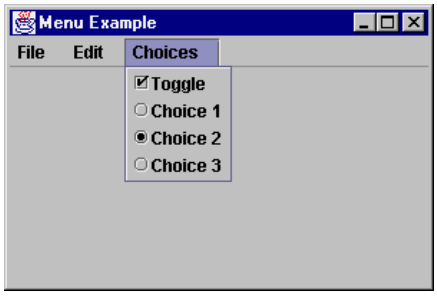
As a result of this, you can place a JMenuBar within any Container, including Applet. [The JApplet class has a setJMenuBar() method to add a JMenuBar.]

- There is a new menu class, JRadioButtonMenuItem, to provide a set of mutually exclusive checkboxes on a menu, when placed within a ButtonGroup.
- Also, you can associate an Icon object with any JMenuItem.

```
public class MenuTester extends JFrame
   implements ActionListener {

  public void actionPerformed (ActionEvent e) {
   System.out.println (e.getActionCommand());
  }

  public MenuTester() {
   super ("Menu Example");

   JMenuBar jmb = new JMenuBar();
   JMenu file = new JMenu ("File");
   JMenuItem item;
   file.add (item = new JMenuItem ("New"));
   item.addActionListener (this);
   file.add (item = new JMenuItem ("Open"));
   item.addActionListener (this);
   file.addSeparator();
   file.add (item = new JMenuItem ("Close"));
   item.addActionListener (this);
   jmb.add (file);

   JMenu edit = new JMenu ("Edit");
   edit.add (item = new JMenuItem ("Copy"));
   item.addActionListener (this);
   Icon tigerIcon = new ImageIcon("SmallTiger.gif");
   edit.add (item =
     new JMenuItem ("Woods", tigerIcon));
   item.setHorizontalTextPosition (JMenuItem.LEFT);
   item.addActionListener (this);
```



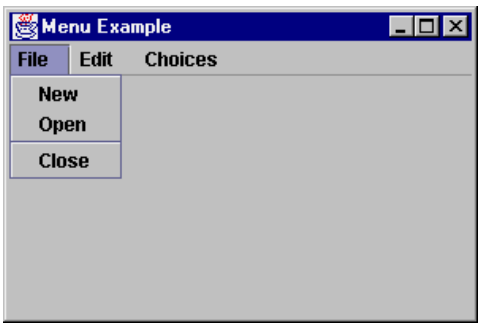
```
    edit.add (item =
      new JMenuItem ("Woods", tigerIcon));
    item.addActionListener (this);
    jmb.add (edit);

    JMenu choice = new JMenu ("Choices");
    JCheckBoxMenuItem check =
      new JCheckBoxMenuItem ("Toggle");
    check.addActionListener (this);
    choice.add (check);
    ButtonGroup rbg = new ButtonGroup();
    JRadioButtonMenuItem rad =
      new JRadioButtonMenuItem ("Choice 1");
    choice.add (rad);
    rbg.add (rad);
    rad.addActionListener (this);
    rad = new JRadioButtonMenuItem ("Choice 2");
    choice.add (rad);
    rbg.add (rad);
    rad.addActionListener (this);
    rad = new JRadioButtonMenuItem ("Choice 3");
    choice.add (rad);
    rbg.add (rad);
    rad.addActionListener (this);

    jmb.add (choice);

    setJMenuBar (jmb);
  }
}
```



Jseparator

The JSeparator object is the menu separator control. The image below shows the separator under the File menu from the example above.

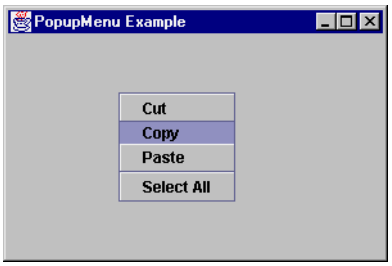
Because Swing menu objects are truly components, you can use JSeparator outside of menus, too. However, normally you just add them to a JMenu with addSeparator().



JpopupMenu

The JPopupMenu component allows you to associate context-sensitive menus with any JComponent. They work similarly to the AWT PopupMenu class, with an addSeparator() method to add a separator bar.

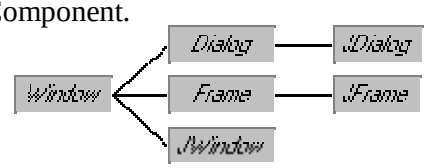
```
public class PopupPanel extends JPanel {
 JPopupMenu popup = new JPopupMenu ();
 public PopupPanel() {
   JMenuItem item;
   popup.add (item = new JMenuItem ("Cut"));
   popup.add (item = new JMenuItem ("Copy"));
   popup.add (item = new JMenuItem ("Paste"));
   popup.addSeparator();
   popup.add (item = new JMenuItem ("Select All"));
   popup.setInvoker (this);
   addMouseListener (new MouseAdapter() {
    public void mousePressed (MouseEvent e) {
     if (e.isPopupTrigger()) {
      popup.show (e.getComponent(),
       e.getX(), e.getY());
     }
    }
    public void mouseReleased (MouseEvent e) {
```



```
      if (e.isPopupTrigger()) {
        popup.show (e.getComponent(),
          e.getX(), e.getY());
      }
    }
  });
 }
}
```

JFrame and Windows

The Window class hierarchy is a little different when the Swing window classes are added. As the diagram shows, they all subclass Window, not JComponent. This means they are not lightweight, have a peer, and cannot be transparent.



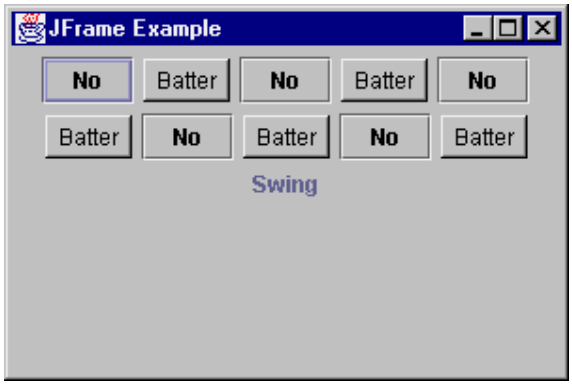
The JFrame class is the replacement for AWT's Frame class. In addition to the ability to add a java.awt.MenuBar via setMenuBar(), you can add a JMenuBar to a JFrame via setJMenuBar().

The other difference of the JFrame class is shared with the JWindow and JDialog classes. No longer do you just add() components to each directly or setLayout() to change the LayoutManager. Now, you must get what's called a content pane, then add components to that or change its layout.

```
public class FrameTester {
 public static void main (String args[]) {
  JFrame f = new JFrame ("JFrame Example");
  Container c = f.getContentPane();
  c.setLayout (new FlowLayout());
  for (int i = 0; i < 5; i++) {
   c.add (new JButton ("No"));
   c.add (new Button ("Batter"));
  }
```

```
    c.add (new JLabel ("Swing"));
    f.setSize (300, 200);
    f.show();
  }
}
```

The reason you have to get a content pane is because the inside of a window is now composed of a JRootPane, which no longer shields you from the inner workings of the Window, as AWT did.



One other difference between JFrame and Frame is JFrame has a property that defines the default close operation. With Frame, nothing happens, by default, if you try to close the frame. On the other hand, JFrame will hide itself when you try to close it. The setDefaultCloseOperation() method lets you define three operations that can happen when the user tries to close a JFrame:

- DO_NOTHING_ON_CLOSE: The AWT Frame behavior
- HIDE_ON_CLOSE: The default behavior. When user tries to close the window, the window will be hidden. You can then setVisible(true) to reshow it.
- DISPOSE_ON_CLOSE: When user tries to close window, it will be disposed.

Both HIDE_ON_CLOSE and DISPOSE_ON_CLOSE perform their operations last, in case an event listener needs to use the information from the closing event.

JrootPane

A JRootPane is a container that consists of two objects, a glass pane and a layered pane. The glass pane is initially invisible, so all you see is the layered pane. The layered pane also

consists of two objects, an optional menu bar and a content pane. You work with the content pane just like you would the inside of a Window, Dialog, or Frame in AWT. The way the glass pane works is if you place a component in it, this component will always display in front of the content pane. This allows things like popup menus and tool tip text to work properly. The layering effect is done with the help of the new JLayeredPane component, explained next.

Normally, the only difference in coding is changing all lines like:

```
aFrame.setLayout (new FlowLayout());
aFrame.add(aComponent);
```

to new lines accessing the content pane:

```
aFrame.getContentPane().setLayout (new FlowLayout());
aFrame.getContentPane().add(aComponent);
```

The rest of the panes are accessed with similar methods, though are rarely accessed directly. The layout management of all these panes is done through a custom layout manager.

```
Container getContentPane();
setContentPane (Container);
Component getGlassPane();
setGlassPane (Component);
JLayeredPane getLayeredPane();
setLayeredPane (JLayeredPane);
JMenuBar getMenuBar();
setMenuBar (JMenuBar);
```

JLayeredPane

The JLayeredPane container keeps its children in layers to define an order to paint its components. When you add a component to the pane, you specify which layer you want it in:

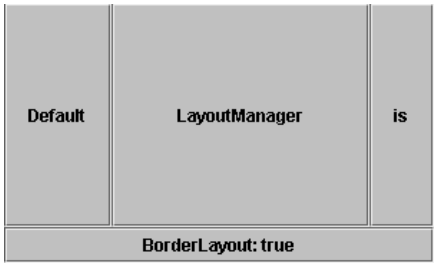
```
layeredPane.add (component, new Integer(5));
```

The default layer is the value JLayeredPane.DEFAULT_LAYER. You can add or subtract values from this value to have things appear above or below, layerwise. The LayoutManager of the pane determines what happens with the layers. Using FlowLayout or GridLayout as the layout only reorders the components as they are added; they will not be drawn on top of each other. For an example of actually drawing overlaid components, see the examples subdirectory that comes with the Swing release.

Swing in Applets

For applets to properly handle the Swing component set, your applets need to subclass JApplet instead of Applet. JApplet is a special subclass of Applet that adds support for JMenuBar and handles the painting support required by Swing child components (along with any other necessary tasks like accessibility support). Also, like JFrame, JApplet has a JContentPane to add components into, instead of directly to the applet. Another difference is the default LayoutManager: in JApplet it is BorderLayout, while in Applet it has always been FlowLayout.

```
public class AppTester extends JApplet {
  public void init () {
    Container c = getContentPane();
    JButton jb = new JButton ("Default");
    c.add (jb, BorderLayout.WEST);
    jb = new JButton ("LayoutManager");
    c.add (jb, BorderLayout.CENTER);
    jb = new JButton ("is");
    c.add (jb, BorderLayout.EAST);
    jb = new JButton ("BorderLayout: " +
      (c.getLayout() instanceof BorderLayout));
    c.add (jb, BorderLayout.SOUTH);
  }
}
```



The LayoutManager is actually a custom subclass of BorderLayout. This subclassing ensures that when a component with no constraints is added, the subclass maps the component to the CENTER area.

Tooltips

A tooltip is a context-sensitive text string that is displayed in a popup window when the mouse rests over a particular object on the screen. Swing provides the JToolTip class to support this; however, you will rarely use it directly. To create a tooltip, you only need to call the setToolTipText() method of JComponent.

```
public class TooltipPanel extends JPanel {
  public TooltipPanel() {
    JButton myButton = new JButton("Hello");
    myButton.setToolTipText ("World");
    add(myButton);
  }
}
```

Toolbars

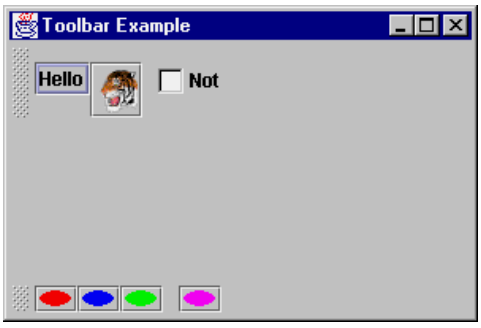
The JToolBar control offers a container that displays its components in a toolbar fashion, across or down, in one row or column, depending upon the area of the screen it is placed in. Certain user-interface models permit floatable toolbars; the default user-interface is one that supports floating. In order to disable the floatable capability, just call the setFloatable() method. It is possible that a particular user interface may ignore this setting.

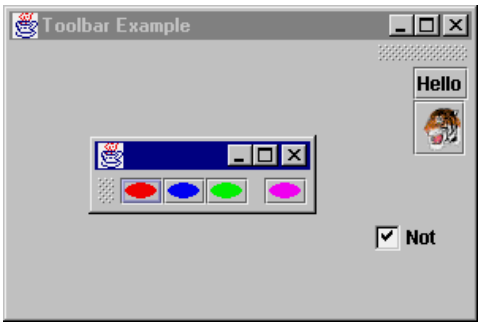
```
// Disabling floating
aToolBar.setFloatable (false);
```

When JToolBar is considered floatable, this means a user can drag it to another area of the screen, or place it in a window external from the original container. To demonstrate a

JToolBar, take a look at the following example. As it demonstrates, there are no restrictions on what components appear within the toolbar. However, it works best if they are all the same type and size.

```
public class ToolbarPanel extends JPanel {
  ToolbarPanel() {
    setLayout (new BorderLayout());
    JToolBar toolbar = new JToolBar();
    JButton myButton = new JButton("Hello");
    toolbar.add(myButton);
    Icon tigerIcon = new ImageIcon("SmallTiger.gif");
    myButton = new JButton(tigerIcon);
    toolbar.add(myButton);
    toolbar.addSeparator();
    toolbar.add (new Checkbox ("Not"));
    add (toolbar, BorderLayout.NORTH);
    toolbar = new JToolBar();
    Icon icon = new AnOvalIcon(Color.red);
    myButton = new JButton(icon);
    toolbar.add(myButton);
    icon = new AnOvalIcon(Color.blue);
    myButton = new JButton(icon);
    toolbar.add(myButton);
    icon = new AnOvalIcon(Color.green);
    myButton = new JButton(icon);
    toolbar.add(myButton);
    toolbar.addSeparator();
    icon = new AnOvalIcon(Color.magenta);
    myButton = new JButton(icon);
    toolbar.add(myButton);
    add (toolbar, BorderLayout.SOUTH);
  }
  class AnOvalIcon implements Icon {
    Color color;
    public AnOvalIcon (Color c) {
```





```
    color = c;
  }
  public void paintIcon (Component c, Graphics g,
                         int x, int y) {
    g.setColor(color);
    g.fillOval (
      x, y, getIconWidth(), getIconHeight());
  }
  public int getIconWidth() {
    return 20;
  }
  public int getIconHeight() {
    return 10;
  }
 }
}
```

After dragging around the toolbar, the user may just leave it looking a little different than you originally planned as shown above.

JtabbedPane

The JTabbedPane component offers a tabbed control for quick accessibility to multiple panels. If you ever tried to use CardLayout in JDK 1.0/1.1, you'll appreciate this: JTabbedPane adds the necessary support for changing from one card to the next. After creating the control, you add cards to it with the addTab() method. There are three forms for the addTab() method. One offers a quick way to associate a JToolTip to a tab, while the others only permit text, an Icon, or both. Any Component subclass can be the object added to each card.

- addTab(String title, Component component) - Create new tab with title as tab label and component shown within tab when selected.

- addTab(String title, Icon icon, Component component) - Adds an options icon to be associated with the title for the tab. Either may be null.
- addTab(String title, Icon icon, Component component, String tip) - Adds tip as the tooltip for the tab.

```
public class TabbedPanel extends JPanel {
  String tabs[] = {"One", "Two", "Three", "Four"};
  public JTabbedPane tabbedPane = new JTabbedPane();
  public TabbedPanel() {
    setLayout (new BorderLayout());
    for (int i=0;i<tabs.length;i++)
      tabbedPane.addTab (tabs[i], null,
                         createPane (tabs[i]));
    tabbedPane.setSelectedIndex(0);
    add (tabbedPane, BorderLayout.CENTER);
  }
  JPanel createPane(String s) {
    JPanel p = new JPanel();
    p.add(new JLabel(s));
    return p;
  }
}
```

Swing Layouts

There are four primary Swing layout managers, two are built into components (ScrollPaneLayout and ViewportLayout) and the remaining two (BoxLayout and OverlayLayout) are used like the ones from java.awt. The BoxLayout also happens to be built into the Box component.

BoxLayout

The BoxLayout layout manager allows you to arrange components along either an x-axis or y-axis. For instance, in a y-axis BoxLayout, components are arranged from top to bottom in the order in which they are added.

Unlike GridLayout, BoxLayout allows components to occupy different amounts of space along the primary axis. A JTextField in a top-to-bottom BoxLayout can take much less space than a JTextArea.

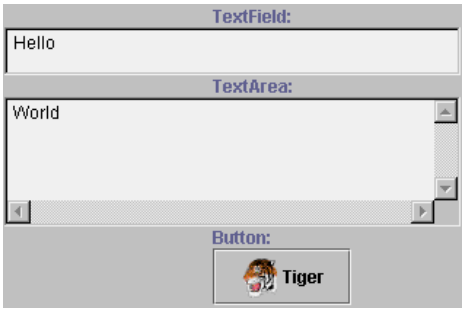
Along the non-primary axis, BoxLayout attempts to make all components as tall as the tallest component (for left-to-right BoxLayouts) or as wide as the widest component (for top-to-bottom BoxLayouts). If a component cannot increase to this size, BoxLayout looks at its Y-alignment property or X-alignment property to determine how to place it within the available space. By default, JComponent objects inherit an alignment of 0.5 indicating that they will be centered. You can override the getAlignmentX() and getAlignmentY() methods of Container to specify a different default alignment. JButton for instance specifies left alignment.

To create a BoxLayout, you must specify two parameters:

```
setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
```

The first parameter specifies the container and the second the major axis of the BoxLayout. Components can then be added as they are in a GridLayout or FlowLayout:

```
  add(myComponent);


class BoxLayoutTest extends JPanel {


  BoxLayoutTest() {
    // Set the layout to a y-axis BoxLayout
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    // Create three components
    TextField textField = new TextField();
    TextArea textArea = new TextArea(4, 20);
    JButton button = new JButton(
```



```
    "Tiger", new ImageIcon("SmallTiger.gif"));

  // Add the three components to the BoxLayout
  add(new JLabel("TextField:"));
  add(textField);
  add(new JLabel("TextArea:"));
  add(textArea);
  add(new JLabel("Button:"));
  add(button);
}}
```

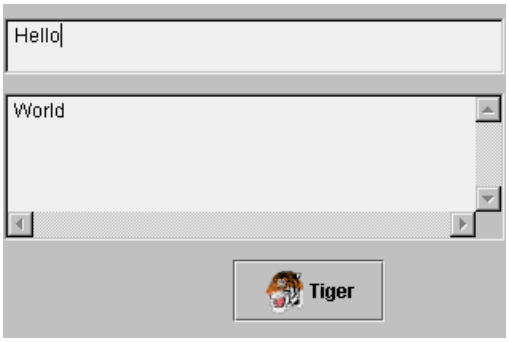
Box

The Box class is a convenience container whose default layout manager is a BoxLayout. Rather than subclassing JPanel as above, the previous example could have subclassed the Box class. In addition to being a BoxLayout container, Box has some very useful static methods for arranging components in a BoxLayout. These methods create non-visual components that act as fillers and spacers.

createVerticalStrut(int)	Returns a fixed height component used for spacing
createHorizontalStrut(int)	Returns a fixed width component used for spacing
createVerticalGlue()	Returns a component whose height expands to absorb excess space between components
createHorizontalGlue()	Returns a component whose width expands to absorb excess space between components
createGlue()	Returns a component whose height will expand for a y-axis box and whose width will expand for an x-axis Box
createRigidArea(Dimension)	Returns a fixed height, fixed width component used for spacing

Now, rather than using labels to space components out as above, you could use struts and glue:

```
public class TestBox extends Box {
  TestBox() {
```



```
    super(BoxLayout.Y_AXIS);

    // Create the three basic text components
    TextField textField = new TextField();
    TextArea textArea = new TextArea(4, 20);
    JButton button = new JButton("Tiger", new
      ImageIcon("SmallTiger.gif"));

    // Separate the three components
    // by struts for spacing
    add(createVerticalStrut(8));
    add(textField);
    add(createVerticalGlue());
    add(textArea);
    add(createVerticalGlue());
    add(button);
    add(createVerticalStrut(8));
  }
}
```

The struts will appear as top and bottom margins and the glue will expand to fill space when the Box is heightened.

ScrollPaneLayout

The ScrollPaneLayout is the layout manager used by a JScrollPane. You do not need to create one, nor associate it to the JScrollPane. That is done for you automatically. The layout defines nine different areas for the JScrollPane:

- one JViewport - in the center for the content
- two JScrollBar objects - one each for horizontal and vertical scrolling
- two JViewport objects - one for a column headers, the other row

- four Component objects - one for each of the corners
  The JScrollPane constants to specify the corners are: LOWER_LEFT_CORNER, LOWER_RIGHT_CORNER, UPPER_LEFT_CORNER, UPPER_RIGHT_CORNER.

The center viewport portion of this layout is of primary interest for simple layouts. A JViewport is itself a container object that can hold components. This allows for some very flexible arrangements. JViewport contains its own layout manager, ViewportLayout.

ViewportLayout

ViewportLayout is the layout manager used by a JViewport. You should never need to use the layout directly, as it is automatically associated with a JViewport object, and positions the internal component for you based upon the JViewport properties.

2.6 LAYOUT MANAGEMENT

Why do you need Layout Managers?

To describe why layout managers are necessary, all you need to do is examine a few of the problems they solve.

- Window Expanded, Components Stay Put
  The first layout sin is to ignore a user-resize. Non-resizable GUIs can be extremely frustrating.

- Components Designed for a Specific Look-And-Feel or Font Size
  Another common problem is expecting all platforms or LookAndFeel libraries (when using Java Foundation Classes (JFC) Project Swing technology) to have the same sizing characteristics.

- Components Designed for a Specific Language

  Some words take up the same amount of space in different languages. "No" is a good example, as it's spelled that way in several languages. However, most words have varying lengths depending on which language you are using.

What are Layout Managers?

A layout manager encapsulates an algorithm for positioning and sizing of GUI components. Rather than building the layout algorithms into your code and watching for window resizing and other layout-modifying events, the algorithm is kept separate. This allows a layout algorithm to be reused for several applications, while simplifying your application code.

LayoutManager is an interface in the Java class libraries that describes how a Container and a layout manager communicate. It describes several methods which:

- Ask for sizing information for the layout manager and the components it manages.
- Tell the layout manager when components are added and removed from the container.
- Size and position the components it manages.

An additional interface, LayoutManager2, was added in JDK 1.1, which adds a few more positioning and validation methods.

Component Size Methods

The Component class defines several size accessor methods to assist the layout process. Each of these methods returns a Dimension object describing the requested size. These methods are as follows:

- public Dimension getPreferredSize()
  This returns the desired size for a component.
- public Dimension getMinimumSize()
  This returns the smallest desired size for a component.
- public Dimension getMaximumSize()
  This returns the largest desired size for a component.

Layout managers will use these sizing methods when they are figuring out where to place components and what size they should be. Layout managers can respect or ignore as much or as little of this information as they see fit. Each layout manager has its own algorithm and may or may not use this information when deciding component placement. Which of these methods is respected or ignored is very important information and should be documented carefully when creating your own layout manager.

Controlling the sizes returned for your own components can be accomplished in two ways, depending on whether you are using the JFC Project Swing components.
If your component is a JFC Project Swing component, you inherit three methods, setPreferredSize(Dimension), setMinimumSize(Dimension) and setMaximumSize(Dimension). You can call these methods directly to explicitly set the size information for a component. For example:

```
JButton okButton = new JButton("Ok");
okButton.setPreferredSize(new Dimension(100,10));
```

If your component is not a JFC Project Swing component, you will need to subclass it to adjust the sizes. For example:

```
public class MyButton extends Button {
  public Dimension getPreferredSize() {
    return new Dimension(100,10);
  }
```

```
}
```

Layout Managers and Containers

A layout manager must be associated with a Container object to perform its work. If a container does not have an associated layout manager, the container simply places components wherever specified by using the setBounds(), setLocation() and/or setSize() methods.

If a container has an associated layout manager, the container asks that layout manager to position and size its components before they are painted. The layout manager itself does not perform the painting; it simply decides what size and position each component should occupy and calls setBounds(), setLocation() and/or setSize() on each of those components.

A LayoutManager is associated with a Container by calling the setLayout(LayoutManager) method of Container. For example:

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
```

Some containers, such as Panel, provide a constructor that takes a layout manager as an argument as well:

```
Panel p = new Panel(new BorderLayout());
```

The add(...) Methods

Containers have several methods that can be used to add components to them. They are:

```
public Component add(Component comp)
public Component add(String name, Component comp)
public Component add(Component comp, int index)
```

```
public void add(Component comp, Object constraints)
public void add(Component comp, Object constraints, int index)
```

Each of these methods adds a component to the container and passes information to the layout manager of the container. All of the methods take a Component parameter, specifying which component to add. Some take an index. This is used to specify an order in the container; some layout managers (such as CardLayout) respect the ordering of added components.

The other parameters, name and constraints are information that can be used by a layout manager to help direct the layout. For example, when adding a component to a container that is managed by a BorderLayout, you specify a compass position as a constraint.

Each of the above add() methods delegates its work to a single addImpl() method:

```
protected void addImpl(
Component comp, Object constraints, int index)
```

Note: addImpl stands for "implementation of the add method."

This method is the one that does all the work. It adds the Component to the Container, and, if a layout manager is managing the container layout, calls the addLayoutComponent() method of the layout manager. It is through addLayoutComponent() that the layout manager receives the constraints (from the add() call).

If you create a subclass of Container and want to override an add() method, you only need to override addImpl(). All other add() methods route through it.

Container Insets

In addition to a layout manager, each Container has a getInsets() method that returns an Insets object. The Insets object has four public fields: top, bottom, left and right.

These insets define the area a container is reserving for its own use (such as drawing a decorative border). Layout managers must respect this area when positioning and sizing the contained components.

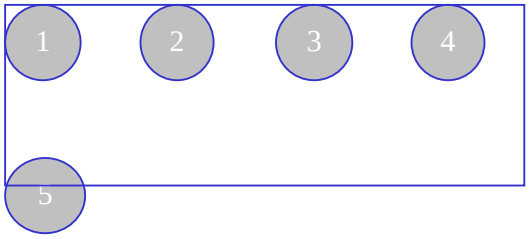
If you are using the JFC Project Swing components, you can use the various Border classes provided to generate these types of effects, instead of using Insets.

The Standard AWT Layout Managers

The AWT library includes five layout managers. These can be used in various combinations to create just about any GUI you may possibly want to write. The standard AWT layout managers are:

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

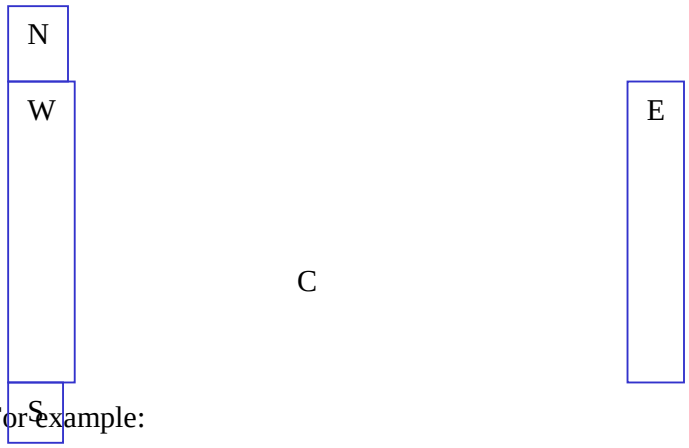
FlowLayout puts components in a row, sized at their preferred size. If the horizontal space in the container is too small to put all the components in one row, FlowLayout uses multiple rows. Within each row, components are centered (the default), left-aligned, or right-aligned as specified when the FlowLayout is created.



Example:

```
toolbar.setLayout(new FlowLayout(FlowLayout.LEFT));
toolbar.add(playButton);
toolbar.add(stopButton);
```

A BorderLayout has five areas: north, south, east, west, and center. If you enlarge the window, the center area gets as much of the available space as possible. The areas expand only as much as necessary to fill all available space. Often, a container uses only one or two of the areas of the BorderLayout -- just the center, or center and south.

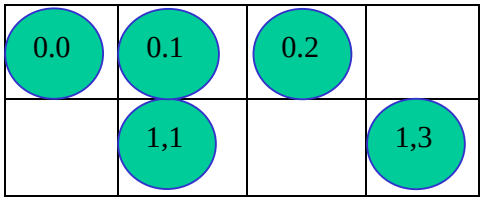


For example:

```
Container contentPane = getContentPane();
//Use the content pane's default BorderLayout.
//contentPane.setLayout(new BorderLayout()); //unnecessary

contentPane.add(new JButton("Button 1 (NORTH)"),
        BorderLayout.NORTH);
contentPane.add(new JButton BorderLayout.CENTER);
contentPane.add(new JButton("2 (CENTER)"),
        ("Button 3 (WEST)"),
        BorderLayout.WEST);
```

```
contentPane.add(new JButton("Long-Named Button 4 (SOUTH)"),
        BorderLayout.SOUTH);
contentPane.add(new JButton("Button 5 (EAST)"),
        BorderLayout.EAST);
```

GridLayout places components in a grid of cells. Each component takes all the available space within its cell, and each cell is exactly the same size. If you resize the GridLayout window, you'll see that the GridLayout changes the cell size so that the cells are as large as possible, given the space available to the container.



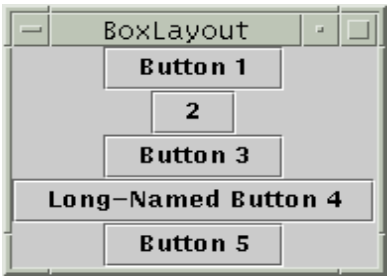

```
contentPanel.setLayout(new
GridLayout(2, 4));
contentPanel.add(startButton, 0, 0);
contentPanel.add(stopButton, 1, 3);
```

Example:

```
Container contentPane = getContentPane()
contentPane.setLayout(new GridLayout(0,2));
contentPane.add(new JButton("Button 1"));
contentPane.add(new JButton("2"));
contentPane.add(new JButton("Button 3"));
ContentPane.add(new JButton("Long-Named Button 4"));
contentPane.add(new JButton("Button 5"));
```

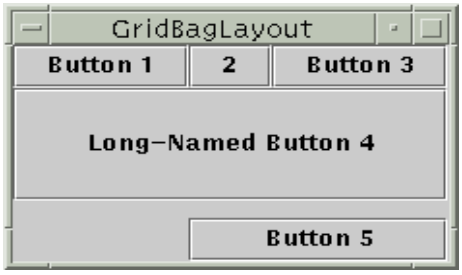
The constructor tells the GridLayout class to create an instance that has two columns and as many rows as necessary.

The Swing packages include a general purpose layout manager named BoxLayout. BoxLayout either stacks its components on top of each other (with the first component at the top) or places them in a tight row from left to right -- your choice. You might think of it as a full-



featured version of FlowLayout. Here is an applet that demonstrates using BoxLayout to display a centered column of components:

By creating one or more lightweight containers that use BoxLayout, you can achieve some layouts for which the more complex GridbagLayout is often used. BoxLayout is also useful in some situations where you might consider using BorderLayout or BorderLayout. One big difference between BoxLayout and the existing AWT layout managers is that BoxLayoutrespects each component's maximum size and X/Y alignment.

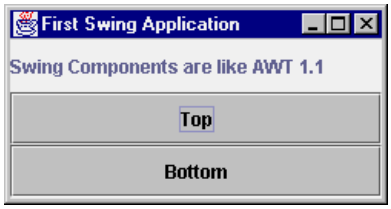


GridBagLayout is the most flexible -- and complex -- layout manager the Java platform provides. A GridBagLayout places components in a grid of rows and columns, allowing specified components to span multiple rows or columns. Not all rows necessarily have the same height. Similarly, not all columns necessarily have the same width. Essentially, GridBagLayout places components in rectangles (cells) in a grid, and then uses the component's preferred sizes to determine how big the cells should be.

EXERCISES

1. Program 1

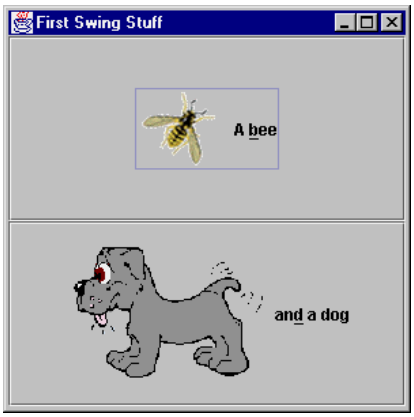
```
/*
 * Demonstrates that basic use of Swing components is just like old AWT versions.  Illustrates components,
frames, layouts, and 1.1 style events.
 */


import javax.swing.*;
import java.awt.*;
import java.awt.event.*;


public class FirstAppl extends Frame {
```




```
  // The initial width and height of the frame
  public static int WIDTH = 250;
  public static int HEIGHT = 130;
  public FirstAppl(String lab) {
    super(lab);
    setLayout(new GridLayout(3,1));
    JButton top = new JButton("Top");
    top.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("top");
      }
    });
    JButton bottom = new JButton("Bottom");
    bottom.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("bottom");
      }
    });
    add(new JLabel("Swing Components are like AWT 1.1"));
    add(top);
    add(bottom);
  }

  public static void main(String s[]) {
    FirstAppl frame = new FirstAppl("First Swing Application");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });

    frame.setSize(WIDTH, HEIGHT);
    frame.show();
  }
}
```

2. Program 2

```
/*
 * Demonstrates ImageIcons with Buttons.  First use of Swing capabilities.
 */
```




```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FirstSwing extends JFrame {
  // The initial width and height of the frame
  public static int WIDTH = 300;
  public static int HEIGHT = 300;

  // images for buttons
  public Icon bee = new ImageIcon("bee.gif");
  public Icon dog = new ImageIcon("dog.gif");

  public FirstSwing(String lab) {
    super(lab);
    JButton top = new JButton("A bee", bee);
    top.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("bee");
      }
    });
    JButton bottom = new JButton("and a dog", dog);
    bottom.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("dog");
      }
    });
```

```
    top.setMnemonic (KeyEvent.VK_B);
    bottom.setMnemonic (KeyEvent.VK_D);

    Container content = getContentPane();
    content.setLayout(new GridLayout(2,1));
    content.add(top);
    content.add(bottom);
  }

  public static void main(String args[]) {
    FirstSwing frame = new FirstSwing("First Swing Stuff");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });
    frame.setSize(WIDTH, HEIGHT);
    frame.setVisible(true);
  }
}
```

3. Program 3

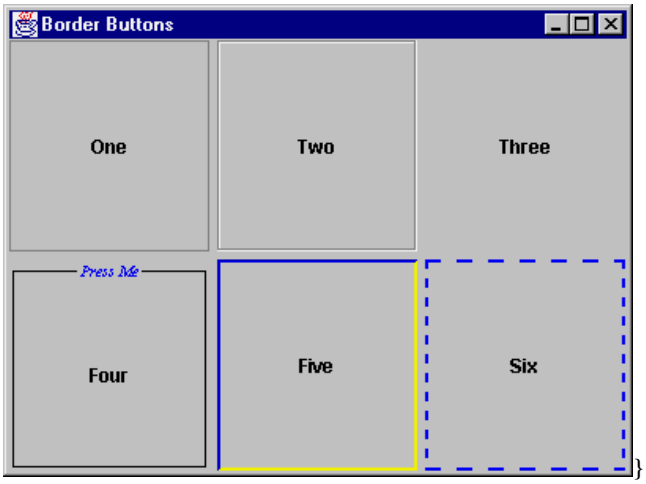
```
/*
 * Demonstrates Borders.  Change the default button
 * border and create your own.
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class Borders extends JFrame {
  // The initial width and height of the frame
  private static int WIDTH = 400;
  private static int HEIGHT = 300;
```

```
public Borders (String title) {
  super(title);
  Container content = getContentPane();
  content.setLayout (new GridLayout (2, 3, 5, 5));
  JButton b;
  b = new JButton("One");
  b.setBorder (LineBorder.createGrayLineBorder());
  content.add (b);
  b = new JButton("Two");
  b.setBorder(new EtchedBorder());
  content.add (b);
  b = new JButton("Three");
  b.setBorder(new EmptyBorder(2,2,2,2));
  content.add (b);
  b = new JButton("Four");
  b.setBorder(new TitledBorder(
    LineBorder.createBlackLineBorder(),
    "Press Me",
    TitledBorder.CENTER,
    TitledBorder.TOP,
    new Font ("Serif", Font.ITALIC, 10),
    Color.blue));
  content.add (b);
  b = new DoubleBorderedButton(
    new BevelBorder (BevelBorder.RAISED, Color.blue, Color.yellow),
    new BevelBorder (BevelBorder.LOWERED, Color.blue, Color.yellow));
  b.setText ("Five");
  content.add (b);
  b = new DoubleBorderedButton(
    new DashedBorder (Color.blue, 10, 6),
    new DashedBorder (Color.magenta, 3, 3));
  b.setText ("Six");
```

```
  content.add (b);
}

public static void main (String args[]) {
  Frame frame = new Borders ("Border Buttons");
  frame.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {System.exit(0);}
  });

  frame.setSize(WIDTH, HEIGHT);
  frame.setVisible(true);
}

class DoubleBorderedButton extends JButton {
  public DoubleBorderedButton (final Border normal, final Border selected) {
    setBorder (normal);
    addMouseListener (new MouseAdapter() {
      public void mousePressed (MouseEvent e) {
        setBorder (selected);
      }
      public void mouseReleased (MouseEvent e) {
        setBorder (normal);
      }
    });
  }
}

class DashedBorder implements Border {
  int THICKNESS = 2;
  Color color;
  int dashWidth;
  int dashHeight;
  public DashedBorder () {
```

```
  this (Color.black, 2, 2);
}
public DashedBorder (Color c, int width, int height) {
  if (width < 1) {
    throw new IllegalArgumentException ("Invalid width: " + width);
  }
  if (height < 1) {
    throw new IllegalArgumentException ("Invalid height: " + height);
  }
  color = c;
  dashWidth = width;
  dashHeight = height;
}
public void paintBorder (Component c, Graphics g, int x, int y, int width, int height) {
  Insets insets = getBorderInsets(c);
  g.setColor (color);
  int numWide = (int)Math.round(width / dashWidth);
  int numHigh = (int)Math.round(height / dashHeight);
  int startPoint;
  for (int i=0;i<=numWide;i+=2) {
    startPoint = x + dashWidth * i;
    g.fillRect (startPoint, y, dashWidth, THICKNESS);
    g.fillRect (startPoint, y+height-insets.bottom, dashWidth, THICKNESS);
  }
  for (int i=0;i<=numHigh;i+=2) {
    startPoint = x + dashHeight * i;
    g.fillRect (x, startPoint, THICKNESS, dashHeight);
    g.fillRect (x+width-insets.right, startPoint, THICKNESS, dashHeight);
  }
}
public Insets getBorderInsets(Component c) {
  return new Insets (THICKNESS, THICKNESS, THICKNESS, THICKNESS);
}
```

```
  public boolean isBorderOpaque() {
    return false;
  }
}
```



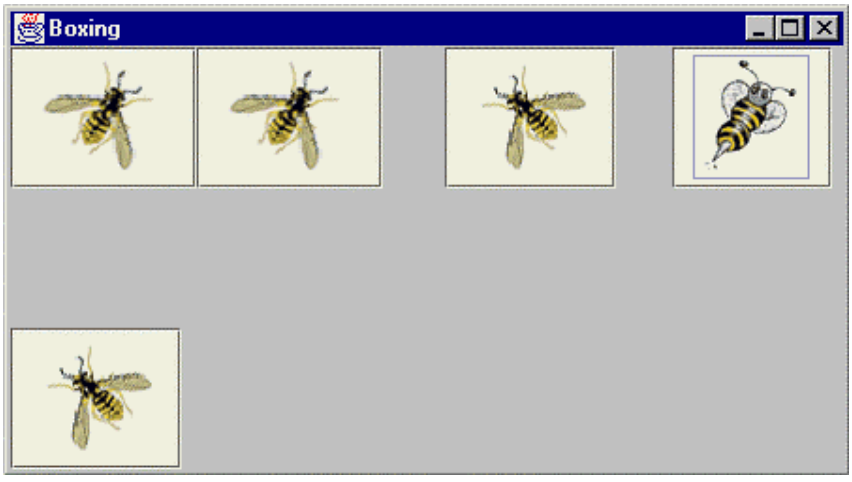
}

4. Program 4

```
/*
 * Demonstrates BoxLayout.  Create vertical box with two buttons at
 * opposite extremes.  Create horizontal box with three buttons
 * evenly spaced across the space except 5 pixels must be left
 * at the right edge of the box.
 */

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.plaf.*;

public class BLayout extends JFrame {
  // The initial width and height of the frame
  private static int WIDTH = 500;
  private static int HEIGHT = 300;
```

```
private static int XSPACE = 5;

// images for buttons
Icon rightB = new ImageIcon("bee-right.gif");
Icon leftB = new ImageIcon("bee-left.gif");
Icon animB  = new ImageIcon("bee-anim.gif");

public BLayout(String lab) {
  super(lab);
  setBackground (Color.lightGray);

  // Change button background to white
  ColorUIResource backgroundColor =
    new ColorUIResource (Color.white);
  UIDefaults defaults = UIManager.getDefaults();
  defaults.put ("Button.background", backgroundColor);

  // Create a vertical panel
  Box vert = Box.createVerticalBox();
  JButton top = new JButton(rightB);
  JButton bottom = new JButton(leftB);
  vert.add(top);
  vert.add(Box.createGlue());
  vert.add(bottom);

  // Create a horizontal panel
  Box horiz = Box.createHorizontalBox();
  JButton left = new JButton(rightB);
  JButton middle = new JButton(leftB);
  JButton right = new JButton(animB);

  horiz.add(left);
  horiz.add(Box.createGlue());
```

```
    horiz.add(middle);
    horiz.add(Box.createGlue());
    horiz.add(right);
    horiz.add(Box.createHorizontalStrut(XSPACE));

    Container content = getContentPane();
    content.add (vert, BorderLayout.WEST);
    content.add (horiz, BorderLayout.CENTER);
  }

  public static void main(String s[]) {
    BLayout frame = new BLayout("Boxing");
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {System.exit(0);}
    });

    frame.setSize(WIDTH, HEIGHT);
    frame.setVisible(true);
  }
}
```



2.7 SUMMARY

Applets are client side components(compiled java bytecodes) that are displayed in side browsers. AWT provides set of clsses to create java graphical user interfaces. JFC is a set of APls for building the GUI-related components of Java applets and applications. Swing extends AWT by supplying many more types of GUI components, providing 100% pure Java implementations of these components. Swing PL&F architecture makes it easy to customize both the appearance and the behavior of any Swing control.

The JFrame class is an extension to the AWT Frame class. The JToggleButton class provides a two-state button. A two-state button is one that doesn't spring back when released This class is used to create a mutually exclusive set of buttons. i.e. creating a set of buttons within the same ButtonGroup object such that only one of those buttons be allowed to be "on" at a time. The Icon interface defines the methods that define an icon. The JLabel class extends the JComponent. Also different layout managers are used to effectively display the different swing components.

2.7 SELF TEST

1. To have a TextArea display scrollbars, you need to place it within a ScrollPane?
   a) True
   b) False

2. Which of the following is Applet not a subclass of?
   a) Object
   b) Component
   c) Container
   d) Panel

e) Window

3. The CheckboxGroup is a subclass of Component.

a) True

b) False

4. Which of the following can contain a MenuBar?

a) Applet

b) Frame

c) Panel

d) Window

5. Using new TextField(10) will create a TextField that only accepts 10 characters of input.

a) True

b) False

6. Which APIs comprise the Java Foundation Classes (JFC)?

a) JNI, JavaBeans, and Swing Components

b) Collections, extensions, and Swing Components

c) Java 2D, Drag-and-Drop, Accessibility, and Swing Components

d) Drag and Drop, Collections, Swing Components, and Accessibility

7. Lightweight components depend on the local windowing toolkit.

a) True

b) False

8. JApplet, JDialog, JFrame, and JWindow are examples of heavyweight components.

a) True

b) False

9. Can a container have multiple layout managers?

a) True

b) False

10. What class controls the order in which components are drawn in their top-level container?

a) Jcontainer

b) JrootPane

c) JlayeredPane

d) JFrame

Answers

1. b)
2. e)
3. b)
4. b)
5. b)
6. c)
7. b)
8. a)
9. b)
10. c)

2.9 Questions

1. Explain the life cycle methods of an applet with an example program

2. Write a simple AWT GUI program to display the dialog box.
3. Write a GUI program to display textbox, two radio buttons, one checkbox and and a submit button(using swing components).
4. Write simple GUI program to display a menu.
5. What is JFC? Explain the different packages of JFC.

UNIT 3

NETWORKING, RMI, JDBC, SERVLETS

3.1 INTRODUCTION

Programmers want to deal with higher-level abstractions that are easier to understand. Java programmers want objects that they can interact with via an intuitive interface, using the Java constructs with which they are familiar. Java networking API provides java objects for network programming.Remote Method Invocation (RMI) technology, first introduced in JDK 1.1, elevates network programming to a higher plane. Although RMI is relatively easy to use, it is a remarkably powerful technology and exposes the average Java developer to an entirely new paradigm--the world of distributed object computing.

JDBC is a Java API that documents a standard framework for dealing with tabular and, generally, relational data. JDBC 2.0 begins a move to make SQL semi-transparent to the programmer. JDBC provides a framework for connecting to databases from Java applications. Servlets are pieces of Java source code that add functionality to a web server in a manner similar to the way applets add functionality to a browser. Servlets are designed to support a request/response computing model that is commonly used in web servers.

3.2 OBJECTIVES

In this unit you will learn about

- What are sockets?
- Types of Sockets
- Java network Programming
- What is RMI?
- How develop an RMI system?
- What is JDBC?
- Programming with JDBC
- What are sevlets?

- Servlet Programming

3.3 NETWORK PROGRAMMING WITH JAVA

Identifying a machine using InetAddress Class

Since java works within the Internet, it requires a way to uniquely identify a machine from all the others in the world. This is accomplished with the IP address that can exist in two forms:

1. The DNS (Domain Name Service) form. For example http://www.yahoo.com
2. Dotted Decimal form, which is 4 numbers separated by dots, such as 123.255.28.120.

In both cases, the IP address is represented internally as a 32-bit number (so each of the quad numbers cannot exceed 255), and you can get a special Java object to represent this number from either of the forms above by using the static InetAddress.getByName() method that's in java.net. The result is an object of type InetAddress that you can use to build a "socket".

As a simple example of using InetAddress.getByName(), consider what happens if you have a dial-up Internet service provider (ISP). Each time you dial up, you are assigned a temporary IP address. But while you're connected, your IP address has the same validity as any other IP address on the Internet. If someone connects to your machine using your IP address then they can connect to a web server or FTP server that you have running on your machine. Of course, they need to know your IP address, and since it's assigned each time you dial up, how can you find out what it is?

The following program uses InetAddress.getByName() to produce your IP address. To use it, you must know the name of your computer.

/** Finds out your network address when you're connected to the Internet. **/

```
import java.net.*;
public class WhoAmI {
public static void main(String[] args)
throws Exception {
if(args.length != 1) {
System.err.println("Usage: WhoAmI MachineName");
System.exit(1);
}
InetAddress a = InetAddress.getByName(args[0]);
System.out.println(a);
}
}
```

So, once you have connected to your ISP run the program:

>java WhoAmI computer name

You will get back a message like this (of course, the address is different each time):

Computer Name/IP address

Socket basics

Computers operate and communicate with one another in a very simple way. Computer chips are a collection of on-off switches that store and transmit data in the form of 1s and 0s. When computers want to share data, all they need to do is stream a few million of these bits and bytes back and forth, while agreeing on speed, sequence, timing, and such. How would you like to worry about those details every time you wanted to communicate information between two applications?

To avoid that, we need a set of packaged protocols that can do the job the same way every time. That would allow us to handle our application-level work without having to

worry about the low-level networking details. These sets of packaged protocols are called protocol stacks. The most common stack is TCP/IP.

Sockets reside roughly at the Session Layer of the OSI model. The Session Layer is sandwiched between the application-oriented upper layers and the real-time data communication lower layers. The Session Layer provides services for managing and controlling data flow between two computers. As part of this layer, sockets provide an abstraction that hides the complexities of getting the bits and bytes on the wire for transmission. In other words, sockets allow us to transmit data by having our application indicate that it wants to send some bytes. Sockets mask the nuts and bolts of getting the job done.

When you write code that uses sockets, that code does work at the Presentation Layer. The Presentation Layer provides a common representation of information that the Application Layer can use. Say you are planning to connect your application to a legacy banking system that understands only EBCDIC. Your application domain objects store information in ASCII format. In this case, you are responsible for writing code at the Presentation Layer to convert data from EBCDIC to ASCII, and then (for example) to provide a domain object to your Application Layer. Your Application Layer can then do whatever it wants with the domain object. The socket-handling code you write lives only at the Presentation Layer. Your Application Layer doesn't have to know anything about how sockets work.

What are Sockets?

The socket is the software abstraction used to represent the "terminals" of a connection between two machines. For a given connection, there's a socket on each machine, and you can imagine a hypothetical "cable" running between the two machines with each end of the "cable" plugged into a socket. Of course, the physical hardware and cabling between machines is completely unknown. The whole point of the abstraction is that we don't have to know more than is necessary.

In a nutshell, a socket on one computer that talks to a socket on another computer creates a communication channel. A programmer can use that channel to send data between the two machines. When you send data, each layer of the TCP/IP stack adds appropriate header information to wrap your data. These headers help the stack get your data to its destination. The Java language hides all of this from you by providing the data to your code on streams, which is why they are sometimes called streaming sockets. The Java platform gives us some simple yet powerful higher-level abstractions that make creating and using sockets easy.

Types of Sockets

Generally speaking, sockets come in two flavors in the java language:

- TCP sockets (implemented by the Socket class)
- UDP sockets (implemented by the DatagramSocket class)

TCP and UDP play the same role, but they do it differently. Both receive transport protocol packets and pass along their contents to the Presentation Layer. TCP divides messages into packets (datagrams) and reassembles them in the correct sequence at the receiving end. It also handles retransmission of missing packets. With TCP, the upper-level layers have much less to worry about. UDP doesn't provide these assembly and retransmission requesting features. It simply passes packets along. The upper layers have to make sure that the message is complete and assembled in correct sequence.

Working with URL Connection class

The Java platform provides implementations of sockets in the java.net package. The three important classes in java.net are:

- URLConnection
- Socket
- ServerSocket

The URLConnection class is the abstract super class of all classes that create a communications link between an application and a URL. URLConnections are most useful for getting documents on web servers, but can be used to connect to any resource identified by a URL. Concrete subclasses of URLConnection (such as HttpURLConnection) provide extra features specific to their implementation. For our example, we'll make use of the default behaviors provided by URLConnection itself.

Connecting to a URL involves several steps:

- Create the URLConnection
- Configure it using various setter methods
- Connect to the URL
- Interact with it using various getter methods

We'll look at some sample code that demonstrates how to use a URLConnection to request a document from a server. We'll begin with the structure for the URLClient class.

```
import java.io.*;
import java.net.*;

public class URLClient {
   protected URLConnection connection;
   public static void main(String[] args) {
   }
   public String getDocumentAt(String urlString) {
   }
}
```

The first order of business is to import java.net and java.io. We give our class one instance variable to hold a URLConnection. Our class has a main() method that handles the logic flow of surfing for a document. Our class also has a getDocumentAt() method that connects to the server and asks it for the given document. We will go into the details of each of these methods next.

The main() method handles the logic flow of surfing for a document:

```
public static void main(String[] args) {
   URLClient client = new URLClient();
   String yahoo = client.getDocumentAt("http://www.yahoo.com");
   System.out.println(yahoo);
}
```

The main() method simply creates a new URLClient and calls getDocumentAt() with a valid URL String. When that call returns the document, we store it in a String and then print it out to the console. The real work, though, gets done in the getDocumentAt() method. The getDocumentAt() method handles the real work of getting a document over the Web:

```
public String getDocumentAt(String urlString) {
   StringBuffer document = new StringBuffer();
   try {
      URL url = new URL(urlString);
      URLConnection conn = url.openConnection();
      BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));
      String line = null;
      while ((line = reader.readLine()) != null)
            document.append(line + "\n");
      reader.close();
   } catch (MalformedURLException e) {
      System.out.println("Unable to connect to URL: " + urlString);
   } catch (IOException e) {
      System.out.println("IOException when connecting to URL: " + urlString);
   }
   return document.toString();
```

```
}
```

The getDocumentAt() method takes a String containing the URL of the document we want to get. We start by creating a StringBuffer to hold the lines of the document. Next, we create a new URL with the urlString we passed in. Then we create a URLConnection and open it:

```
URLConnection conn = url.openConnection();
```

Once we have a URLConnection, we get its InputStream and wrap it in an InputStreamReader, which we then wrap in a BufferedReader so that we can read lines of the document we're getting from the server.

```
BufferedReader reader = new BufferedReader(new  InputStreamReader(conn.getInputStream()));
```

Having BufferedReader makes reading the contents of the document easy. We call readLine() on reader in a while loop:

```
String line = null;
while ((line = reader.readLine()) != null)
   document.append(line + "\n");
```

The call to readLine() is going to block until it reaches a line termination character in the incoming bytes on the InputStream. If it doesn't get one, it will keep waiting. It will return null only when the connection is closed. In this case, once we get a line, we append it to the StringBuffer called document, along with a newline character. This preserves the format of the document that was read on the server side. When we're done reading lines, we close the BufferedReader:

```
reader.close();
```

If the urlString supplied to a URL constructor is invalid, a MalformedURLException is thrown. If something else goes wrong, such as when getting the InputStream on the connection, an IOException is thrown.

Beneath the covers, URLConnection uses a socket to read from the URL we specified (which just resolves to an IP address), but we don't have to know about it and we don't care. Before we move on, let's review the steps to create and use a URLConnection:

- Instantiate a URL with a valid URL String of the resource you're connecting to
- Open a connection on that URL.
- Wrap the InputStream for that connection in a BufferedReader so you can read lines.
- Read the document using your BufferedReader.
- Close your BufferedReader.

The complete code listing for URLClient is given below:

```
import java.io.*;
import java.net.*;

public class URLClient {
   protected HttpURLConnection connection;
   public String getDocumentAt(String urlString) {
      StringBuffer document = new StringBuffer();
      try {
         URL url = new URL(urlString);
         URLConnection conn = url.openConnection();
         BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream()));

         String line = null;
         while ((line = reader.readLine()) != null)
            document.append(line + "\n");

         reader.close();
      } catch (MalformedURLException e) {
         System.out.println("Unable to connect to URL: " + urlString);
      } catch (IOException e) {
         System.out.println("IOException when connecting to URL: " + urlString);
      }
```

```
        return document.toString();
    }
    public static void main(String[] args) {
        URLClient client = new URLClient();
        String yahoo = client.getDocumentAt("http://www.yahoo.com");
        System.out.println(yahoo);
    }
}
```

A Socket Client and Server

In this section, we illustrate how you can use Socket and ServerSocket in your Java code. The client uses a Socket to connect to a server. The server listens on port 3000 with a ServerSocket. The client requests the contents of a file on the server's C: drive. For the sake of clarity, we have split the example into client side and the server side. At the end, we'll put it all together so that you can see the entire picture. The client and the server can be run on a single machine.

Creating the RemoteFileClient class

Here is the structure for the RemoteFileClient class:

```
import java.io.*;
import java.net.*;

public class RemoteFileClient {
    protected String hostIp;
    protected int hostPort;
    protected BufferedReader socketReader;
    protected PrintWriter socketWriter;

    public RemoteFileClient(String aHostIp, int aHostPort) {
```

```
        hostIp = aHostIp;
        hostPort = aHostPort;
    }
    public static void main(String[] args) {
    }
    public void setUpConnection() {
    }
    public String getFile(String fileNameToGet) {
    }
    public void tearDownConnection() {
    }
}
```

First we import java.net and java.io. The java.net package gives you the socket tools you need. The java.io package gives you tools to read and write streams, which is the only way you can communicate with TCP sockets. The class instance variables are used to support reading from and writing to socket streams, and to store details of the remote host to which we will connect. The constructor for our class takes an IP address and a port number for a remote host and assigns them to instance variables. Our class has a main() method and three other methods. We'll go into the details of these methods later. For now, just know that setUpConnection() will connect to the remote server, getFile() will ask the remote server for the contents of fileNameToGet, and tearDownConnection() will disconnect from the remote server.

Implementing main()

Here we implement the main() method, which will create the RemoteFileClient, use it to get the contents of a remote file, and then print the result:

```
public static void main(String[] args) {
    RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
    remoteFileClient.setUpConnection();
String fileContents =
        remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
```

```
    remoteFileClient.tearDownConnection();
    System.out.println(fileContents);
}
```

The main() method instantiates a new RemoteFileClient (the client) with an IP address and port number for the host. Then, we tell the client to set up a connection to the host. Next, we tell the client to get the contents of a specified file on the host. Finally, we tell the client to tear down its connection to the host. We print out the contents of the file to the console.

Setting up a connection

Here we implement the setUpConnection() method, which will set up our Socket and give us access to its streams:

```
public void setUpConnection() {
    try {
        Socket client = new Socket(hostIp, hostPort);
        socketReader = new BufferedReader(
                                new InputStreamReader(client.getInputStream()));
        socketWriter = new PrintWriter(client.getOutputStream());
    } catch (UnknownHostException e) {
        System.out.println("Error setting up socket connection: unknown host at " + hostIp + ":" + hostPort);
    } catch (IOException e) {
      System.out.println("Error setting up socket connection: " + e);
    }
}
```

The setUpConnection() method creates a Socket with the IP address and port number of the host:

```
Socket client = new Socket(hostIp, hostPort);
```

We wrap the Socket's InputStream in a BufferedReader so that we can read lines from the stream. Then, we wrap the Socket's OutputStream in a PrintWriter so that we can send our request for a file to the server:

```
socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
socketWriter = new PrintWriter(client.getOutputStream());
```

Remember that our client and server simply pass bytes back and forth. Both the client and the server have to know what the other is going to be sending so that they can respond appropriately. In this case, the server knows that we'll be sending it a valid file path.

Talking to the host

Here we implement the getFile() method, which will tell the server what file we want and receive the contents from the server when it sends the contents back:

```
public String getFile(String fileNameToGet) {
   StringBuffer fileLines = new StringBuffer();
   try {
      socketWriter.println(fileNameToGet);
      socketWriter.flush();
      String line = null;
      while ((line = socketReader.readLine()) != null)
         fileLines.append(line + "\n");
   } catch (IOException e) {
     System.out.println("Error reading from file: " + fileNameToGet);
   }
   return fileLines.toString();
}
```

A call to the getFile() method requires a valid file path String. It starts by creating the StringBuffer called fileLines for storing each of the lines that we read from the file on the server:

```
StringBuffer fileLines = new StringBuffer();
```

In the try{}catch{} block, we send our request to the host using the PrintWriter that was established during connection setup:

```
socketWriter.println(fileNameToGet);
socketWriter.flush();
```

Note that we flush() the PrintWriter here instead of closing it. This forces data to be sent to the server without closing the Socket. Once we've written to the Socket, we are expecting some response. We have to wait for it on the Socket's InputStream, which we do by calling readLine() on our BufferedReader in a while loop. We append each returned line to the fileLines StringBuffer (with a newline character to preserve the lines):

```
String line = null;
while ((line = socketReader.readLine()) != null)
   fileLines.append(line + "\n");
```

Tearing down a connection

Here we implement the tearDownConnection() method, which will "clean up" after we're done using our connection:

```
public void tearDownConnection() {
   try {
      socketWriter.close();
      socketReader.close();
   } catch (IOException e) {
   System.out.println("Error tearing down socket connection: " + e);
   }
}
```

The tearDownConnection() method simply closes the BufferedReader and PrintWriter we created on our Socket's InputStream and OutputStream, respectively. Doing this closes the underlying streams that we acquired from the Socket, so we have to catch the possible IOException. Before we move on to the server end of things, let's review the steps to create and use a Socket:

- Instantiate a Socket with the IP address and port of the machine you're connecting to.
- Get the streams on that Socket for reading and writing.
- Wrap the streams in instances of BufferedReader/PrintWriter
- Read from and write to the Socket.
- Close your open streams.

The complete code listing for RemoteFileClient is given below:

```
import java.io.*;
import java.net.*;

public class RemoteFileClient {
   protected BufferedReader socketReader;
   protected PrintWriter socketWriter;
   protected String hostIp;
   protected int hostPort;

   public RemoteFileClient(String aHostIp, int aHostPort) {
      hostIp = aHostIp;
      hostPort = aHostPort;
   }
   public String getFile(String fileNameToGet) {
      StringBuffer fileLines = new StringBuffer();

      try {
         socketWriter.println(fileNameToGet);
         socketWriter.flush();
```

```
        String line = null;
        while ((line = socketReader.readLine()) != null)
            fileLines.append(line + "\n");
    } catch (IOException e) {
        System.out.println("Error reading from file: " + fileNameToGet);
    }

    return fileLines.toString();
}
public static void main(String[] args) {
    RemoteFileClient remoteFileClient = new RemoteFileClient("127.0.0.1", 3000);
    remoteFileClient.setUpConnection();
    String fileContents = remoteFileClient.getFile("C:\\WINNT\\Temp\\RemoteFile.txt");
    remoteFileClient.tearDownConnection();
    System.out.println(fileContents);
}
public void setUpConnection() {
    try {
        Socket client = new Socket(hostIp, hostPort);

        socketReader = new BufferedReader(new InputStreamReader(client.getInputStream()));
        socketWriter = new PrintWriter(client.getOutputStream());

    } catch (UnknownHostException e) {
        System.out.println("Error setting up socket connection: unknown host at " + hostIp + ":" + hostPort);
    } catch (IOException e) {
  System.out.println("Error setting up socket connection: " + e);
    }
}
public void tearDownConnection() {
    try {
        socketWriter.close();
```

```
            socketReader.close();
        } catch (IOException e) {
    System.out.println("Error tearing down socket connection: " + e);
        }
    }
}
```

Creating the RemoteFileServer class

Here is the structure for the RemoteFileServer class:

```
import java.io.*;
import java.net.*;
public class RemoteFileServer {
    protected int listenPort = 3000;
    public static void main(String[] args) {
    }
    public void acceptConnections() {
    }
    public void handleConnection(Socket incomingConnection) {
    }
}
```

As with the client, we first import java.net and java.io. Next, we give our class an instance variable to hold the port to listen to for incoming connections. By default, this is port 3000. Our class has a main() method and two other methods. We'll go into the details of these methods later. For now, just know that acceptConnections() will allow clients to connect to the server, and handleConnection() interacts with the client Socket to send the contents of the requested file to the client.

Implementing main()

Here we implement the main() method, which will create a RemoteFileServer and tell it to accept connections:

```
public static void main(String[] args) {
   RemoteFileServer server = new RemoteFileServer();
   server.acceptConnections();
}
```

The main() method on the server side is even simpler than on the client side. We instantiate a new RemoteFileServer, which will listen for incoming connection requests on the default listen port. Then we call acceptConnections() to tell the server to listen

Accepting connections

Here we implement the acceptConnections() method, which will set up a ServerSocket and wait for connection requests:

```
public void acceptConnections() {
   try {
      ServerSocket server = new ServerSocket(listenPort);
      Socket incomingConnection = null;
      while (true) {
         incomingConnection = server.accept();
         handleConnection(incomingConnection);
      }
   } catch (BindException e) {
      System.out.println("Unable to bind to port " + listenPort);
   } catch (IOException e) {
      System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
   }
}
```

The acceptConnections() method creates a ServerSocket with the port number to listen to. We then tell the ServerSocket to start listening by calling accept() on it. The accept() method blocks until a connection request comes in. At that point, accept() returns a new Socket bound to a randomly assigned port on the server, which is passed to handleConnection(). Notice that this accepting of connections is in an infinite loop.

Whenever you create a ServerSocket, Java code may throw an error if it can't bind to the specified port. So we have to catch the possible BindException here. And just like on the client side, we have to catch an IOException that could be thrown when we try to accept connections on our ServerSocket. Note that you can set a timeout on the accept() call by calling setSoTimeout() with number of milliseconds to avoid a really long wait. Calling setSoTimeout() will cause accept() to throw an IOException after the specified elapsed time.

Handling connections

Here we implement the handleConnection() method, which will use streams on a connection to receive input and write output:

```
public void handleConnection(Socket incomingConnection) {
  try {
    OutputStream outputToSocket = incomingConnection.getOutputStream();
    InputStream inputFromSocket = incomingConnection.getInputStream();
    BufferedReader streamReader =
      new BufferedReader(new InputStreamReader(inputFromSocket));

    FileReader fileReader = new FileReader(new File(streamReader.readLine()));
    BufferedReader bufferedFileReader = new BufferedReader(fileReader);
    PrintWriter streamWriter =
      new PrintWriter(incomingConnection.getOutputStream());
    String line = null;
    while ((line = bufferedFileReader.readLine()) != null) {
      streamWriter.println(line);
```

```
        }
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

As with the client, we get the streams associated with the Socket we just made, using getOutputStream() and getInputStream(). As on the client side, we wrap the InputStream in a BufferedReader and the OutputStream in a PrintWriter. On the server side, we need to add some code to read the target file and send the contents to the client line by line. Here's the important code:

```
 FileReader fileReader = new FileReader(new File(streamReader.readLine()));
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
String line = null;
while ((line = bufferedFileReader.readLine()) != null) {
    streamWriter.println(line);
}
```

This code needs some detailed explanation. Let's look at it bit by bit:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
```

First, we make use of our BufferedReader on the Socket's InputStream. We should be getting a valid file path, so we construct a new File using that path name. We make a new FileReader to handle reading the file.

```
BufferedReader bufferedFileReader = new BufferedReader(fileReader);
```

Here we wrap our FileReader in a BufferedReader to let us read the file line by line. Next, we call readLine() on our BufferedReader. This call will block until bytes come in. When we get

some bytes, we put them in our local line variable, and then write them out to the client. When we're done reading and writing, we close the open streams. Note that we closed streamWriter and streamReader after we were done reading from the Socket. You might ask why we didn't close streamReader immediately after reading in the file name. The reason is that when you do that, your client won't get any data. If you close the streamReader before you close streamWriter, you can write to the Socket all you want but no data will make it across the channel (it's closed).

Let's review the steps to create and use a ServerSocket:

- Instantiate a ServerSocket with a port on which you want it to listen for incoming client connections.
- Call accept() on the ServerSocket to block while waiting for connection.
- Get the streams on that underlying Socket for reading and writing.
- Wrap the streams as necessary to simplify your life.
- Read from and write to the Socket.
- Close your open streams (never close your Reader before your Writer).

The complete code listing for RemoteFileServer is given below:

```
import java.io.*;
import java.net.*;

public class RemoteFileServer {
   int listenPort;
   public RemoteFileServer(int aListenPort) {
      listenPort = aListenPort;
   }
   public void acceptConnections() {
      try {
         ServerSocket server = new ServerSocket(listenPort);
         Socket incomingConnection = null;
         while (true) {
            incomingConnection = server.accept();
```

```
            handleConnection(incomingConnection);
        }
    } catch (BindException e) {
        System.out.println("Unable to bind to port " + listenPort);
    } catch (IOException e) {
        System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
    }
}
public void handleConnection(Socket incomingConnection) {
    try {
        OutputStream outputToSocket = incomingConnection.getOutputStream();
        InputStream inputFromSocket = incomingConnection.getInputStream();

        BufferedReader streamReader = new BufferedReader(new InputStreamReader(inputFromSocket));

        FileReader fileReader = new FileReader(new File(streamReader.readLine()));

        BufferedReader bufferedFileReader = new BufferedReader(fileReader);
        PrintWriter streamWriter = new PrintWriter(incomingConnection.getOutputStream());
        String line = null;
        while ((line = bufferedFileReader.readLine()) != null) {
            streamWriter.println(line);
        }
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
public static void main(String[] args) {
    RemoteFileServer server = new RemoteFileServer(3000);
    server.acceptConnections();
```

```
   }
}
```

A multithreaded Server

The previous example gives you the basics of how to handle only one client at a time. The handleConnection() is a blocking method. Only when it has completed its dealings with the current connection can the server accept another client. Most of the time, you will want (and need) a multithreaded server.

There aren't too many changes you need to make to RemoteFileServer to begin handling multiple clients simultaneously. As a matter of fact, had we discussed backlogging earlier, we would have just one method to change, although we'll need to create something new to handle the incoming connections. We will show you here also how ServerSocket handles lots of clients waiting to use the server.

Here we implement the revised acceptConnections() method, which will create a ServerSocket that can handle a backlog of requests, and tell it to accept connections:

```
public void acceptConnections() {
      try {
      ServerSocket server = new ServerSocket(listenPort, 5);
      Socket incomingConnection = null;
      while (true) {
         incomingConnection = server.accept();
         handleConnection(incomingConnection);
      }
   } catch (BindException e) {
   System.out.println("Unable to bind to port " + listenPort);
   } catch (IOException e) {
   System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
   }
}
```

Our new server still needs to acceptConnections() so this code is virtually identical. The highlighted line indicates the one significant difference. For this multithreaded version, we now specify the maximum number of client requests that can backlog when instantiating the ServerSocket. If we don't specify the max number of client requests, the default value of 50 is assumed.

Here's how it works. Suppose we specify a backlog of 5 and that five clients request connections to our server. Our server will start processing the first connection, but it takes a long time to process that connection. Since our backlog is 5, we can have up to five requests in the queue at one time. We're processing one, so that means we can have five others waiting. That's a total of six either waiting or being processed. If a seventh client asks for a connection while our server is still busy accepting connection one (remember that 2 - 6 are still in queue), that seventh client will be refused. We will illustrate limiting the number of clients that can be connected simultaneously in our pooled server example.

Handling connections: Part 1

Here we'll talk about the structure of the handleConnection() method, which spawns a new Thread to handle each connection. We'll discuss this in two parts. We'll focus on the method itself in this panel, and then examine the structure of the ConnectionHandler helper class used by this method in the next panel.

```
public void handleConnection(Socket connectionToHandle) {
    new Thread(new ConnectionHandler(connectionToHandle)).start();
}
```

This method represents the big change to our RemoteFileServer. We still call handleConnection() after the server accepts a connection, but now we pass that Socket to an instance of ConnectionHandler, which is Runnable. We create a new Thread with our ConnectionHandler and start it up. The ConnectionHandler's run() method contains the Socket reading/writing and File reading code that used to be in handleConnection() on RemoteFileServer.

Handling connections: Part 2

Here is the structure for the ConnectionHandler class:

```
import java.io.*;
import java.net.*;

public class ConnectionHandler implements Runnable{
  Socket socketToHandle;
  public ConnectionHandler(Socket aSocketToHandle) {
    socketToHandle = aSocketToHandle;
  }
  public void run() {
  }
}
```

This helper class is pretty simple. As with our other classes so far, we import java.net and java.io. The class has a single instance variable, socketToHandle, that holds the Socket handled by the instance.

The constructor for our class takes a Socket instance and assigns it to socketToHandle. Notice that the class implements the Runnable interface. Classes that implement this interface must implement the run() method, which our class does. We'll go into the details of run() later. For now, just know that it will actually process the connection using code identical to what we saw before in our RemoteFileServer class.

Implementing run()

Here we implement the run() method, which will grab the streams on our connection, use them to read from and write to the connection, and close them when we are done:

```
public void run() {
    try {
        PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream());
        BufferedReader streamReader =
            new BufferedReader(new InputStreamReader(socketToHandle.getInputStream()));
        String fileToRead = streamReader.readLine();
        BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));
        String line = null;
        while ((line = fileReader.readLine()) != null)
            streamWriter.println(line);
        fileReader.close();
        streamWriter.close();
        streamReader.close();
    } catch (Exception e) {
        System.out.println("Error handling a client: " + e);
    }
}
```

The run() method on ConnectionHandler does what handleConnection() on RemoteFileServer did. First, we wrap the InputStream and OutputStream in a BufferedReader and a PrintWriter, respectively (using getOutputStream() and getInputStream() on the Socket). Then we read the target file line by line with this code:

```
FileReader fileReader = new FileReader(new File(streamReader.readLine()));
    BufferedReader bufferedFileReader = new BufferedReader(fileReader);
    String line = null;
    while ((line = bufferedFileReader.readLine()) != null) {
        streamWriter.println(line);
    }
```

Remember that we should be getting a valid file path from the client, so we construct a new File using that path name, wrap it in a FileReader to handle reading the file, and then wrap that in a BufferedReader to let us read the file line by line. We call readLine() on our

BufferedReader in a while loop until we have no more lines to read. Remember that the call to readLine() will block until bytes come in. When we get some bytes, we put them in our local line variable, and then write them out to the client. When we're done reading and writing, we close the open streams.

Let's review the steps to create and use a multithreaded version of the server:

- Modify acceptConnections() to instantiate a ServerSocket with a default 50-connection backlog (or whatever specific number you want, greater than 1).
- Modify handleConnection() on the ServerSocket to spawn a new Thread with an instance of ConnectionHandler.
- Implement the ConnectionHandler class, borrowing code from the handleConnection() method on RemoteFileServer.

.

The complete code for the multithreaded server is given below:

```
import java.io.*;
import java.net.*;

public class MultithreadedRemoteFileServer {
   protected int listenPort;
   public MultithreadedRemoteFileServer(int aListenPort) {
      listenPort = aListenPort;
   }
   public void acceptConnections() {
      try {
         ServerSocket server = new ServerSocket(listenPort, 5);
         Socket incomingConnection = null;
         while (true) {
            incomingConnection = server.accept();
            handleConnection(incomingConnection);
         }
      } catch (BindException e) {
         System.out.println("Unable to bind to port " + listenPort);
      } catch (IOException e) {
```

```
            System.out.println("Unable to instantiate a ServerSocket on port: " + listenPort);
        }
    }
    public void handleConnection(Socket connectionToHandle) {
        new Thread(new ConnectionHandler(connectionToHandle)).start();
    }
    public static void main(String[] args) {
        MultithreadedRemoteFileServer server = new MultithreadedRemoteFileServer(3000);
        server.acceptConnections();
    }
}
```

The ConnectionHandler class code is as below:

```
import java.io.*;
import java.net.*;

public class ConnectionHandler implements Runnable {
    protected Socket socketToHandle;
    public ConnectionHandler(Socket aSocketToHandle) {
        socketToHandle = aSocketToHandle;
    }
    public void run() {
        try {
            PrintWriter streamWriter = new PrintWriter(socketToHandle.getOutputStream());
            BufferedReader streamReader = new BufferedReader(new InputStreamReader(socketToHandle.getInputStream()));

            String fileToRead = streamReader.readLine();
            BufferedReader fileReader = new BufferedReader(new FileReader(fileToRead));

            String line = null;
            while ((line = fileReader.readLine()) != null)
                streamWriter.println(line);

            fileReader.close();
            streamWriter.close();
            streamReader.close();
```

```
      } catch (Exception e) {
         System.out.println("Error handling a client: " + e);
      }
   }
}
```

Datagrams

The examples you've seen so far use the Transmission Control Protocol (TCP, also known as stream-based sockets), which is designed for ultimate reliability and guarantees that the data will get there. It allows retransmission of lost data, it provides multiple paths through different routers in case one goes down, and bytes are delivered in the order they are sent.

The support for datagrams in Java has the same feel as its support for TCP sockets, but there are significant differences. With datagrams, you put a **DatagramSocket** on both the client and server, but there is no analogy to the **ServerSocket** that waits around for a connection. That's because there is no "connection," but instead a datagram just shows up. Another fundamental difference is that with TCP sockets, once you've made the connection you don't need to worry about who's talking to whom anymore; you just send the data back and forth through conventional streams. However, with datagrams, the datagram packet must know where it came from and where it's supposed to go. That means you must know these things for each datagram packet that you load up and ship off.

A **DatagramSocket** sends and receives the packets, and the **DatagramPacket** contains the information. When you're receiving a datagram, you need only provide a buffer in which the data will be placed; the information about the Internet address and port number where the information came from will be automatically initialized when the packet arrives through the **DatagramSocket**. So the constructor for a **DatagramPacket** to receive datagrams is:

```
DatagramPacket(buf, buf.length)
```

in which **buf** is an array of **byte**. Since **buf** is an array, you might wonder why the constructor couldn't figure out the length of the array on its own. Actually, it's a throwback to C-style programming, in which of course arrays can't tell you how big they are.

You can reuse a receiving datagram; you don't have to make a new one each time. Every time you reuse it, the data in the buffer is overwritten. The maximum size of the buffer is restricted only by the allowable datagram packet size, which limits it to slightly less than 64Kbytes. However, in many applications you'll want it to be much smaller, certainly when you're sending data. Your chosen packet size depends on what you need for your particular application. When you send a datagram, the **DatagramPacket** must contain not only the data, but also the Internet address and port where it will be sent. So the constructor for an outgoing **DatagramPacket** is:

DatagramPacket(buf, length, inetAddress, port)

This time, **buf** (which is a **byte** array) already contains the data that you want to send out. The **length** might be the length of **buf**, but it can also be shorter, indicating that you want to send only that many bytes. The other two arguments are the Internet address where the packet is going and the destination port within that machine.2. Multiple clients will send datagrams to a server, which will echo them back to the same client that sent the message. To simplify the creation of a **DatagramPacket** from a **String** and vice-versa, the example begins with a utility class, **Dgram**, to do the work for you:

```
//: Dgram.java
// A utility class to convert back and forth between Strings and          //DataGramPackets.
import java.net.*;
public class Dgram {
public static DatagramPacket toDatagram(
String s, InetAddress destIA, int destPort) {
// Deprecated in Java 1.1, but it works:
byte[] buf = new byte[s.length() + 1];
s.getBytes(0, s.length(), buf, 0);
```

```
// The correct Java 1.1 approach, but it's
// Broken (it truncates the String):
// byte[] buf = s.getBytes();
return new DatagramPacket(buf, buf.length,
destIA, destPort);
}
public static String toString(DatagramPacket p){
// The Java 1.0 approach:
// return new String(p.getData(),
// 0, 0, p.getLength());
// The Java 1.1 approach:
return
new String(p.getData(), 0, p.getLength());
}
}
```

The first method of **Dgram** takes a **String**, an **InetAddress,** and a port number and builds a **DatagramPacket** by copying the contents of the **String** into a **byte** buffer and passing the buffer into the **DatagramPacket** constructor. Notice the "+1" in the buffer allocation – this was necessary to prevent truncation. The **getBytes( )** method of **String** is a special operation that copies the **char**s of a **String** into a **byte** buffer. This method is now deprecated; Java 1.1 has a "better" way to do this but it's commented out here because it truncates the **String**. So you'll get a deprecation message when you compile it under Java 1.1, but the behavior will be correct.

The **Dgram.toString( )** method shows both the Java 1.0 approach and the Java 1.1 approach (which is different because there's a new kind of **String** constructor).

Here is the server for the datagram demonstration:

```
//: ChatterServer.java
// A server that echoes datagrams
import java.net.*;
```

```
import java.io.*;
import java.util.*;

public class ChatterServer {
static final int INPORT = 1711;
private byte[] buf = new byte[1000];
private DatagramPacket dp =
new DatagramPacket(buf, buf.length);
// Can listen & send on the same socket:
private DatagramSocket socket;
public ChatterServer() {
try {
socket = new DatagramSocket(INPORT);
System.out.println("Server started");
while(true) {
// Block until a datagram appears:
socket.receive(dp);
String rcvd = Dgram.toString(dp) +
", from address: " + dp.getAddress() +
", port: " + dp.getPort();
System.out.println(rcvd);
String echoString =
"Echoed: " + rcvd;
// Extract the address and port from the
// received datagram to find out where to
// send it back:
DatagramPacket echo =
Dgram.toDatagram(echoString,
dp.getAddress(), dp.getPort());
socket.send(echo);
}
} catch(SocketException e) {
System.err.println("Can't open socket");
```

```
System.exit(1);
} catch(IOException e) {
System.err.println("Communication error");

e.printStackTrace();
}
}
public static void main(String[] args) {
new ChatterServer();
}
}
```

The **ChatterServer** contains a single **DatagramSocket** for receiving messages, instead of creating one each time you're ready to receive a new message. The single **DatagramSocket** can be used repeatedly. This **DatagramSocket** has a port number because this is the server and the client must have an exact address where it wants to send the datagram. It is given a port number but not an Internet address because it resides on "this" machine so it knows what its Internet address is (in this case, the default **localhost**). In the infinite **while** loop, the **socket** is told to **receive( )**, whereupon it blocks until a datagram shows up, and then sticks it into our designated receiver, the **DatagramPacket dp**. The packet is converted to a **String** along with information about the Internet address and socket where the packet came from. This information is displayed, and then an extra string is added to indicate that it is being echoed back from the server.

Now there's a bit of a quandary. As you will see, there are potentially many different Internet addresses and port numbers that the messages might come from – that is, the clients can reside on any machine. To send a message back to the client that originated it, you need to know that client's Internet address and port number. Fortunately, this information is conveniently packaged inside the **DatagramPacket** that sent the message, so all you have to do is pull it out using **getAddress( )** and **getPort( )**, which are used to build the **DatagramPacket echo** that is sent back through the same socket that's doing the receiving.

In addition, when the socket sends the datagram, it automatically adds the Internet address and port information of this machine, so that when the client receives the message, it can use **getAddress( )** and **getPort( )** to find out where the datagram came from. In fact, the only time that **getAddress( )** and **getPort( )** don't tell you where the datagram came from is if you create a datagram to send and you call **getAddress( )** and **getPort( )** before you send the datagram (in which case it tells the address and port of this machine, the one the datagram is being sent from). This is an essential part of datagrams: you don't need to keep track of where a message came from because it's always stored inside the datagram. In fact, the most reliable way to program is if you don't try to keep track, but instead always extract the address and port from the datagram in question (as is done here).

To test this server, here's a program that makes a number of clients, all of which fire datagram packets to the server and wait for the server to echo them back.

```
//: ChatterClient.java
// Tests the ChatterServer by starting multiple
// clients, each of which sends datagrams.
import java.lang.Thread;
import java.net.*;
import java.io.*;
public class ChatterClient extends Thread {
// Can listen & send on the same socket:
private DatagramSocket s;
private InetAddress hostAddress;
private byte[] buf = new byte[1000];

private DatagramPacket dp =
new DatagramPacket(buf, buf.length);
private int id;
public ChatterClient(int identifier) {
id = identifier;
try {
// Auto-assign port number:
```

```
s = new DatagramSocket();
hostAddress =
InetAddress.getByName("localhost");
} catch(UnknownHostException e) {
System.err.println("Cannot find host");
System.exit(1);
} catch(SocketException e) {
System.err.println("Can't open socket");
e.printStackTrace();
System.exit(1);
}
System.out.println("ChatterClient starting");
}
public void run() {
try {
for(int i = 0; i < 25; i++) {
String outMessage = "Client #" +
id + ", message #" + i;
// Make and send a datagram:
s.send(Dgram.toDatagram(outMessage,
hostAddress,
ChatterServer.INPORT));
// Block until it echoes back:
s.receive(dp);
// Print out the echoed contents:
String rcvd = "Client #" + id +
", rcvd from " +
dp.getAddress() + ", " +
dp.getPort() + ": " +
Dgram.toString(dp);
System.out.println(rcvd);
}
} catch(IOException e) {
```

```
e.printStackTrace();
System.exit(1);
}
}
public static void main(String[] args) {
for(int i = 0; i < 10; i++)
new ChatterClient(i).start();
}
}
```

ChatterClient is created as a **Thread** so that multiple clients can be made to bother the server. Here you can see that the receiving **DatagramPacket** looks just like the one used for **ChatterServer**. In the constructor, the **DatagramSocket** is created with no arguments since it doesn't need to advertise itself as being at a particular port number. The Internet address used for this socket will be "this machine" (for the example, **localhost**) and the port number will be automatically assigned, as you will see from the output. This **DatagramSocket**, like the one for the server, will be used both for sending and receiving. The **hostAddress** is the Internet address of the host machine you want to talk to. The one part of the program in which you must know an exact Internet address and port number is the part in which you make the outgoing **DatagramPacket**. As is always the case, the host must be at a known address and port number so that clients can originate conversations with the host.

Each thread is given a unique identification number (although the port number automatically assigned to the thread would also provide a unique identifier). In **run( )**, a message **String** is created that contains the thread's identification number and the message number this thread is currently sending. This **String** is used to create a datagram that is sent to the host at its address; the port number is taken directly from a constant in **ChatterServer**. Once the message is sent, **receive( )** blocks until the server replies with an echoing message. All of the information that's shipped around with the message allows you to see that what comes back to this particular thread is derived from the message that originated from it. In

this example, even though UDP is an “unreliable” protocol, you’ll see that all of the datagrams get where they’re supposed to.

When you run this program, you’ll see that each of the threads finishes, which means that each of the datagram packets sent to the server is turned around and echoed to the correct recipient; otherwise one or more threads would hang, blocking until their input shows up. You might think that the only right way to, for example, transfer a file from one machine to another is through TCP sockets, since they’re “reliable.” However, because of the speed of datagrams they can actually be a better solution. You simply break the file up into packets and number each packet. The receiving machine takes the packets and reassembles them; a “header packet” tells the machine how many to expect and any other important information. If a packet is lost, the receiving machine sends a datagram back telling the sender to retransmit.

3.4 REMOTE METHOD INVOCATION

What is RMI?

Remote Method Invocation (RMI) is a way of communication between two objects. RMI's purpose is to make objects in separate virtual machines look and act like local objects. The virtual machine that calls the remote object is sometimes referred to as a client. Similarly, we refer to the VM that contains the remote as a server.

A primary goal for the RMI designers was to allow programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. To do this, they had to carefully map how Java classes and objects work in a single Java Virtual Machine (JVM) to a new model of how classes and objects would work in a distributed (multiple JVM) computing environment.

This section introduces the RMI architecture from the perspective of the distributed or remote Java objects, and explores their differences through the behavior of local Java objects. The RMI architecture defines how objects behave, how and when exceptions can occur, how memory is managed, and how parameters are passed to, and returned from, remote methods

Comparison of Distributed and Nondistributed Java Programs

The RMI architects tried to make the use of distributed Java objects similar to using local Java objects. While they succeeded, some important differences are listed in the table below. As of now do not worry if you do not understand all of the difference. They will become clear as you explore the RMI architecture.

	Local Object	**Remote Object**
Object Definition	A local object is defined by a Java class.	A remote object's exported behavior is defined by an interface that must extend the Remote interface.
Object Implementation	A local object is implemented by its Java class.	A remote object's behavior is executed by a Java class that implements the remote interface.
Object Creation	A new instance of a local object is created by the new operator.	A new instance of a remote object is created on the host computer with the new operator. A client cannot directly create a new remote object (unless using Java 2 Remote Object Activation).
Object Access	A local object is accessed directly via an object reference variable.	A remote object is accessed via an object reference variable which points to a proxy stub implementation of the remote interface.
References	In a single JVM, an object	A "remote reference" is a pointer to a proxy

	reference points directly at an object in the heap.	object (a "stub") in the local heap. That stub contains information that allows it to connect to a remote object, which contains the implementation of the methods.
Active References	In a single JVM, an object is considered "alive" if there is at least one reference to it.	In a distributed environment, remote JVMs may crash, and network connections may be lost. A remote object is considered to have an active remote reference to it if it has been accessed within a certain time period (the lease period). If all remote references have been explicitly dropped, or if all remote references have expired leases, then a remote object is available for distributed garbage collection.
Finalization	If an object implements the finalize() method, it is called before an object is reclaimed by the garbage collector.	If a remote object implements the Unreferenced interface, the unreferenced method of that interface is called when all remote references have been dropped.
Garbage Collection	When all local references to an object have been dropped, an object becomes a candidate for garbage collection.	The distributed garbage collector works with the local garbage collector. If there are no remote references and all local references to a remote object have been dropped, then it becomes a candidate for garbage collection through the normal means.
Exceptions	Exceptions are either Runtime exceptions or Exceptions. The Java compiler forces a program to handle all Exceptions.	RMI forces programs to deal with any possible RemoteException objects that may be thrown. This was done to ensure the robustness of distributed applications.

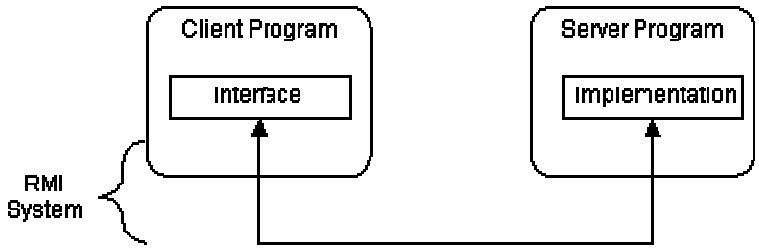
Java RMI Architecture

Interfaces: The Heart of RMI

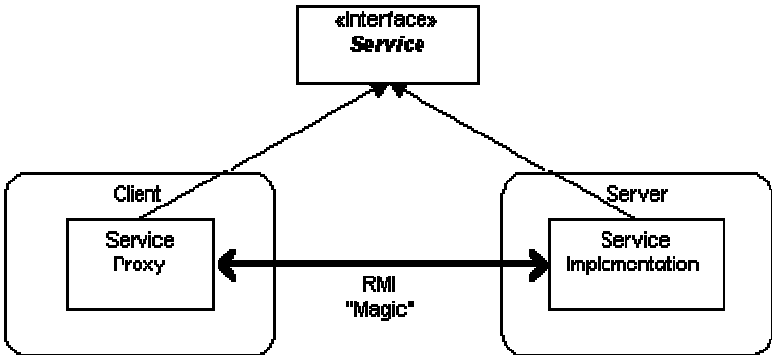
The RMI architecture is based on one important principle: the definition of behavior and the implementation of that behavior are separate concepts. RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs. This fits nicely with the needs of a distributed system where clients are concerned about the definition of a service and servers are focused on providing the service.

Specifically, in RMI, the definition of a remote service is coded using a Java interface. The implementation of the remote service is coded in a class. Therefore, the key to understanding RMI is to remember that interfaces define behavior and classes define implementation.

While the following diagram illustrates this separation, remember that a Java interface does not contain executable code.



RMI supports two classes that implement the same interface. The first class is the implementation of the behavior, and it runs on the server. The second class acts as a proxy for the remote service and it runs on the client. This is shown in the following diagram.



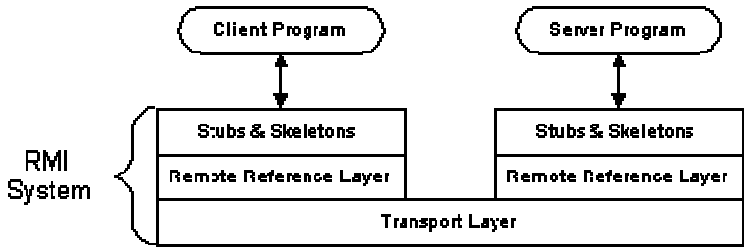
A client program makes method calls on the proxy object, RMI sends the request to the remote JVM, and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client's program.

RMI Architecture Layers

The RMI implementation is essentially built from three abstraction layers. The first is the Stub and Skeleton layer, which lies just beneath the view of the developer. This layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

The next layer is the Remote Reference Layer. This layer understands how to interpret and manage references made from clients to the remote service objects. In JDK 1.1, this layer connects clients to remote service objects that are running and exported on a server. The connection is a one-to-one (unicast) link. In the Java 2 SDK, this layer was enhanced to support the activation of dormant remote service objects via Remote Object Activation.

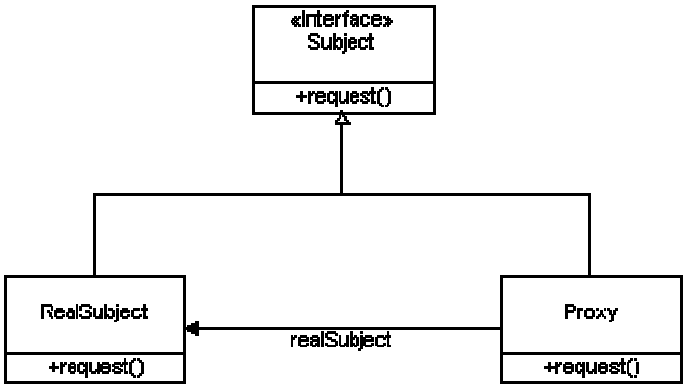
The transport layer is based on TCP/IP connections between machines in a network. It provides basic connectivity, as well as some firewall penetration strategies.



By using a layered architecture each of the layers could be enhanced or replaced without affecting the rest of the system. For example, the transport layer could be replaced by a UDP/IP layer without affecting the upper layers.

Stub and Skeleton Layer

The stub and skeleton layer of RMI lie just beneath the view of the Java developer. In this layer, RMI uses the Proxy design pattern. In the Proxy pattern, an object in one context is represented by another (the proxy) in a separate context. The proxy knows how to forward method calls between the participating objects. The following class diagram illustrates the Proxy pattern.



In RMI's use of the Proxy pattern, the stub class plays the role of the proxy, and the remote service implementation class plays the role of the RealSubject.

A skeleton is a helper class that is generated for RMI to use. The skeleton understands how to communicate with the stub across the RMI link. The skeleton carries on a

conversation with the stub; it reads the parameters for the method call from the link, makes the call to the remote service implementation object, accepts the return value, and then writes the return value back to the stub.

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses reflection to make the connection to the remote service object. You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

Remote Reference Layer

The Remote Reference Layer defines and supports the invocation semantics of the RMI connection. This layer provides a RemoteRef object that represents the link to the remote service implementation object. The stub objects use the invoke() method in RemoteRef to forward the method call. The RemoteRef object understands the invocation semantics for remote services.

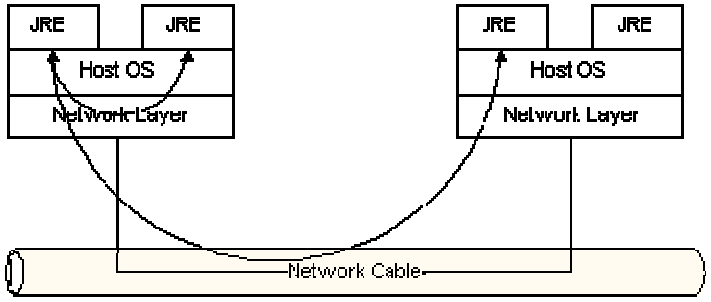
The JDK 1.1 implementation of RMI provides only one way for clients to connect to remote service implementations: a unicast, point-to-point connection. Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system. (If it is the primary service, it must also be named and registered in the RMI Registry).

The Java 2 SDK implementation of RMI adds a new semantic for the client-server connection. In this version, RMI supports activatable remote objects. When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant. If it is dormant, RMI will instantiate the object and restore its state from a disk file. Once an activatable object is in memory, it behaves just like JDK 1.1 remote service implementation objects.

Other types of connection semantics are possible. For example, with multicast, a single proxy could send a method request to multiple implementations simultaneously and accept the first reply (this improves response time and possibly improves availability). In the future, Sun may add additional invocation semantics to RMI.

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP. Even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack. (This is why you must have an operational TCP/IP configuration on your computer to run the Exercises in this chapter). The following diagram shows the unfettered use of TCP/IP connections between JVMs.



As you know, TCP/IP provides a persistent, stream-based connection between two machines based on an IP address and port number at each end. Usually a DNS name is used instead of an IP address. In the current release of RMI, TCP/IP connections are used as the foundation for all machine-to-machine connections.

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP). JRMP is a proprietary, stream-based protocol that is only partially specified is now in two versions. With the Java 2 SDK RMI service interfaces are not required to

extend from java.rmi.Remote and their service methods do not necessarily throw RemoteException.

The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles. While the transport layer prefers to use multiple TCP/IP connections, some network configurations only allow a single TCP/IP connection between a client and server (some browsers restrict applets to a single network connection back to their hosting server). In this case, the transport layer multiplexes multiple virtual connections within a single TCP/IP connection.

Naming Remote Objects

During the presentation of the RMI Architecture, one question has been repeatedly postponed: "How does a client find an RMI remote service? " Now you'll find the answer to that question. Clients find remote services by using a naming or directory service. This may seem like circular logic. How can a client locate a service by using a service? In fact, that is exactly the case. A naming or directory service is run on a well-known host and port number.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI). RMI itself includes a simple service called the RMI Registry, rmiregistry. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

On a host machine, a server program creates a remote service by first creating a local object that implements that service. Next, it exports that object to RMI. When the object is exported, RMI creates a listening service that waits for clients to connect and request the service. After exporting, the server registers the object in the RMI Registry under a public name.

On the client side, the RMI Registry is accessed through the static class Naming. It provides the method lookup() that a client uses to query a registry. The method lookup() accepts a URL that specifies the server host name and the name of the desired service. The method returns a remote reference to the service object. The URL takes the form:

rmi://<host_name> [:<name_service_port>] /<service_name>

where the host_name is a name recognized on the local area network (LAN) or a DNS name on the Internet. The name_service_port only needs to be specified only if the naming service is running on a different port to the default 1099.

Using RMI

In this section, you will build a simple remote calculator service and use it from a client program. A working RMI system is composed of several parts.

- Interface definitions for the remote services
- Implementations of the remote services
- Stub and Skeleton files
- A server to host the remote services
- An RMI Naming service that allows clients to find the remote services
- A class file provider (an HTTP or FTP server)
- A client program that needs the remote services

To simplify things, you can use a single directory for the client and server code. By running the client and the server out of the same directory, you will not have to set up an HTTP or FTP server to provide the class files. Following steps can be taken to build a system:

1. Write and compile Java code for interfaces
2. Write and compile Java code for implementation classes
3. Generate Stub and Skeleton class files from the implementation classes

4. Write Java code for a remote service host program
5. Develop Java code for RMI client program
6. Install and run RMI system

Interfaces

The first step is to write and compile the Java code for the service interface. The Calculator interface defines all of the remote features offered by the service:

```
public interface Calculator
        extends java.rmi.Remote {
   public long add(long a, long b)
      throws java.rmi.RemoteException;

   public long sub(long a, long b)
      throws java.rmi.RemoteException;

   public long mul(long a, long b)
      throws java.rmi.RemoteException;

   public long div(long a, long b)
      throws java.rmi.RemoteException;
}
```

Notice this interface extends Remote, and each method signature declares that it may throw a RemoteException object.

Store this file in one of your directories and compile it with the Java compiler:

```
>javac Calculator.java
```

Implementation

Next, you write the implementation for the remote service. This is the CalculatorImpl class:

```
public class CalculatorImpl
   extends java.rmi.server.UnicastRemoteObject
   implements Calculator {


        // Implementations must have an explicit constructor in order to declare the RemoteException
        exception
   public CalculatorImpl()
      throws java.rmi.RemoteException {
      super();
   }
    public long add(long a, long b)
      throws java.rmi.RemoteException {
      return a + b;
   }

   public long sub(long a, long b)
      throws java.rmi.RemoteException {
      return a - b;
   }

   public long mul(long a, long b)
      throws java.rmi.RemoteException {
      return a * b;
   }

   public long div(long a, long b)
      throws java.rmi.RemoteException {
      return a / b;
   }
}
```

Again, store this code into your directory and compile it.

The implementation class uses UnicastRemoteObject to link into the RMI system. In the example the implementation class directly extends UnicastRemoteObject. This is not a requirement. A class that does not extend UnicastRemoteObject may use its exportObject() method to be linked into RMI.

When a class extends UnicastRemoteObject, it must provide a constructor that declares that it may throw a RemoteException object. When this constructor calls super(), it activates code in UnicastRemoteObject that performs the RMI linking and remote object initialization.

Stubs and Skeletons

You next use the RMI compiler, rmic, to generate the stub and skeleton files. The compiler runs on the remote service implementation class file.

```
>rmic CalculatorImpl
```

Try this in your directory. After you run rmic you should find the file Calculator_Stub.class and, if you are running the Java 2 SDK, Calculator_Skel.class.

Options for the JDK 1.1 version of the RMI compiler, rmic, are:

```
Usage: rmic <options> <class names>
        where <options> includes:
        -keep  Do not delete intermediate generated source files
        -keepgenerated (same as "-keep")
        -g     Generate debugging info
        -depend   Recompile out-of-date  files recursively
        -nowarn   Generate no warnings
        -verbose   Output messages about what the compiler is doing
        -classpath <path>     Specify where to find input source and class files
        -d <directory>   Specify where to  place generated class files
        -J<runtime flag> Pass argument to the java interpreter
```

The Java 2 platform version of rmic add three new options:

```
        -v1.1  Create stubs/skeletons for JDK 1.1 stub protocol version
        -vcompat  (default)
              Create stubs/skeletons compatible with both JDK 1.1 and Java 2
                    stub protocol versions
```

-v1.2 Create stubs for Java 2 stub protocol version only

Host Server

Remote RMI services must be hosted in a server process. The class CalculatorServer is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;
public class CalculatorServer {

  public CalculatorServer() {
    try {
      Calculator c = new CalculatorImpl();
      Naming.rebind("rmi://localhost:1099/CalculatorService", c);
    } catch (Exception e) {
      System.out.println("Trouble: " + e);
    }
  }
  public static void main(String args[]) {
    new CalculatorServer();
  }
}
```

Client

The source code for the client follows:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class CalculatorClient {

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

```
        Calculator c = (Calculator)
                        Naming.lookup(          "rmi://localhost/CalculatorService");
        System.out.println( c.sub(4, 3) );
        System.out.println( c.add(4, 5) );
        System.out.println( c.mul(3, 6) );
        System.out.println( c.div(9, 3) );
      }
      catch (MalformedURLException murle) {
        System.out.println();
        System.out.println("MalformedURLException");
        System.out.println(murle);
      }
      catch (RemoteException re) {
        System.out.println();
        System.out.println("RemoteException");
        System.out.println(re);
      }
      catch (NotBoundException nbe) {
        System.out.println();
        System.out.println("NotBoundException");
        System.out.println(nbe);
      }
      catch ( java.lang.ArithmeticException ae) {
        System.out.println();
        System.out.println("java.lang.ArithmeticException");
        System.out.println(ae);
      }
    }
}
```

Running the RMI System

You are now ready to run the system. You need to start three consoles, one for the server, one for the client, and one for the RMIRegistry.

Start with the Registry. You must be in the directory that contains the classes you have written. From there, enter the following:

```
>rmiregistry
```

The registry will start running and you can switch to the next console. In the second console start the server hosting the CalculatorService, and enter the following:

```
>java CalculatorServer
```

It will start, load the implementation into memory and wait for a client connection. In the last console, start the client program.

```
>java CalculatorClient
```

If all goes well you will see the following output:

```
 1
9
18
3
```

That's it; you have created a working RMI system. Even though you ran the three consoles on the same computer, RMI uses your network stack and TCP/IP to communicate between the three separate JVMs. This is a full-fledged RMI system.

Parameters in RMI

You have seen that RMI supports method calls to remote objects. When these calls involve passing parameters or accepting a return value, how does RMI transfer these between JVMs? What semantics are used? Does RMI support pass-by-value or pass-by-reference? The answer depends on whether the parameters are primitive data types, objects, or remote objects.

Parameters in a Single JVM

First, review how parameters are passed in a single JVM. The normal semantics for Java technology is pass-by-value. When a parameter is passed to a method, the JVM makes a

copy of the value, places the copy on the stack and then executes the method. When the code inside a method uses a parameter, it accesses its stack and uses the copy of the parameter. Values returned from methods are also copies.

When a primitive data type (boolean, byte, short, int, long, char, float, or double) is passed as a parameter to a method, the mechanics of pass-by-value are straightforward. The mechanics of passing an object as a parameter are more complex. Recall that an object resides in heap memory and is accessed through one or more reference variables. And, while the following code makes it look like an object is passed to the method println()

```
String s = "Test";
System.out.println(s);
```

in the mechanics it is the reference variable that is passed to the method. In the example, a copy of reference variable s is made (increasing the reference count to the String object by one) and is placed on the stack. Inside the method, code uses the copy of the reference to access the object.

Now we will see how RMI passes parameters and return values between remote JVMs.

Primitive Parameters

When a primitive data type is passed as a parameter to a remote method, the RMI system passes it by value. RMI will make a copy of a primitive data type and send it to the remote method. If a method returns a primitive data type, it is also returned to the calling JVM by value.

Values are passed between JVMs in a standard, machine-independent format. This allows JVMs running on different platforms to communicate with each other reliably.

Object Parameters

When an object is passed to a remote method, the semantics change from the case of the single JVM. RMI sends the object itself, not its reference, between JVMs. It is the object that is passed by value, not the reference to the object. Similarly, when a remote method returns an object, a copy of the whole object is returned to the calling program.

Unlike primitive data types, sending an object to a remote JVM is a nontrivial task. A Java object can be simple and self-contained, or it could refer to other Java objects in complex graph-like structure. Because different JVMs do not share heap memory, RMI must send the referenced object and all objects it references. (Passing large object graphs can use a lot of CPU time and network bandwidth.)

RMI uses a technology called Object Serialization to transform an object into a linear format that can then be sent over the network wire. Object serialization essentially flattens an object and any objects it references. Serialized objects can be de-serialized in the memory of the remote JVM and made ready for use by a Java program.

Remote Object Parameters

RMI introduces a third type of parameter to consider: remote objects. As you have seen, a client program can obtain a reference to a remote object through the RMI Registry program. There is another way in which a client can obtain a remote reference, it can be returned to the client from a method call. In the following code, the BankManager service getAccount() method is used to obtain a remote reference to an Account remote service.

```
BankManager bm;
Account     a;
try {
   bm = (BankManager) Naming.lookup(
          "rmi://BankServer/BankManagerService"
       );
   a  = bm.getAccount( "ACCOUNT A" );
   // Code that uses the account
}
catch (RemoteException re) {
}
```

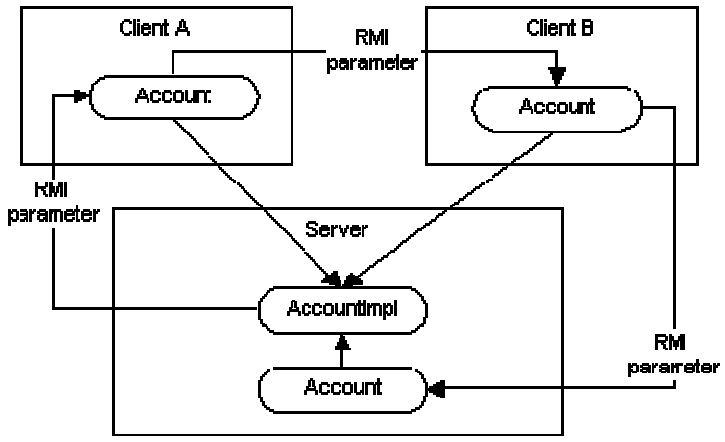
In the implementation of getAccount(), the method returns a (local) reference to the remote service.

```
public Account
  getAccount(String accountName) {
   // Code to find the matching account
   AccountImpl ai =
    // return reference from search
   return ai;
}
```

When a method returns a local reference to an exported remote object, RMI does not return that object. Instead, it substitutes another object (the remote proxy for that service) in the return stream.

The following diagram illustrates how RMI method calls might be used to:

- Return a remote reference from Server to Client A
- Send the remote reference from Client A to Client B
- Send the remote reference from Client B back to Server
-



Notice that when the AccountImpl object is returned to Client A, the Account proxy object is substituted. Subsequent method calls continue to send the reference first to Client B and then back to Server. During this process, the reference continues to refer to one instance of the remote service.

It is particularly interesting to note that when the reference is returned to Server, it is not converted into a local reference to the implementation object. While this would result in a

speed improvement, maintaining this indirection ensures that the semantics of using a remote reference is maintained.

3.5 JAVA DATA BASE CONNECTIVITY

Application architecture

One of the most important design issues when developing a Java database application is the overall system architecture; in particular, how many different components should be deployed. Traditionally, this is characterized by how many tiers, or layers, the application requires. There are two basic architectural models that can describe a system: the two-tier model and the n-tier model.

The two-tier model

The two-tier model is the traditional client-server framework; it has a client tier and a server tier. This simple model requires that the client be intimately aware of the database server. Thus, for example, the client needs database-specific code resulting in a tight coupling between the two tiers. This tight coupling has several advantages. First, it can decrease development time due to the fact the overall system is considerably simpler and smaller. Second, the tight coupling can potentially improve system performance as the client can easily take advantage of specific server functionality that might not be available to a less tightly coupled system. On the other hand, this tight coupling can lead to several problems. Most notably, system maintenance can become more difficult because changes in the server can break the client or visa versa. Furthermore, if the database changes, all of the client code will need to be modified. If the client is highly distributed, propagating changes throughout the system can be difficult, and in some scenarios impossible. As a result, two-tier applications can be useful in a corporate LAN environment where complete control of all clients is achieved.

The n-tier model

The n-tier model has a client tier, at least one server tier, and at least one middle layer. Because of the extra tier, many of the problems that affected the two-tier model are no longer an issue. For example, the middle layer now maintains the database connection information. This means the clients only need to know about the middle tier. Because the middle tier is generally operating at the same physical location as the server (for instance, both components can be behind the same firewall), maintaining the middle tier is considerably easier than maintaining several hundred client installations. Another advantage of the n-tier approach is that the overall system can easily scale to handle more users. All one needs to do is add more middle tiers or server tiers, depending on the results of the profiling operations. Because middle tiers are typically implemented using Web servers -- using JavaServer Pages and Servlets technologies -- it is simple to add load-balancing or even new hardware components. Java language provides many of the necessary components, pre-built, for constructing n-tier applications.

Introduction to JDBC

SQL is a language used to create, manipulate, examine, and manage relational databases. Because SQL is an application-specific language, a single statement can be very expressive and can initiate high-level actions, such as sorting and merging data. SQL was standardized in 1992 so that a program could communicate with most database systems without having to change the SQL commands. Unfortunately, you must connect to a database before sending SQL commands, and each database vendor has a different interface, as well as different extensions of SQL.

ODBC, a C-based interface to SQL-based database engines, provides a consistent interface for communicating with a database and for accessing database metadata (information about the database system vendor, how the data is stored, and so on). Individual

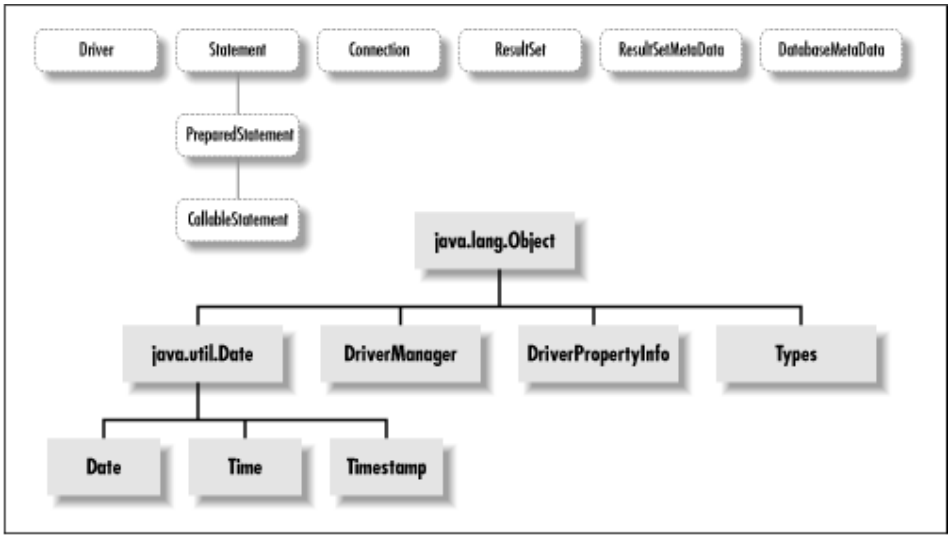
vendors provide specific drivers or "bridges" to their particular database management system. Consequently, thanks to ODBC and SQL, you can connect to a database and manipulate it in a standard way.

Though SQL is well suited for manipulating databases, it is unsuitable as a general application language and programmers use it primarily as a means of communicating with databases--another language is needed to feed SQL statements to a database and process results for visual display or report generation. Unfortunately, you cannot easily write a program that will run on multiple platforms even though the database connectivity standardization issue has been largely resolved. For example, if you wrote a database client in C++, you would have to totally rewrite the client for each platform; that is to say, your PC version would not run on a Macintosh. There are two reasons for this. First, C++ as a language is not portable for the simple reason that C++ is not completely specified, for example, how many bits does an `int` hold? Second and more importantly, support libraries such as network access and GUI libraries are different on each platform.

You can run a Java program on any Java-enabled platform without even recompiling that program. The Java language is completely specified and, by definition, a Java-enabled platform must support a known core of libraries. One such library is JDBC (Java Database Connectivity), which you can think of as a Java version of ODBC. JDBC allows us to construct SQL statements and embed them inside Java API calls. Database vendors are already busy creating bridges from the JDBC API to their particular systems. JavaSoft has also provided a bridge driver that translates JDBC to ODBC, allowing you to communicate with legacy databases that have no idea that Java exists. Using Java in conjunction with JDBC provides a truly portable solution to writing database applications. The JDBC-ODBC bridge driver is just one of four types of drivers available to support JDBC connectivity. It comes packaged with the JDK 1.1 or as a separate package for use with 1.0 systems.

JDBC driver fundamentals

A casual inspection of the JDBC API quickly shows the dominance of interfaces within the API, which might lead a user to wonder where the work is done. Actually this is strength of the approach that the JDBC developers took because the actual implementation is provided by JDBC Driver vendors, who in turn provide the classes that implement the necessary interfaces. Following figure shows the classes and interfaces of java.sql, the JDBC API package.



From a programming perspective, there are two main classes responsible for establishing a connection with a database. The first class is **DriverManager**, which is one of the few actual classes provided in the JDBC API. **DriverManager** is responsible for managing a pool of registered drivers, essentially abstracting the details of using a driver so that programmer does not have to deal with them directly. The second class is the actual JDBC **Driver** class. These are provided by independent vendors. The JDBC Driver class is responsible for establishing the database connection and handling all of the communication with the database. JDBC drivers come in four different types.

Registering a JDBC driver

The first step in the process of creating a connection between a Java application and a database is the registration of a JDBC driver with the Java virtual machine (JVM) in which the Java application is running. The connection and all database communications are controlled by the **Driver Manager** object. To establish a connection, a suitable JDBC driver for the target database must be registered with the **DriverManager** object.

The JDBC specification, states that JDBC drivers are supposed to register themselves with the **DriverManager** object automatically when they are loaded into a JVM. For example, the following code snippet uses a static initializer to first create an instance of the **persistentjava** JDBC driver and then register it with the **DriverManager**.

```
static {
java.sql.DriverManager.registerDriver(new com.persistentjava.JdbcDriver()) ;
}
```

Registering a driver is simply a matter of loading the driver class into the JVM, which can be done in several different ways. One way to do this is with the ClassLoader **Class.forName(com.persistentjava.JdbcDriver) ;**. Another method, which is not as well known, uses the **jdbc.drivers** system property. This method can be used in one of three different ways:

* From the command line:

    java -Djdb.drivers=com.persistentjava.JdbcDriver Connect

* Within the java application:

    System.setProperty("jdbc.drivers", "com.persistentjava.JdbcDriver") ;

* By setting the **jdbc.drivers** property in the System property file, which is generally system dependent

By separating the drivers with a colon, multiple drivers can be registered using the aforementioned system property technique. One of the benefits of using the system property technique is that drivers can be easily swapped in and out of the JVM without modifying any code (or at least with minimal code changes). If multiple drivers are registered, their order of precedence is: 1) JDBC drivers registered by the **jdbc.drivers** property at JVM initialization, and 2) JDBC drivers dynamically loaded. Because the **jdbc.drivers** property is only checked once upon the first invocation of a **DriverManager()** method, it's important to ensure all drivers are registered correctly before establishing the database connection.

Not all JVMs are created equal, however, and some JVMs do not follow the JVM specification. As a result, static initializers do not always work as advertised. This results in multiple ways to register a JDBC driver, including:

* Class.forName("com.persistentjava.JdbcDriver").newInstance();

* DriverManager.registerDriver(new com.persistentjava.JdbcDriver()) **;**

One final issue is that **Class.forname()** can throw a **ClassNotFoundException**, so you need to wrap the registration code in an appropriate exception handler.

JDBC driver URLs

Once a JDBC driver has been registered with the **DriverManager**, it can be used to establish a connection to a database. But how does the **DriverManager** select the right driver, given that any number of different drivers might actually be registered? The technique is quite simple: each JDBC driver uses a specific JDBC URL (which has the same format as web addresses) as a means of self-identification. The format of the URL is straightforward: **jdbc:sub-protocol:database locator**. The sub-protocol is specific to the JDBC driver and

can be odbc, oracle, db2, and so on depending on the actual JDBC driver vendor. The database locator is a driver-specific indicator for uniquely specifying the database with which an application wants to interact. Depending on the type of driver, this locater may include a hostname, a port, and a database system name.

When presented with a specific URL, the **DriverManager** iterates through the collection of registered drivers until one of the drivers recognizes the specified URL. If no suitable driver is found, an **SQLException** is thrown. The following list demonstrates several specific examples of actual JDBC URLs:

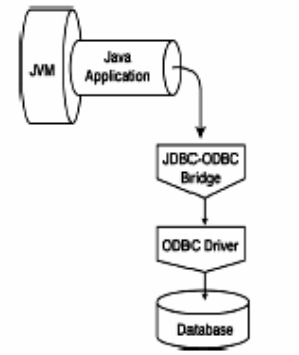
* jdbc:odbc:jdbc
* jdbc:oracle:thin:@persistentjava.com:1521:jdbc;
* jdbc:db2:jdbc

Many drivers, including the JDBC-ODBC bridge driver, accept additional parameters at the end of the URL such as username and password. The method for obtaining a database connection, given a specific JDBC URL, is to call **getConnection()** on the **DriverManager** object. This method comes in several flavors:

* DriverManager.getConnection(url) ;
* DriverManager.getConnection(url, username, password) ;
* DriverManager.getConnection(url, dbproperties) ;

Here, **url** is a **String** object that is the JDBC URL; username and password are **String** objects that are the username and password that the JDBC application should use to connect to the data source; and **dbproperties** is a Java **properties** object that encapsulates all of the parameters (possibly including username and password) that a JDBC driver requires to successfully make a connection. Now that we have the driver basics in hand, we can examine the individual driver types in more detail.

Type one drivers



Type one drivers come in one variety: they all use the JDBC-ODBC bridge, which is included as a standard part of the JDK. Type one drivers are differentiated by the ODBC (Open Database Connectivity) driver attached to the JDBC-ODBC bridge. To connect to a different data source, you simply have to register (or effectively bind) a different ODBC data source, using the ODBC Administrator, to the appropriate data source name.

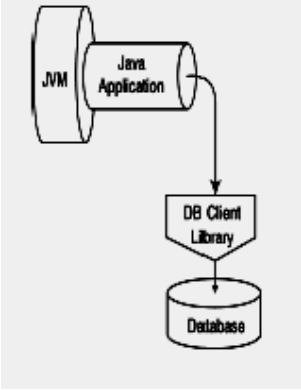
The problem for type one drivers is their use in distributed applications. Because the bridge itself does not support distributed communication, the only way type one drivers can work across a network is if the ODBC driver itself supports remote interactions. For simple ODBC drivers, this is not an option, and while big databases do typically have ODBC drivers that can work remotely, they cannot compete performance-wise with the pure Java JDBC drivers.

The class name for the JDBC-ODBC bridge driver is **sun.jdbc.odbc.JdbcOdbcDriver** and the JDBC URL takes the form **jdbc:odbc:dsn**, where **dsn** is the Data Source Name used to register the database with the ODBC Administrator. For example, if a database is registered with an ODBC data source name of d_source, a username of java, and a password of sun, the following code snippet can be used to establish a connection.

```
String url = "jdbc:odbc:d_source" ;
Connection con ;
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver") ;
```

```
} catch(java.lang.ClassNotFoundException e) {
System.err.print("ClassNotFoundException: ") ;
System.err.println(e.getMessage()) ;
return ;
}
try {
con = DriverManager.getConnection(url, "java", "sun");
} catch(SQLException ex) {
System.err.println("SQLException: " + ex.getMessage());
} finally {
try{
con.close ;
} catch(SQLException ex) {
System.err.println(SQLException: " + ex.getMessage()) ;
}
}
```

Type two drivers



Type two drivers are also known as partial Java drivers, in that they translate the JDBC API directly into a database-specific API. The database client application (the host that is running the JVM) must have the appropriate database client library, which might include binary code installed and possibly running. But, using a type two model restricts the developer to client platforms and operating systems supported by the database vendor's client library.

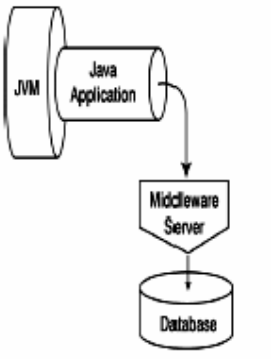
This model can work effectively, however, when the client base is tightly controlled. This typically occurs in corporate LANs. One example of a type two driver is the DB2 JDBC

application driver. The following example demonstrates how to establish a connection using a DB2 driver.

```
String url = "jdbc:db2:d_source" ;
try {
Class.forName("COM.server1.db2.jdbc.app.DB2Driver") ;
} catch(java.lang.ClassNotFoundException e) {
System.err.print("ClassNotFoundException: ") ;
System.err.println(e.getMessage()) ;
return ;
}
...
```

Type three drivers

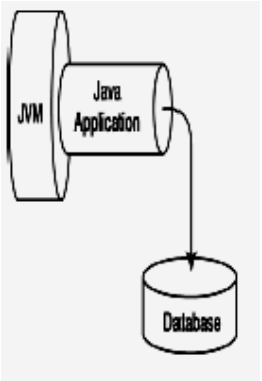
Type three drivers are pure Java drivers that transform the JDBC API into a database-independent protocol. The JDBC driver does not communicate with the database directly; it communicates with a middleware server, which in turn communicates with the database. This extra level of indirection provides flexibility in that different databases can be accessed from the same code because the middleware server hides the specifics from the Java application. To switch to a different database, you only need to change parameters in the middleware server. (One point of note: the database format you are accessing must be supported by the middleware server.)



The downside to type three drivers is that the extra level of indirection can hurt overall system performance. On the other hand, if an application needs to interact with a variety of database formats, a type three driver is an efficient approach due to the fact that the same JDBC driver is used regardless of the underlying database. In addition, because the middleware server can be installed on a specific hardware platform, certain optimizations can be performed to capitalize on profiling

results.

Type four drivers



Type four drivers are pure Java drivers that communicate directly with the database. Many programmers consider this the best type of driver, as it typically provides optimal performance and allows the developer to leverage database-specific functionality. Of course this tight coupling can hinder flexibility, especially if you need to change the underlying database in an application. This type of driver is often used in applets and other highly distributed applications.

A Complete JDBC Example

Running through a simple example will help you grasp the overall concepts of JDBC. The fundamental issues encountered when writing any database application are:

- **Creating a database**. You can either create the database outside of Java, via tools supplied by the database vendor, or via SQL statements fed to the database from a Java program.
- **Connecting to an ODBC data source**. An ODBC data source is a database that is registered with the ODBC driver. In Java you can use either the JDBC to ODBC bridge, or JDBC and a vendor-specific bridge to connect to the datasource.
- **Inserting information into a database**. Again, you can either enter data outside of Java, using database-specific tools, or with SQL statements sent by a Java program.
- **Selectively retrieving information**. You use SQL commands from Java to get results and then use Java to display or manipulate that data.

Creating a Database

For this example, consider the scenario of tracking coffee usage at the XYZ University Cafe. A weekly report must be generated for University management that includes total coffee sales and the maximum coffee consumed by a programmer in one day. Here is the data:

Coffee Consumption at Cafe Jolt, XYZ University

Programmer	Day	# Cups
Amith	Mon	1
Bhavani	Mon	2
Prashanth	Tue	8
Ramesh	Tue	2
Rajesh	Tue	3
Sumanth	Wed	2
John	Thu	3
Peter	Thu	1
Sandhya	Fri	9
Sudha	Fri	3
Sathish	Fri	4

To create this database, you can feed SQL statements to an ODBC data source via the JDBC-ODBC bridge. First, you will have to create an ODBC data source. You have many choices—you could, for example, connect an Oracle or Sybase or MS Access database. For simplicity create a MS Access ODBC data source to use for this example. Call this ODBC data source CafeJolt.

NOTE: There are basically two steps in creating a MS Access data source: The first is to create an ODBC-accessible data source to use, and the second is register it with your system. Perform the following tasks:

1. Pick a system to use as the data source.
    - Microsoft Access
        - Make a directory for the database
        - Create a new database file, testdb.mdb
2. From Windows 95/NT 4.0:
    - Bring up Control Panel
        - Select the Start button
        - Select the Settings menu item
        - Select the Control Panel menu item
    - Find and double-click on the ODBC Icon (32-bit/with **32** on it). This brings up the 'Data Sources' window.
    - Select Add. This brings up the Add Data Source window.
    - Select the driver for the type of driver you want.
      If you selected Microsoft Access, the ODBC Microsoft Access 7.0 Setup window appears.
        - Name the data source
        - Fill in a description.
        - Click on the Select button to bring up a file dialog.
        - Locate the directory created in task 1.
        - Select OK to accept new driver.

To enter the data into the testdb database, create a Java **application** that follows these steps:

1. Load the JDBC-ODBC bridge. You must load a driver that tells the JDBC classes how to talk to a data source. In this case, you will need the class JdbcOdbcDriver:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

This can also be specified from the command line via the jdbc.drivers system property:

java -Djdbc.drivers=sun.jdbc.odbc.JdbcOdbcDriver AProgram

2. Connect to a data source. A URL is used to connect to a particular JDBC data source. Given that you have loaded the JDBC-ODBC bridge driver, URLs should be in the following form jdbc:odbc:data-source-name. Using the DriverManager class, you request a connection to a URL and the DriverManager selects the appropriate driver; here, only the driver JdbcOdbcDriver is loaded.

Connection con = DriverManager.getConnection(URL, username, password);

Where the username and password are empty strings, "", in this case. Or they can be left out., by just entering the URL.

Example : Connection con = DriverManager.getConnection(Jdbc:odbc: CafeJolt);

3. Send SQL statements to create the table. Ask the connection object for a Statement object:

Statement stmt = con.createStatement();

Then, execute the following SQL statement to create a table called JoltData.

```
create table JoltData (
programmer varchar (32),
day char (3),
cups integer,
variety varchar (20));
```

The Java code to do this is:

```
stmt.execute(   "create table JoltData ("+
                "programmer varchar (32),"+
```

```
            "day char (3),"+
            "cups integer);"
        );
```

After you have created the table, you can the insert the appropriate values such as:

```
insert into JoltData values ('Gilbert', 'Mon', 1);
insert into JoltData values ('Wally', 'Mon', 2);
insert into JoltData values ('Edgar', 'Tue', 8);
...
```

Here is the Java source for a complete application to create table JoltData and insert the required rows.

```
import java.sql.*;  //import all the JDBC classes

public class CreateJoltData {

   static String[] SQL = {
      "create table JoltData ("+
         "programmer varchar (32),"+
         "day varchar (3),"+
         "cups integer);",
      "insert into JoltData values ('Amith', 'Mon', 1);",
      "insert into JoltData values ('Bhavani', 'Mon', 2);",
      "insert into JoltData values ('Prashanth', 'Tue', 8);",
      "insert into JoltData values ('Ramesh', 'Tue', 2);",
      "insert into JoltData values ('Rajesh', 'Tue', 3);",
      "insert into JoltData values ('Sumanth', 'Wed', 2);",
      "insert into JoltData values ('John', 'Thu', 3);",
      "insert into JoltData values ('Peter', 'Thu', 1);",
      "insert into JoltData values ('Sandhya', 'Fri', 9);",
      "insert into JoltData values ('Sudha', 'Fri', 3);",
      "insert into JoltData values ('Sathish', 'Fri', 4);",
```

```
};

public static void main(String[] args) {
   String URL = "jdbc:odbc:CafeJolt";
   String username = "";     // Users can leave out the user
   String password = "";     // name and password

   try {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   } catch (Exception e) {
      System.out.println("Failed to load JDBC/ODBC driver.");
      return;
   }

   Statement stmt = null;
   Connection con=null;
   try {
      con = DriverManager.getConnection (
         URL,
         username,
         password); // username ,password can be left out
      stmt = con.createStatement();
   } catch (Exception e) {
      System.err.println("problems connecting to "+URL);
   }

   try {
      // execute SQL commands to create table, insert data
      for (int i=0; i<SQL.length;i++){
                  stmt.execute(SQL[i]);
                  }
                  con.close();
                  }catch (Exception e){
```

```
                System.out.println("problems with SQL sent to "+URL+": "+e.getMessage());
        }

  }
}
```

Review what you have done so far. After creating a data source visible to ODBC, you connected to that source via the JDBC-ODBC bridge and sent a series of SQL statements to create a table called JoltData filled with rows of data. You can examine the contents of your "database".

Getting Information from a Database

To retrieve information from a database, use SQL select statements via the Java Statement.executeQuery method, which returns results as rows of data in a ResultSet object. The results are examined row-by-row using the ResultSet.next and ResultSet.getXXX methods.

Consider how you would obtain the maximum number of cups of coffee consumed by a programmer in one day. In terms of SQL, one way to get the maximum value is to sort the table by the cups column in descending order. The programmer column is selected, so the name attached to the most coffee consumption can also be printed. Use the SQL statement:

```
SELECT programmer, cups FROM JoltData ORDER BY cups DESC;
```

From Java, execute the statement with:

```
ResultSet result = stmt.executeQuery(
  "SELECT programmer,
  cups FROM JoltData ORDER BY cups DESC;");
```

The cups column of the first row of the result set will contain the largest number of cups:

Sandhya	9
Prashanth	8
Sathish	4
Rajesh	3
John	3
Sudha	3
Bhavani	2
Ramesh	2
Sumanth	2
Amith	1
Peter	1

Examine the ResultSet by:

1. "Moving" to the first row of data. Perform:

```
result.next();
```

2. Extracting data from the columns of that row. Perform:

```
String name = result.getString("programmer");
int cups = result.getInt("cups");
```

The information can be printed easily via:

```
System.out.println("Programmer "+name+
```

```
        " consumed the most coffee: "+cups+" cups.");
```

resulting in the following output:

Programmer Sandhya consumed the most coffee: 9 cups.

Computing the total sales for the week is a matter of adding up the cups column. Use an SQL select statement to retrieve the cups column:

```
result = stmt.executeQuery(
        "SELECT         cups FROM JoltData;");
```

Peruse the results by calling method next until it returns false, indicating that there are no more rows of data:

```
// for each row of data
cups = 0;
while(result.next()) {
        cups += result.getInt("cups");
}
```

Print the total number of cups sold:

```
System.out.println("Total sales of "+cups+" cups of coffee.");
```

The output should be:

Total sales of 38 cups of coffee.

Here is the Java source for a complete application to examine the JoltData table and generate the report.

```
import java.sql.*;

public class JoltReport {
   public static void main (String args[]) {
      String URL = "jdbc:odbc:CafeJolt";
      String username = "";
      String password = "";

      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (Exception e) {
         System.out.println("Failed to load JDBC/ODBC driver.");
         return;
      }

      Statement stmt = null;
      Connection con=null;
      try {
         con = DriverManager.getConnection (
            URL,
            username,
            password);
         stmt = con.createStatement();
      } catch (Exception e) {
         System.err.println("problems connecting to "+URL);
      }

      try {
         ResultSet result = stmt.executeQuery(
```

```
      "SELECT programmer, cups FROM JoltData ORDER BY cups DESC;");
         result.next();  // move to first row
         String name = result.getString("programmer");
         int cups = result.getInt("cups");
         System.out.println("Programmer "+name+
            " consumed the most coffee: "+cups+" cups.");

         result = stmt.executeQuery(
            "SELECT cups FROM JoltData;");

         // for each row of data
         cups = 0;
         while(result.next()) {
            cups += result.getInt("cups");
         }
      System.out.println("Total sales of "+cups+" cups of coffee.");

         con.close();
       }
       catch (Exception e) {
        e.printStackTrace();
       }
    }
}
```

Obtaining Result MetaData Type Information

You will occasionally need to obtain type information about the result of a query. For example, the SQL statement:

```
SELECT * from JoltData
```

will return a ResultSet with the same number of columns (and rows) as the table, JoltData. If you do not know how many columns there are beforehand, you must use metadata via the ResultSetMetaData class to find out. Continuing the Cafe Jolt scenario, determine the number and type of columns returned by the same SQL query

```
SELECT programmer, cups FROM JoltData ORDER BY cups DESC;
```

First, perform the usual execute method call:

```
ResultSet result = stmt.executeQuery(
   "SELECT programmer,
   cups FROM JoltData ORDER BY cups DESC;");
```

Then obtain the column and type metadata from the ResultSet:

```
ResultSetMetaData meta = result.getMetaData();
```

You can query the ResultSetMetaData easily to determine how many columns there are:

```
 int columns = meta.getColumnCount();
```

and then walk the list of columns printing out their name and type:

```
int numbers = 0;
for (int i=1;i<=columns;i++) {
  System.out.println (meta.getColumnLabel(i) + "\t"
              + meta.getColumnTypeName(i));
  if (meta.isSigned(i)) { // is it a signed number?
       numbers++;
  }
```

```
}
System.out.println ("Columns: " +
  columns + " Numeric: " + numbers);
```

Here is the Java source for a complete application to print out some metadata associated with the results of the query.

```
import java.sql.*;

public class JoltMetaData {
   public static void main (String args[]) {
      String URL = "jdbc:odbc:CafeJolt";
      String username = "";
      String password = "";

      try {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      } catch (Exception e) {
         System.out.println("Failed to load JDBC/ODBC driver.");
         return;
      }

      Statement stmt = null;
      Connection con=null;
      try {
         con = DriverManager.getConnection (
            URL,
            username,
            password);
         stmt = con.createStatement();
      } catch (Exception e) {
         System.err.println("problems connecting to "+URL);
```

```
      }

      try {
         ResultSet result = stmt.executeQuery(
      "SELECT programmer, cups FROM JoltData ORDER BY cups DESC;");
         ResultSetMetaData meta = result.getMetaData();

         int numbers = 0;
         int columns = meta.getColumnCount();
         for (int i=1;i<=columns;i++) {
            System.out.println (meta.getColumnLabel(i) + "\t"
                     + meta.getColumnTypeName(i));
            if (meta.isSigned(i)) { // is it a signed number?
               numbers++;
            }
         }
 System.out.println ("Columns: " + columns + " Numeric: " + numbers);

         con.close();
      }
      catch (Exception e) {
        e.printStackTrace();
      }
   }
}
```

Connecting a Java program to a database

Currently, there are two choices for connecting your Java program to a data source. First, you can obtain a JDBC driver from your database vendor that acts as a bridge from

JDBC to their database connection interface. Second, JavaSoft provides a JDBC-ODBC bridge called class JdbcOdbcDriver and, hence, you can connect to any ODBC data source.

Once you have established a JDBC database link, open a connection to that data source through a Connection object obtained via DriverManager.getConnection, which selects the appropriate driver for talking with that source. All ODBC data sources are identified via a URL in the form:

jdbc:odbc:data-source-name

The getConnection method returns a Connection to the specified source using the JDBC-ODBC driver. For example, to connect to an ODBC source called mage with a user name of parrt and a password of mojava, you would use:

```
// load the JDBC-ODBC bridge by referencing it
try {
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
} catch (Exception e) {
  System.out.println(
    "Failed to load JDBC/ODBC driver.");
  return;
}
// get a connection
try {
  con = DriverManager.getConnection(
        "jdbc:odbc:mage",
        "parrt",
        "mojava");
} catch (Exception e) {
  System.err.println("problems connecting to "+URL);
}
```

Given a connection, you can create statements, execute queries, and so on.See the NOTE above for specific information about setting up ODBC data sources on a PC.

Talking to a Database

Given a connection to a database, you can send SQL statements to manipulate that database. Using the Connection.createStatement method, obtain a Statement object and then execute method executeQuery or executeUpdate. JDBC does not put any restrictions on the SQL you send via the execute methods. but you must ensure that the data source you are connecting to supports whatever SQL you are using. To be JDBC-compliant, however, the data source must support at least SQL-2 Entry Level capabilities.

Database Updates

Assuming the variable con contains a valid Connection object obtained from the method DriverManager.getConnection, simple SQL update statements (SQL INSERT, UPDATE or DELETE) can be sent to your database by creating a Statement and then calling method executeUpdate. For example, to create a table called Data with one row of data, use the following:

```
Statement stmt = con.createStatement();
// create table with name and height columns
stmt.executeUpdate(
  "create table Data (
    name varchar (32), height integer);");
stmt.executeUpdate(  "insert into Data values ('John', 68);");
con.close();
// close connection, committing transaction.
```

The method executeUpdate returns the number of rows affected with 0 for SQL statements that return nothing.

Database Queries

In order to query a database (via the SQL SELECT statement), use the method executeQuery, which returns a ResultSet object. The ResultSet object returned is never null and contains the rows of data selected by the SQL statement. For example, the following code fragment selects two columns of data from our table called Data in ascending height order:

```
ResultSet result = stmt.executeQuery(
  "SELECT name, height
    FROM Data ORDER BY height ASC;");
```

The rows of resulting data are accessed in order, but the elements in the various columns can be accessed in any order. However, for maximum portability among databases, JavaSoft recommends that the columns be accessed in order from left-to-right, and that each row be read only once. There is a "row cursor" in the result that points at the current row. The method ResultSet.next moves the cursor from row to row. Before reading the first row, call method next to initialize the cursor to the first row. The following code fragment shows how to read the first two rows of data and print them out.

```
 String name;
int height;
if ( result.next() ) {  // move to first row
  name = result.getString("name");
  height = result.getInt("height");
  System.out.println(name+":"+height);
}
if ( result.next() ) {      // get second row
  name = result.getString("name");
  height = result.getInt("height");
  System.out.println(name+":"+height);
}
```

The method next returns false when another row is not available.

Note that column names are not case-sensitive, and if more than one column has the same name, the first one is accessed. Where possible, the column index should be used. You can ask the ResultSet for the index of a particular column if you do not know it beforehand.

```
int nameCol = result.findColumn ("name");
int heightCol = result.findColumn ("height");
```

Information about the properties of a ResultSet column is provided by the class ResultSetMetaData and returned by the ResultSet.getMetaData method.

Prepared Statements

When performing the same operation multiple times, use a PreparedStatement for runtime efficiency, which is precompiled by the SQL engine (assuming your data source supports this feature). Prepared statements are also useful when you have lots of arguments to specify for a particular SQL command.

PreparedStatement is an extension of Statement and, consequently, behaves like a Statement except that you create them with the method Connection.prepareStatement, instead of the method Connection.createStatement:

```
PreparedStatement prep = con.prepareStatement(
   "INSERT into Data values (?, ?)");
```

The IN arguments, indicated by '?', can be filled by in by setXXX methods. Ensure that the parameter (indicated by an index number starting from 1) you are setting matches the type of the value you are passing. For example, for a table called Data with two columns, name of type varchar and height of type integer, the parameters to prep can be set via:

```
prep.setString(1, "Jim");
prep.setInt(2, 70);
```

Finally, execute the the prepared statement with the parameters set most recently via the executeUpdate method:

```
if (prep.executeUpdate () != 1) {
   throw new Exception ("Bad Update");
}
```

Metadata

You can access information about the database as a whole, or about a particular query ResultSet. This section describes how DatabaseMetaData and ResultSetMetaData objects are obtained and queried.

Information about a database

When you need to know about the capabilities, or the vendor, of a database, ask the associated Connection object for its metadata:

```
Connection con = DriverManager.getConnection(
   "jdbc:odbc:mydatasource",
   "user", "password");
DatabaseMetaData md = con.getMetaData();
```

There are many questions you can ask. For example, the following code fragment asks the database for its product name and how many simultaneous connections can be made to it.

```
if (md==null) {
  System.out.println("No Database Meta Data");
```

```
} else {
  System.out.println("Database Product Name   : " +
       md.getDatabaseProductName());
  System.out.println("Allowable active connections: "+
       md.getMaxConnections());
}
```

Information about a table within a database

To find out the number and types of the columns in a table accessed via a ResultSet, obtain a ResultSetMetaData object.

```
ResultSet result = ...;
ResultSetMetaData meta = result.getMetaData();
int numbers = 0;
int columns = meta.getColumnCount();
System.out.println ("Columns: " + columns);
```

Transactions

A transaction is a set of statements that have been executed and committed or rolled back. To commit a transaction, call the method commit on the appropriate connection; use the rollback to remove all changes since the last commit. By default, all new connections are in auto-commit mode, which means that each "execute" is a complete transaction. Call Connection.setAutoCommit to change the default. Any locks held by a transaction are released upon the method commit.

```
Connection con = DriverManager.getConnection(...);
con.setAutoCommit(false);
Statement s = con.createStatement();
s.executeUpdate("SQL statement 1");
s.executeUpdate("SQL statement 2");
s.executeUpdate("SQL statement 3");
```

```
con.commit();
//transaction (3 statements) committed here
```

All JDBC-compliant drivers support transactions. Check the DatabaseMetaData associated with the appropriate connection to determine the level of transaction-support a database provides.

Stored Procedures

A stored procedure is a block of SQL code stored in the database and executed on the server. The CallableStatement interface allows you to interact with them. Working with CallableStatement objects is very similar to working with PreparedStatements. The procedures have parameters and can return either ResultSets or an update count. Their parameters can be either input or output parameters. Input parameters are set via the setXXX methods. Output parameters need to be registered via the CallableStatement.registerOutParamter method. Stored procedures need to be supported by the database in order to use them. You can ask the DatabaseMetaData if it supports it via the method supportsStoredProcedures.

To demonstrate this interaction, return to Cafe Jolt. And, instead of a weekly total, the manager asks for the daily total of a particular day of the week. You can create a stored procedure to help, but this is usually created by the database developer, not the applications programmer. Once the procedure is created, the user does not need to know how it works internally, just how to call it.

Like any other SQL statement, you need a Connection and a Statement to create the procedure:

```
Statement stmt = con.createStatement();
```

Then execute the SQL statement:

```
CREATE PROCEDURE getDailyTotal
    @day char(3), @dayTotal int output
    AS
    BEGIN
      SELECT @dayTotal = sum (cups)
      FROM JoltData
      WHERE day = @day
    END
```

The Java code is:

```
stmt.execute ("CREATE PROCEDURE getDailyTotal " +
    "@day char(3), @dayTotal int output " +
    "AS " +
    "BEGIN " +
    "   SELECT @dayTotal = sum (cups) " +
    "   FROM JoltData " +
    "   WHERE day = @day " +
    "END"
);
```

Once created, you call it through a CallableStatement:

```
CallableStatement cstmt = con.prepareCall (
  "{call getDailyTotal (?, ?)}");
cstmt.setString (1, "Mon");
cstmt.registerOutParameter (2, java.sql.Types.INTEGER);
cstmt.executeUpdate();
System.out.println ("Total is " + cstmt.getInt (2));
```

The exact syntax to create a stored procedure may differ between database vendors. Also, if there are no output parameters for the stored procedure, you can it by using a PreparedStatement. And, if there happens to be no input or output parameters, you can use a Statement.

JDBC Exception Types

JDBC provides three types of exceptions:

- SQLException
- SQLWarning
- DataTruncation

SQLExceptions

The SQLException is the basis of all JDBC exceptions. It consists of three pieces of information, a String message, like all children of Exception, another String containing the XOPEN SQL state (as described by specification), and a driver/source specific int for an additional error code.

In addition, multiple SQLException instances can be chained together.

SQLWarnings

The SQLWarning class is similar to SQLException, however it is considered a noncritical error and is not thrown. It is up to the programmer to poll for SQLWarning messages through the getWarnings methods of Connection, ResultSet, and Statement. If you do not poll for them, you will never receive them. Also, the next time something is done with a Connection, ResultSet, or Statement, the previous warnings are cleared out.

Data Truncation

The DataTruncation class is a special type of SQLWarning. It is reported with other SQLWarning instances and indicates when information is lost during a read or write operation. To detect a truncation, it is necessary to perform an instanceof DataTruncation check on each SQLWarning from a getWarnings chain.

3.6 SERVLETS

Introduction to servlets

What is a Servlet?

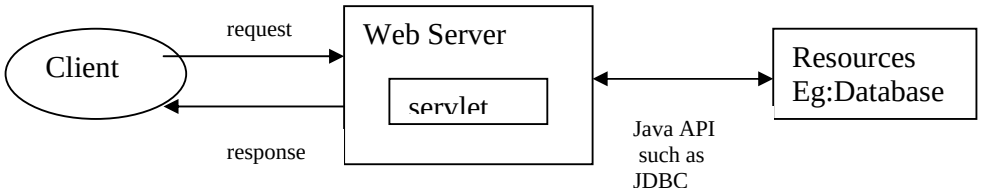
A servlet is a server-side software program, written in Java that handles messaging between a client and server. The Java Servlet API defines a standard interface for the request and response messages so that your servlets can be portable across platforms and across different Web application servers. Servlets can respond to client requests by dynamically constructing a response that is sent back to the client. Because servlets are written in the Java programming language, they have access to the full set of Java APIs. This makes them ideal for implementing complex business application logic and especially for accessing data elsewhere in the enterprise. The Java Database Connectivity (JDBC) API, which allows Java programs to access relational databases is one example.

A single servlet can be invoked multiple times to serve requests from multiple clients. A servlet can handle multiple requests concurrently and can synchronize requests. Servlets can forward requests to other servers and servlets.

How does a servlet work?

A servlet runs inside a Java-enabled application server. Application servers are special kind of Web servers; they extend the capabilities of a Web server to handle requests for servlets, enterprise beans, and Web applications. There is a distinct difference between a Web server and an application server. While both can run in the same machine, the Web server runs client code such as applets and the application server runs the servlet code. The

server itself loads, executes, and manages servlets. The server uses a Java bytecode interpreter to run Java programs; this is called the Java Virtual Machine (JVM).



The basic flow is this:

1. The client sends a request to the server.
2. The server instantiates (loads) the servlet and creates a thread for the servlets process. The servlet is loaded upon the first request; it stays loaded until the server shuts down.
3. The server sends the request information to the servlet.
4. The servlet builds a response and passes it to the server.
5. The server sends the response back to the client.

The servlet dynamically contructs the response using information from the client request, plus data gathered from other sources if needed. Such sources could be other servlets, shared objects, resource files, and databases. Shared resources, for example, could include in-memory data such as instance or class variables and external objects such as files, database connections, and network connections.

How do you run a servlet?

Servlets are either invoked as an explicit URL reference or are embedded in HTML and invoked from a Web application. You don't execute a Java command to run a servlets; instead, you issue a URL command pointing to the location of the servlet.

Servlets are located in any directory on a machine where an application server is running. Typically, there is a servlet directory to which you will ftp or copy your class files

so the application server can find them. This location can be configured differently, depending on the administrator and the environment.

Servlets are loaded when they are invoked for the first time from a client. Alternatively, they can be loaded when the application server is started; this is a configuration decision. The server must support the version level at which you've written your Java servlets. Most servers support servlets written based on the Java Servlet Development Kit (JSDK) Version 2.1.

How are servlets different from CGI programs?

Common gateway interface (CGI) programs are also used to construct dynamic Web content in response to a request. But servlets have several advantages over CGI. Servlets provide a component-based, platform-independent method for building Web applications, without the performance limitations of CGI programs.

Servlets are:

- **Portable across platforms and across different Web servers.** Servlets enable you to do server-side programming without writing to platform-specific APIs; the Java Servlet API is a standard Java extension.
- **Persistent.** A servlet remains in memory once loaded, which means it can maintain system resources -- such as a database connection – between requests.
- **Efficient.** When a client makes multiple calls to a servlet, the server creates and loads the servlet only once. Each additional call performs only the business logic processing. CGI processes are loaded with each request, which slows performance. In addition, the JVM uses lightweight Java threads to handle servlet requests, rather than a weighty operating system process used by CGI.

- **Able to separate presentation from business logic.** This makes it easier to split a project into distinct parts for easier development and maintenance.
- **Able to access a library of HTTP-specific calls** and to benefit from the continuing development of the Java language itself.

The Java Servlet API

What is the Java Servlet API?

The Java Servlet API is a set of classes that define a standard interface between a Web client and a Web servlet. In essence, the API encases requests as objects so the server can pass them to the servlet; the responses are similarly encapsulated so the server can pass them back to a client.

The Java Servlet API has two packages. javax.servlet contains classes to support generic protocol-independent servlets, and javax.servlet.http includes specific support for the HTTP protocol.

The architecture of the servlet API is that of a classic service provider with a **service( )** method through which all client requests will be sent by the servlet container software, and life cycle methods **init( )** and **destroy( )**, which are called only when the servlet is loaded and unloaded.

```
public interface Servlet {
  public void init(ServletConfig config) throws ServletException;
  public ServletConfig getServletConfig();
  public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException;
  public String getServletInfo();
  public void destroy();
}
```

getServletConfig( )'s sole purpose is to return a **ServletConfig** object that contains initialization and startup parameters for this servlet. **getServletInfo( )** returns a string containing information about the servlet, such as author, version, and copyright.

The **GenericServlet** class is a shell implementation of this interface and is typically not used. The **HttpServlet** class is an extension of **GenericServlet** and is designed specifically to handle the HTTP protocol— **HttpServlet** is the one that you'll use most of the time.

The most convenient attribute of the servlet API is the auxiliary objects that come along with the HttpServlet class to support it. If you look at the **service( )** method in the **Servlet** interface, you'll see it has two parameters: **ServletRequest** and **ServletResponse**. With the **HttpServlet** class these two object are extended for HTTP: **HttpServletRequest** and **HttpServletResponse**.

On the Web: HTTP servlets

The Servlet interface class is the central abstraction of the Java Servlet API. This class defines the methods that manage the servlet and its communications with clients.

To write an HTTP servlet for use on the Web, use the HttpServlet class.

- A client request to a servlet is represented by an HttpServletRequest object. This object encapsulates the communication from the client to the server. It can contain information about the client environment and any data to be sent to the servlet from the client.
- The response from the servlet back to the client is represented by an HttpServletResponse object. This is often the dynamically generated response, such as an HTML page, and is built with data from the request and from other sources accessed by the servlet.

Key methods for processing HTTP servlets

Your subclass of HttpServlet must override at least one method. Typically, servlets override the doGet or doPost method. GET requests are typical browser requests for Web pages, issued when a user types a URL or follows a link. POST requests are generated when a user submits an HTML form that specifies post. The HTTP POST method allows the client to send data of unlimited length to the Web server a single time and is useful when posting information such as credit card numbers. The same servlet can handle both GET and POST by having doGet call doPost, or vice versa.

Other commonly used methods include:

* service, the lowest common denominator method for implementing a servlet; most people use doGet or doPost
* doPut, for HTTP PUT requests
* doDelete, for HTTP DELETE requests
* init and destroy, to manage resources that are held for the life of the servlet
* getServletInfo, which the servlet uses to provide information about itself

The role of the server

A home for servlets

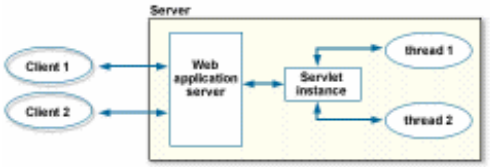
Servlet code follows the standard API, but the servers in which the code runs vary from free servlet engines to commercial servers. If you're just learning servlets, it may be useful to download a free server or servlet engine (eg JSDK 2.0) for your development and testing purposes. A servlet engine runs within the application server and manages the servlet. It loads and initializes the servlet, passes requests and responses to the servlet and client, manages multiple instances of a servlet, and acts as a request dispatcher. It destroys the servlet at the end of its useful life. Servlets are unloaded and removed from memory when the server shuts down.

Loading and initialization

A server loads and instantiates a servlet dynamically when its services are first requested. You can also configure the Web server to load and instantiate specific servlets when the Web server is initialized. The init method of the servlet performs the servlet initialization. It is called once for each servlet instance, before any requests are handled. Example tasks performed in the init method include loading default data or database connections.

Handling multithreading

When there are simultaneous requests to a servlet, the server handles each client request by creating a new thread for each request. There is a single thread per client request; this thread is used again and again each time the same client makes a request to the servlet.



For example:

- A servlet is loaded by the Web application server. There is one instance of a servlet object at a time, and it remains persistent for the life of the servlet.
- Two browser clients request the services of the servlet. The server creates a handler thread, for each request, against the instance object. Each thread has access to variables that were initialized when the servlet was loaded.
- Each thread handles its own requests. The server sends responses back to the appropriate client.

Request/response handling

The request/response handling done inside the servlet (with the HttpServletRequest and HttpServletResponse objects) is different from the request handling done by the Web server. The server handles communication with the external entities such as the client by passing the input, then the output, to the servlet.

A basic HTTP servlet

Scenario: Build a servlet that writes to a browser

In this example, we'll write a very basic servlet to print an HTML page. The servlet calls a standard Java utility to print the date and local time. This servlet has little real function beyond simply demonstrating the functions of a Servlet.

Add import statements

The import statements give us access to other Java packages. Specifically, the first provides access to standard input output (IO) classes, and the second and third to the Java Servlet API set of classes and interfaces.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

Declare a new class

The next statement extends the HttpServlet class (javax.servlet.http.HttpServlet) to make our new class, called SimplePrint, an HTTP protocol servlet. We then create a string that names the servlet.

```
public class SimplePrint extends HttpServlet
{
public static final String TITLE = "Test servlet";
```

Handle HTTP requests

The service method handles the HTTP requests; in essence, it allows the servlet to respond to a request. We can also use the more specific doGet and doPost methods in place of the service method.

```
public void service(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
```

Set up communication between server and client

Next we set the content type (text/html is the most common), get the communication channel with the requesting client for the response, and get the server identification.

```
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String server = getServletConfig().
getServletContext().getServerInfo();
```

Write the data

Lastly, we write the data. We indicate the server is working and invoke a standard Java utility to display the local time and date.

```
out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML
>"
+ "<HTML>"
+ "<HEAD>"
+ " <TITLE>" + TITLE + "</TITLE>"
+ " <META NAME=\"Author\" CONTENT=\"" + server + "\">"
+ "</HEAD>"
+ "<BODY BGCOLOR=\"#FFFFFF\">"
+ " <CENTER>"
```

```
+ " <H1>" + TITLE + "</H1>"
+ " <H2>Whoo Hoo, " + server + " is working!</H2>"
+ " <H3>[ local time is <font color='#FF9900'>"
+ new java.util.Date() + "</font> ]</H3>"
+ " </CENTER>"
+ "</BODY>"
+ "</HTML>");
}
}
```

Here is the entire source code for SimplePrint.java.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
* This servlet is used to print a simple HTML page.
*/
public class SimplePrint extends HttpServlet
{
public static final String TITLE = "Test servlet";
/**
* This function handles HTTP requests
*/
public void service (HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
// set content type and other response header fields first
response.setContentType("text/html");
// get the communication channel with the requesting client
PrintWriter out = response.getWriter();
// get the server identification
String server = getServletConfig().
```

```
getServletContext().
getServerInfo();
// write the data
out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML
>"
+ "<HTML>"
+ "<HEAD>"
+ " <TITLE>" + TITLE + "</TITLE>"
+ " <META NAME=\"Author\" CONTENT=\"" + server + "\">"
+ "</HEAD>"
+ "<BODY BGCOLOR=\"#FFFFFF\">"
+ " <CENTER>"
+ " <H1>" + TITLE + "</H1>"
+ " <H2>Whoo Hoo, " + server + " is working!</H2>"
+ " <H3>[ local time is <font color='#FF9900'>"
+ new java.util.Date() + "</font> ]</H3>"
+ " </CENTER>"
+ "</BODY>"
+ "</HTML>");
}
}
```

Compile the code and put the class file in the servlet directory. Run the servlet using a URL with the following syntax. Be sure to remember that the class file name is case sensitive.

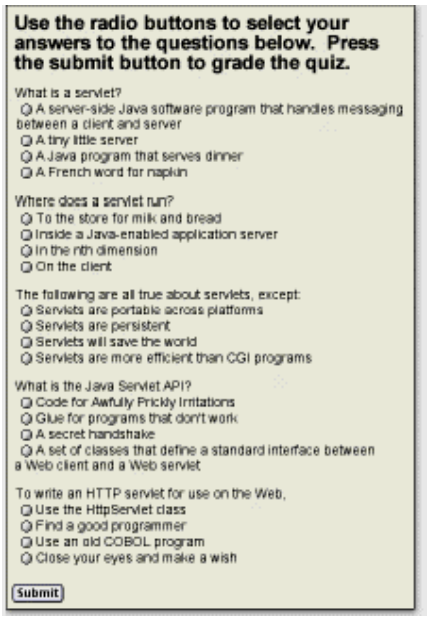
http://localhost:8080/servlet/SimplePrint

An HTTP servlet that processes data

Scenario: Build an HTML form that processes data

In this example, we'll write a servlet to generate an HTML form. Quiz is a simple servlet that handles a GET request submitted from a form in a Web browser. The servlet

processes the form data, grades the answers, and displays the results. The image below shows the HTML page this servlet generates.



Add import statements and class declaration

First add the import statements that give us access to other Java packages and declare a new class called Quiz.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Quiz extends HttpServlet
{
```

Add strings for questions and answers

These declarations contain the data for questions and answers. In this example, the code stipulates five questions, each with four possible answers. We also define the set of valid answers to these questions. The userAnswers array will contain the user responses. The submit string is initially set to null, which means user has not yet submitted the form (and we don't yet calculate the grade).

```
String [] questions =
{"What is a servlet?",
"Where does a servlet run?",
"The following are all true about servlets, except:",
"What is the Java Servlet API?",
"To write an HTTP servlet for use on the Web,"};
String [] [] possibleAnswers =
{
{ "A server-side Java software program that handles\
```

```
messaging between a client and server",
"A tiny little server",
"A Java program that serves dinner",
"A French word for napkin" },
{ "To the store for milk and bread",
"Inside a Java-enabled application server",
"In the nth dimension",
"On the client" },
{ "Servlets are portable across platforms",
"Servlets are persistent",
"Servlets will save the world",
"Servlets are more efficient than CGI programs" },
{ "Code for Awfully Prickly Irritations",
"Glue for programs that don't work",
"A secret handshake",
"A set of classes that define a standard interface between\
a Web client and a Web servlet" },
{ "Use the HttpServlet class",
"Find a good programmer",
"Use an old COBOL program",
"Close your eyes and make a wish" }
};
int [] answers = {1,2,3,4,1};
int [] userAnswers = new int[5];
String submit = null;
```

Implement the doGet and doPost methods

doGet and doPost are the servlet's service methods. doGet is called by the server to allow a servlet to handle a GET request; doPost to handle a POST request. The service method receives standard HTTP requests from the public service method and dispatches them to the doXXX methods defined in this class.

The code for the doGet method reads the request data and gets the answers submitted by the user. Because the parameters are strings, and we are using ints, we need to call the static parseInt method and catch the appropriate exception. The code also checks to make sure the quiz was submitted.

The getWriter method on the response object enables us to send character text to the client. In this servlet, doPost calls doGet. This is to ensure that if some program issues a POST request to this servlet (which could alter data), the servlet will enact the doGet method in response.

```
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
PrintWriter out = response.getWriter();
printHeader(out, response);
for (int k = 0; k < answers.length; k++)
{
try
{
userAnswers[k] = (Integer.parseInt
(request.getParameter("answer"+k)));
}
catch (NumberFormatException nfe)
{
userAnswers[k] = 0;
}
}
submit = request.getParameter("submit");
printQuiz(out);
printClosing(out);
}
```

```
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
this.doGet(request, response);
}
```

Print the HTML form

The rest of the code executes the functions for printing the form. First, print the header. In HTTP, headers must be sent before the response body. And, you need to set the content type before accessing the PrintWriter object. This tells the browser what type of data we're sending.

Next, print the quiz form. This code uses the FORM function to print each question. It also checks to see whether the quiz was submitted (so it knows whether to grade it).

```
public void printHeader(PrintWriter output,
HttpServletResponse resp)
{
resp.setContentType("text/html");
output.println("<HTML>\n" +
"<HEAD>\n" +
"<TITLE>Servlet Quiz</TITLE>\n" +
"</HEAD>\n" +
"<BODY BGCOLOR=\"#ffffff\">\n" +
"<H2>");
}
public void printQuiz(PrintWriter output)
{
output.println("Use the radio buttons to select your " +
"answers to the\nquestions below. Press " +
"the submit button to grade the quiz.\n" +
```

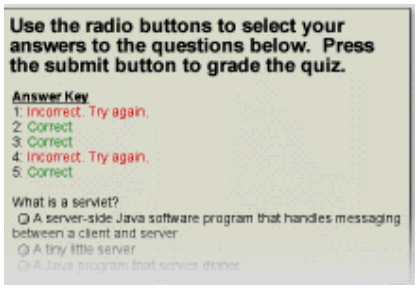
```
"</H2><FORM METHOD=\"GET\" ACTION=\"./Quiz\">");
if (submit != null)
{
grade(output);
}
```

Then, print each question with its possible answers.

```
for (int i = 0; i < questions.length; i++)
{
printQuestion(output, i);
}
output.println("<INPUT TYPE=\"submit\" NAME=\"submit\">\n" +
"</FORM>");
}
public void printClosing(PrintWriter output)
{
output.println("</BODY>\n" +
"</HTML>");
}
public void printQuestion(PrintWriter out, int questionIndex)
{
out.println("<P>" + questions[questionIndex] + "<BR>\n");
for (int j = 0; j < possibleAnswers[questionIndex].length; j++)
{
out.println("<INPUT TYPE=\"radio\" NAME=\"answer" +
questionIndex + "\" VALUE=\"" + (j+1) + "\">" +
possibleAnswers[questionIndex][j] +
"<BR>");
}
out.println("</P>");
}
```

Grade the quiz

Finally, grade the quiz once it is submitted by the user. The code to grade the quiz loops through the answers and compares them with userAnswers. Correct responses are printed in green; incorrect in red:




```
public void grade (PrintWriter out)
{
out.println("<P><B><U>Answer Key</U></B><BR>");
for (int p = 0; p < answers.length; p++)
{
out.println((p+1) + ":");
if (answers[p] == userAnswers[p])
{
out.println("<FONT");
}
else
{
out.println("<FONT COLOR=\"ff0000\">Incorrect. +
Try again.</FONT><BR>");
}
}
out.println("</P>");
}
}
```

Here's the total source code for Quiz.java .

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
/**
```

```
* Quiz is a simple servlet that handles a GET request submitted from a form in a web browser. Quiz grades the
form results and displays the results for the user.
*/
public class Quiz extends HttpServlet
{
String [] questions =
{"What is a servlet?",
"Where does a servlet run?",
"The following are all true about servlets, except:",
"What is the Java Servlet API?",
"To write an HTTP servlet for use on the Web,"};
String [] [] possibleAnswers =
{
{ "A server-side Java software program that handles\
messaging between a client and server",
"A tiny little server",
"A Java program that serves dinner",
"A French word for napkin" },
{ "To the store for milk and bread",
"Inside a Java-enabled application server",
"In the nth dimension",
"On the client" },
{ "Servlets are portable across platforms",
"Servlets are persistent",
"Servlets will save the world",
"Servlets are more efficient than CGI programs" },
{ "Code for Awfully Prickly Irritations",
"Glue for programs that don't work",
"A secret handshake",
"A set of classes that define a standard interface between\
a Web client and a Web servlet" },
{ "Use the HttpServlet class",
"Find a good programmer",
```

```
"Use an old COBOL program",
"Close your eyes and make a wish" }
};
int [] answers = {1,2,3,4,1};
int [] userAnswers = new int[5];
String submit = null;
/**
* This method handles the GET request from the web browser.
*/
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
PrintWriter out = response.getWriter();
printHeader(out, response);
for (int k = 0; k < answers.length; k++)
{
try
{
// Get answers from the form submitted by the user
userAnswers[k] = (Integer.parseInt(request.
getParameter("answer"+k)));
}
catch (NumberFormatException nfe)
{
userAnswers[k] = 0;
}
}
// Checking to see if quiz was submitted.
submit = request.getParameter("submit");
printQuiz(out);
printClosing(out);
}
```

```
/**
* This method passes any POST requests to the doGet method.
*/
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
this.doGet(request, response);
}
/**
* Print the top portion of the HTML page.
* No dynamic content here.
*/
public void printHeader(PrintWriter output,
HttpServletResponse resp)
{
//Tell the browser what type of data we are sending.
resp.setContentType("text/html");
output.println("<HTML>\n" +
"<HEAD>\n" +
"<TITLE>Servlet Quiz</TITLE>\n" +
"</HEAD>\n" +
"<BODY BGCOLOR=\"#ffffff\">\n" +
"<H2>");
}
/**
* Prints the form for the Quiz.
*/
public void printQuiz(PrintWriter output)
{
output.println("Use the radio buttons to select " +
"your answers to the\nquestions below. " +
"Press the submit button to grade " +
```

```
"the quiz.\n" +
"</H2><FORM METHOD=\"GET\" ACTION=\"./Quiz\">");
// Check to see if a quiz was submitted.
// If so, grade the quiz.
if (submit != null)
{
grade(output);
}
// Print all the questions
for (int i = 0; i < questions.length; i++)
{
printQuestion(output, i);
}
output.println("<INPUT TYPE=\"submit\" NAME=\"submit\">\n" +
"</FORM>");
}
/**
* Closes the HTML page.
*/
public void printClosing(PrintWriter output)
{
output.println("</BODY>\n" +
"</HTML>");
}
/**
* Prints questions from the question list.
*/
public void printQuestion(PrintWriter out,
int questionIndex)
{
out.println("<P>" + questions[questionIndex] + "<BR>\n");
for (int j = 0; j < possibleAnswers[questionIndex].length; j++)
{
```

```
out.println("<INPUT TYPE=\"radio\" NAME=\"answer" +
questionIndex +
"\" VALUE=\"" + (j+1) + "\">" +
possibleAnswers[questionIndex][j] +
"<BR>");
}
out.println("</P>");
}
/**
* Grades the quiz submitted by the user.
*/
public void grade (PrintWriter out)
{
out.println("<P><B><U>Answer Key</U></B><BR>");
for (int p = 0; p < answers.length; p++)
{
out.println((p+1) + ":");
if (answers[p] == userAnswers[p])
{
out.println("<FONT
");
}
else
{
out.println("<FONT COLOR=\"ff0000\">Incorrect. Try
</FONT><BR>");
}
}
}
}
```

Session Tracking

Why do you need session tracking?

Internet communication protocols consist of two types: stateful and stateless. The server associates a state with a connection; it remembers who the user is and handles all user requests.

- Stateful protocols, such as telnet and FTP, can process multiple operations before closing a connection. The server knows that all requests came from a single person.
- HTTP is a stateless protocol, which means every time a client asks for a Web page, the protocol opens a separate connection to the server, and the server doesn't know the context from one connection to the next. Each transaction is a single isolated transaction. Imagine if every time you added an item to your shopping cart, it forgot all your previous items, and couldn't even remember whose cart it was in the first place.

The session tracking API

Techniques for identifying a session and associating data with it, even over multiple connections, include using cookies, URL rewriting, and hidden form fields. These techniques do require additional work for managing the information involved.

To eliminate the need for manually managing the session information within your code (no matter what session tracking technique you're using), you use the HttpSession class of the Java Servlet API. The HttpSession interface allows servlets to:

- View and manage information about a session
- Make sure information persists across multiple user connections, to include multiple page requests as well as connections

Session tracking techniques: Cookies

A cookie is nothing more than a small piece of information sent by a Web server to a browser. The browser stores the cookie on the local disk, and whenever another call is made to the URL that the cookie is associated with, the cookie is quietly sent along with the call, thus providing the desired information back to that server (generally, providing some way that the server can be told that it’s you calling). Clients can, however, turn off the browser’s ability to accept cookies. If your site must track a client who has turned off cookies, then another method of session tracking (URL rewriting or hidden form fields) must be incorporated by hand, since the session tracking capabilities built into the servlet API are designed around cookies.

Cookies are probably the most common approach for session tracking. Cookies store information about a session in a human-readable file on the client's machine. Subsequent sessions can access the cookie to extract information. The server associates a session ID from the cookie with the data from that session. This gets more complicated when there are multiple cookies involved, when you have to decide when to expire the cookie, and when you have to generate so many unique session identifiers. Additionally, a cookie cannot grow more than 4K in size, and no domain can have more than 20 cookies.

Cookies pose some privacy concerns for users. Some people don't like the fact that a program can store and retrieve information about their habits. In addition, people are wary that sensitive information -- such as a credit card number -- could be stored in a cookie. Unfortunately, it is easy to use cookies inappropriately. As a result, some users disable cookies or delete them altogether. Therefore, you should not depend on them as your sole mechanism for session tracking.

The Cookie class

The servlet API provides the **Cookie** class. This class incorporates all the HTTP header details and allows the setting of various cookie attributes. Using the cookie is simply a matter of adding it to the response object. The constructor takes a cookie name as the first

argument and a value as the second. Cookies are added to the response object before you send any content.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(oreo);
```

Cookies are recovered by calling the **getCookies( )** method of the **HttpServletRequest** object, which returns an array of cookie objects.

```
Cookie[] cookies = req.getCookies();
```

You can then call **getValue( )** for each cookie, to produce a **String** containing the cookie contents. In the above example, **getValue("TIJava")** will produce a **String** containing “2000.”

Session tracking techniques: URL rewriting

URL rewriting is used to append data on the end of each URL that identifies the session. The server associates the identifier with data it has stored about the session. The URL is constructed using an HTTP GET. It may include a query string containing pairs of parameters and values. For example:

```
http://www.server.com/getPreferences?uid=username&bgcolor=red&fgcolor=blue.
```

URLs can get quite lengthy.

URL rewriting is a good solution, especially if users have disabled cookies, in which case it becomes a most excellent solution. But be sure to consider the following:

- You have to be sure to append the information to every URL that references your site.
- Appending parameters brings up privacy issues; you may not want the actual data you

are tracking to be visible.

- There's a loophole with this technique: users can leave the session and come back using a bookmark, in which case your session information is lost.

These problems are not insurmountable, just tedious to tackle.

Session tracking techniques: Hidden form fields in HTML

Hidden form fields store information about the session. The hidden data can be retrieved later by using the HTTPServletRequest object. When a form is submitted, the data is included in the Get or Post.

But the form fields can be used only on dynamically generated pages, so their use is limited. And there are security holes: people can view the HTML source to see the stored data.

The HttpSession object

No matter which technique(s) you've used to collect session data, you need to store it somewhere. In this section, we use the HttpSession object to store the session data from , the servlet. This object matches a user to session data stored on a server. The basic steps for using the HttpSession object are:

- Obtain a session object
- Read or write to it
- Either terminate the session by expiring it, or do nothing so it will expire on its own

A session persists for a certain time period, up to forever, depending on the value set in the servlet. A unique session ID is used to track multiple requests from the same client to the server.

Persistence is valid within the context of the Web application, which may be across multiple servlets. A servlet can access an object stored by another servlet; the object is distinguished by name and is considered bound to the session. These objects (called attributes when you set and get them) are available to other servlets within the scope of a request, a session, or an application.

Key methods used in HttpSession

HttpSession maintains and accesses the session information within a servlet so you don't have to manipulate the information directly in your code.

Important methods used in HttpSession include:

- isNew(). Returns true if the client doesn't yet know about the session. If the client has disabled cookies, then a session is new on each request.
- getId(). Returns a string containing the unique identifier assigned to this session. Useful when using URL rewriting to identify the session.
- setAttribute(). Binds an object to this session, using the name specified. (Note: this replaces the setValue() method of JSDK 2.1.)
- getAttribute(). Returns the object (with the specified name) bound in this session. (Note: this replaces the getValue() method of JSDK 2.1.)
- setMaxInactiveInterval(). Specifies the time between client requests before the servlet invalidates this session. A negative time indicates the session should never timeout.
- invalidate(). Expires the current session and unbinds the object bound to it.

Here’s an example that implements session tracking with the servlet API:

```
//servlets:SessionPeek.java
// Using the HttpSession class.
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
  public void service(HttpServletRequest req,
  HttpServletResponse res)
  throws ServletException, IOException {
    // Retrieve Session Object before any
    // output is sent to the client.
    HttpSession session = req.getSession();
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("<HEAD><TITLE> SessionPeek ");
    out.println(" </TITLE></HEAD><BODY>");
    out.println("<h1> SessionPeek </h1>");
    // A simple hit counter for this session.
    Integer ival = (Integer)
      session.getAttribute("sesspeek.cntr");
    if(ival==null)
      ival = new Integer(1);
    else
      ival = new Integer(ival.intValue() + 1);
    session.setAttribute("sesspeek.cntr", ival);
    out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
    out.println("<h2>");
    out.println("Saved Session Data </h2>");
    // Loop through all data in the session:
```

```
Enumeration sesNames =
  session.getAttributeNames();
while(sesNames.hasMoreElements()) {
  String name =
    sesNames.nextElement().toString();
  Object value = session.getAttribute(name);
  out.println(name + " = " + value + "<br>");
}
out.println("<h3> Session Statistics </h3>");
out.println("Session ID: "
  + session.getId() + "<br>");
out.println("New Session: " + session.isNew()
  + "<br>");
out.println("Creation Time: "
  + session.getCreationTime());
out.println("<I>(" +
  new Date(session.getCreationTime())
  + ")</I><br>");
out.println("Last Accessed Time: " +
  session.getLastAccessedTime());
out.println("<I>(" +
  new Date(session.getLastAccessedTime())
  + ")</I><br>");
out.println("Session Inactive Interval: "
  + session.getMaxInactiveInterval());
out.println("Session ID in Request: "
  + req.getRequestedSessionId() + "<br>");
out.println("Is session id from Cookie: "
  + req.isRequestedSessionIdFromCookie()
  + "<br>");
out.println("Is session id from URL: "
  + req.isRequestedSessionIdFromURL()
  + "<br>");
```

```
    out.println("Is session id valid: "
      + req.isRequestedSessionIdValid()
      + "<br>");
    out.println("</BODY>");
    out.close();
  }
  public String getServletInfo() {
    return "A session tracking servlet";
  }
}
```

Inside the **service( )** method, **getSession( )** is called for the request object, which returns the **Session** object associated with this request. The **Session** object does not travel across the network, but instead it lives on the server and is associated with a client and its requests.

getSession( ) comes in two versions: no parameter, as used here, and **getSession(boolean)**. **getSession(true)** is equivalent to **getSession( )**. The only reason for the **boolean** is to state whether you want the session object created if it is not found. **getSession(true)** is the most likely call, hence **getSession( )**.

The **Session** object, if it is not new, will give us details about the client from previous visits. If the **Session** object is new then the program will start to gather information about this client's activities on this visit. Capturing this client information is done through the **setAttribute( )** and **getAttribute( )** methods of the session object.

```
java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name, java.lang.Object value)
```

The **Session** object uses a simple name-value pairing for loading information. The name is a **String**, and the value can be any object derived from **java.lang.Object**. **SessionPeek** keeps track of how many times the client has been back during this session.

This is done with an **Integer** object named **sesspeek.cntr**. If the name is not found an **Integer** is created with value of one, otherwise an **Integer** is created with the incremented value of the previously held **Integer**. The new **Integer** is placed into the **Session** object. If you use same key in a **setAttribute( )** call, then the new object overwrites the old one. The incremented counter is used to display the number of times that the client has visited during this session.

getAttributeNames( ) is related to **getAttribute( )** and **setAttribute( )**; it returns an enumeration of the names of the objects that are bound to the **Session** object. A **while** loop in **SessionPeek** shows this method in action.

You may wonder how long a **Session** object hangs around. The answer depends on the servlet container you are using; they usually default to 30 minutes (1800 seconds), which is what you should see from the **ServletPeek** call to **getMaxInactiveInterval( )**. Tests seem to produce mixed results between servlet containers. Sometimes the **Session** object can hang around overnight. You can try this by setting the inactive interval with **setMaxInactiveInterval( )** to 5 seconds and see if your **Session** object hangs around or if it is cleaned up at the appropriate time. This may be an attribute you will want to investigate while choosing a servlet container

3.7 SUMMARY

Java Networking classes provide an high level interface to the programmer, for doing network programming. All the programmer has to do is to use the APIs provided by Java without bothering about how they are implemented actually. RMI provides a way for distributed computing using the Java platform. RMI defines ans supports the distributed object model necessary for this. JDBC is an interface for connecting Java programs with SQL-based databases. Servlets are pieces of Java source code that add functionality to a web server in a manner similar to the way applets add functionality to a browser. From the Java

Servlet Development Kit (JSDK), you use the Java Servlet API to create servlets for responding to requests from clients.

3.8 SELF TEST

1. TCP sockets are also known as stream sockets.

a) True

b) Flase

2. The accept() method returns a n object of type Socket

a) True

b) Flase

3. What interface must all interfaces for remote services extend?

a) java.rmi.RemoteService

b) java.rmi.Remotable

c) java.rmi.server.Remote

d) java.rmi.Remote

4. When defining a remote interface, what exception is typically thrown as part of each method signature?

a) IOException

b) RemoteException

c) RemoteCollisionException

d) RemoteControlException

5. How are remote object parameters transferred between client and server?

a) A proprietary protocol that is determined by the vendor of the RMI implementation

b) Standard Java Serialization

c) Internet Inter-ORB Protocol (IIOP)

d) Java Remote Method Protocol (JRMP)

6. Which of the following will not cause a JDBC driver to be loaded and registered with the DriverManager?

a) Class.forName(driverString);

b) new DriverClass();

c) Include driver name in jdbc.drivers system property

d) None of the above

7. SQLWarnings from multiple Statement method calls (like executeUpdate) will build up until you ask for them all with getWarnings and getNextWarning.

a) True

b) False

8. If one intends to work with a ResultSet, which of these PreparedStatement methods will not work?

a) execute()

b) executeQuery()

c) executeUpdate()

9. Servlets can respond to client requests by dynamically constructing a response

a) True

b) False

10. object that contains initialization and startup parameters for a servlet.

a) ServletRequest

b) ServletResponse

c) ServletConfig

Answers

1. a
2. a
3. d
4. b
5. b
6. d
7. b
8. c
9. a
10. c

3.9 QUESTIONS

1. Write a skelatal program to retrieve a document from a webserver using URLConnection class.
2. Explain the working of a Java RMI system.
3. Write a TCP socket server program that echoes what ever it listens from the client.
4. Explain the steps involved in connecting to a database from a Java program.
5. Write a simple JDBC program to insert some rows in to a database.
6. What is the role of server in servlet programming?
7. Write a simple servlet that sends an HTML page to the client.
8. What is session tracking? What are the different techniques used inn session tracking.