

CMR ENGINEERING COLLEGE

Kandlakoya (V), Medchal Road, Hyderabad – 501 401

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OBJECT ORIENTED PROGRAMMING

Course File

(2017-2018)

(II B.Tech – I Semester)

Prepared By

Mr. S V RAMANA

Assistant Professor

Department of CSE



C.M.R. ENGINEERING COLLEGE

Kandlakoya (V), Medchal (M), Ranga Reddy (D),
Hyderabad- 501401. Andhra Pradesh
Phones: 08418-200037
Web: www.cmrec.ac.in

Syllabus**JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY****HYDERABAD****II Year B.Tech I-Sem****T P C****4+1* 0 4****OBJECT ORIENTED PROGRAMMING****UNIT I:**

Object-oriented thinking- A way of viewing world – Agents and Communities, messages and methods, Responsibilities, Classes and Instances, Class Hierarchies- Inheritance, Method binding, Overriding and Exceptions, Summary of Object-Oriented concepts. Java buzzwords, An Overview of Java, Data types, Variables and Arrays, operators, expressions, control statements, Introducing classes, Methods and Classes, String handling.

Inheritance– Inheritance concept, Inheritance basics, Member access, Constructors, Creating Multilevel hierarchy, super uses, using final with inheritance, Polymorphism-ad hoc polymorphism, pure polymorphism, method overriding, abstract classes, Object class, forms of inheritance- specialization, specification, construction, extension, limitation, combination, benefits of inheritance, costs of inheritance.

UNIT II:

Packages- Defining a Package, CLASSPATH, Access protection, importing packages.

Interfaces- defining an interface, implementing interfaces, Nested interfaces, applying interfaces, variables in interfaces and extending interfaces.

Stream based I/O(java.io) – The Stream classes-Byte streams and Character streams, Reading console Input and Writing Console Output, File class, Reading and writing Files, Random access file operations, The Console class, Serialization, Enumerations, auto boxing, generics.

UNIT III :

Exception handling - Fundamentals of exception handling, Exception types, Termination or resumptive models, Uncaught exceptions, using try and catch, multiple catch clauses, nested try statements, throw, throws and finally, built- in exceptions, creating own exception sub classes.

Multithreading- Differences between thread-based multitasking and process-based multitasking, Java thread model, creating threads, thread priorities, synchronizing threads, inter thread communication.

UNIT IV:

The Collections Framework (java.util)- Collections overview, Collection Interfaces, The Collection classes- Array List, Linked List, Hash Set, Tree Set, Priority Queue, Array Deque. Accessing a Collection via an Iterator, Using an Iterator, The For-Each alternative, Map Interfaces and Classes, Comparators, Collection algorithms, Arrays, The Legacy Classes and Interfaces- Dictionary, hashtable, Properties, Stack, Vector More Utility classes, String Tokenizer, Bit Set, Date, Calendar, Random, Formatter, Scanner

UNIT V:

GUI Programming with Swing – Introduction, limitations of AWT, MVC architecture, components, containers. Understanding Layout Managers, Flow Layout, Border Layout, Grid Layout, Card Layout, Grid Bag Layout.

Event Handling- The Delegation event model- Events, Event sources, Event Listeners, Event classes, Handling mouse and keyboard events, Adapter classes, Inner classes, Anonymous Inner classes.

A Simple Swing Application

Applets – Applets and HTML, Security Issues, Applets and Applications, passing parameters to applets. Creating a Swing Applet, Painting in Swing, A Paint example, Exploring Swing Controls- JLabel and Image Icon, JText Field, The Swing Buttons- JButton, JToggleButton, JCheckBox, JRadioButton, JTabbedPane, JScrollPane, JList, JComboBox, Swing Menus, Dialogs.

TEXTBOOKS :

1. Java The complete reference, 9th edition, Herbert Schildt, McGraw Hill Education (India) Pvt. Ltd.
2. Understanding Object-Oriented Programming with Java, updated edition, T. Budd, Pearson Education.

REFERENCES :

1. An Introduction to programming and OO design using Java, J. Nino and F.A. Hosch, John Wiley & sons.
2. Introduction to Java programming, Y. Daniel Liang, Pearson Education.
3. Object Oriented Programming through Java, P. Radha Krishna, Universities Press.
4. Programming in Java, S. Malhotra, S. Chudhary, 2nd edition, Oxford Univ. Press.
5. Java Programming and Object oriented Application Development, R. A. Johnson, Cengage Learning.

11. Lecture Notes

11.1 UNIT - 1 Object oriented thinking

Need For OOP Paradigms:

- Object oriented programming is at the core of java.
- All Computer programs consist of two elements: code and data.
- Furthermore a program can be conceptually organized around its code or around its data.
- That is some programs are written around “What is happening” and others are written around “who is being affected”.
- These are the two paradigms that govern how a program is constructed.
- The first way is called the Process-Oriented Model.
- This approach characterizes a program as a series of linear steps.
- The process oriented model can be thought of code acting on data.
- Procedural languages like C employ this kind of paradigm.
- To manage increasing complexity, the second approach, called object oriented programming, was conceived.
- Object oriented programming organizes a program around its data.
- An object oriented program can be characterized as data controlling access to code.

1. A way of viewing world – Agents

- OOP uses an approach of treating a real world agent as an object.
- Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data.
- An object-oriented program can be characterized as data controlling access to code by switching the controlling entity to data.

2. Responsibility

- primary motivation is the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.
- Objects with clear responsibilities
- Each class should have a clear responsibility.
- If you can't state the purpose of a class in a single, clear sentence, then perhaps your class structure needs some thought.

3. Messages

- We all like to use programs that let us know what's going on. Programs that keep us informed often do so by displaying status and error messages.
- These messages need to be translated so they can be understood by end users around the world.

- The Section discusses translatable text messages. Usually, you're done after you move a message String into a ResourceBundle.
- If you've embedded variable data in a message, you'll have to take some extra steps to prepare it for translation.

Methods:

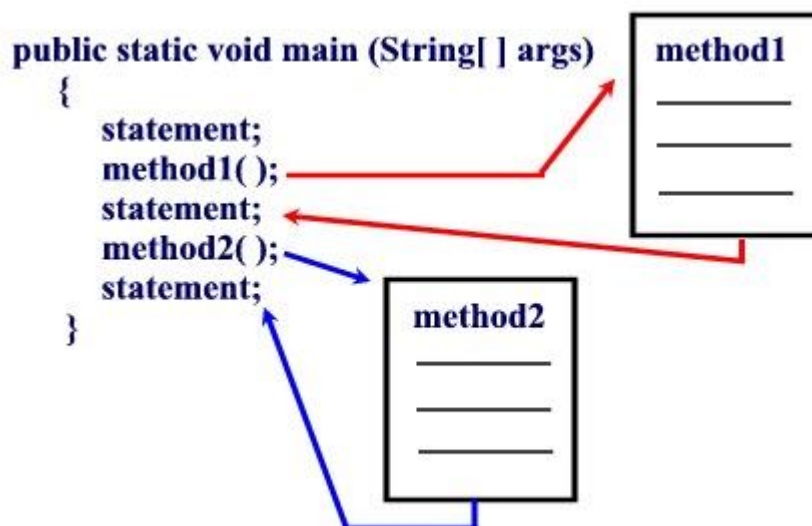
A method is a set of code which is referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name. Think of a method as a subprogram that acts on data and often returns a value.

Each method has its own name. When that name is encountered in a program, the execution of the program branches to the body of that method. When the method is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.

There are two basic types of methods:

Built-in: Build-in methods are part of the compiler package, such as `System.out.println ()` and `System.exit(0)`.

User-defined: User-defined methods are created by you, the programmer. These methods take on names that you assign to them and perform tasks that you create.



Classes:

A class is a blueprint or prototype from which objects are created. This section defines a class that models the state and behavior of a real-world object. It intentionally focuses on the basics, showing how even simple classes can cleanly model state and behavior. It is the place where we declare our members and methods.

Creating a class:

A class is created in the following way

```
Class <class name>
{
    Member variables;
    Methods;
}
```

An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life. This lesson explains how state and behavior are represented within an object, introduces the concept of data encapsulation, and explains the benefits of designing your software in this manner.

Bundling code into individual software objects provides a number of benefits, including:

1. **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.
2. **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.
3. **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.
4. **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace *it*, not the entire machine.

An instance or an object for a class is created in the following way

```
<class name> <object name>=new <constructor>();
```

Encapsulation:

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much we are accelerating, the pitch of the surface we are on, and the position of the shift

lever. We, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. We can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

Polymorphism:

Polymorphism (from the Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). We might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values,

and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, we would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java we can specify a general set of stack routines that all share the same names. More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action*. It is the compiler’s job to select the *specific action* (that is, method) as it applies to each situation. We, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface. Polymorphism allows us to create clean, sensible, readable, and resilient code.

Class Hierarchies (Inheritance):

Different kinds of objects often have a certain amount in common with each other. Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear). Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes. In this example, Bicycle now becomes the *super class* of Mountain Bike, Road Bike, and Tandem Bike. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:

The syntax for creating a subclass is simple. At the beginning of your class declaration, use the `extends` keyword, followed by the name of the class to inherit from:

```
class <sub class> extends <super class> {  
    // new fields and methods defining a sub class would go here  
}
```

The different types of inheritance are

1. Single level Inheritance.
2. Multilevel Inheritance.
3. Hierarchical inheritance.

There is one more type of inheritance called multiple inheritance but it is not used in the way as other inheritances but it needs a special concept called interfaces.

Method Binding:

- Binding denotes association of a name with a class.
- Static binding is a binding in which the class association is made during compile time. This is also called as early binding.
- Dynamic binding is a binding in which the class association is not made until the object is created at execution time. It is also called as late binding.

Abstraction:

- An essential component of object oriented programming is Abstraction

- Humans manage complexity through abstraction.
- For example people do not think a car as a set of tens and thousands of individual parts.
- They think of it as a well defined object with its own unique behavior.
- This abstraction allows people to use a car ignoring all details of how the engine, transmission and braking systems work.
- In computer programs the data from a traditional process oriented program can be transformed by abstraction into its component objects.
- A sequence of process steps can become a collection of messages between these objects.
- Thus each object describes its own behavior.

Overriding:

- In a class hierarchy when a sub class has the same name and type signature as a method in the super class, then the method in the subclass is said to override the method in the super class.
- When an overridden method is called from within a sub class, it will always refer to the version of that method defined by the sub class.
- The version of the method defined by the super class will be hidden.

Exceptions:

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words an exception is a run time error.
- A java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- Now the exception is caught and processed.

Java was conceived by James gosling, Patrick Naughton, chriswarth, Ed frank and Mike Sheridan at sun Microsystems.

The original impetus for java was not internet instead primary motivation was the need for a platform independent (i.e. Architectural neutral) independent language.

Java's Byte code:

The key that allows java to solve the both security and portability problems is that the output of a java compiler is not executable code rather it is byte code.

Byte code is highly optimized set of instructions designed to be executed by java runtime systems, which is called Java Virtual Machine (JVM). JVM is interpreter for byte code. Translating a java program into byte code helps makes it much easier to run a Program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it.

The Java Buzzwords:

No discussion of the genesis of Java is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

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

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++

Robust

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs were given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has

been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Java was engineered for interpretation, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. “High-performance cross-platform” is no longer an oxymoron.

Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intraaddress-space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote Method Invocation (RMI)*. This feature brings an unparalleled level of abstraction to client/server programming.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of byte code may be dynamically updated on a running system.

Security

Every time that you download a “normal” program, you are risking a viral infection. Even so, most users still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. Java answers both of these concerns by providing a “firewall” between a networked application and your computer. When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer.

Portability

Many types of computers and operating systems are in use throughout the world—and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Data Types:

Java defines eight simple types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **Boolean**. These can be put in four groups:

- Integers this group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.
- Floating-point numbers this group includes **float** and **double**, which represent numbers with fractional precision.
- Characters this group includes **char**, which represents symbols in a character set, like letters and numbers.
- Boolean this group includes **Boolean**, which is a special type for representing true/false values.

Integers:

The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Floating-point:

There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Variables:

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value][, identifier [= value] ...];
```

The *type* is one of Java's atomic types, or the name of a class or interface. The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c; // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

The Scope and Lifetime of Variables:

All of the variables used till now have been declared at the start of the **main()** method. However, Java allows variables to be declared within any block. A block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope*. Thus, each time you start a new block, you are creating Scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects. Most other computer languages define two general categories of scopes: global and local.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope. However, the reverse is not true. Objects declared within the inner scope will not be visible outside it.

To understand the effect of nested scopes, consider the following program:

```
// Demonstrate block scope.
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10;
if(x == 10) { // start new scope
int y = 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}
```

```
}
```

As the comments indicate, the variable **x** is declared at the start of **main()**'s scope and is accessible to all subsequent code within **main()**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope.

Here is another important point to remember: variables are created when their scope is entered and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered.

For example, consider the next program.

```
// Demonstrate lifetime of a variable.
class Lifetime {
public static void main(String args[]) {
    int x;
    for(x = 0; x < 3; x++) {
        int y = -1; // y is initialized each time block is entered
        System.out.println("y is: " + y); // this always prints -1
        y = 100;
        System.out.println("y is now: " + y);
    }
}
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is always reinitialized to **-1** each time the inner **for** loop is entered. Even though it is subsequently assigned the value **100**, this value is lost. One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope.

Arrays:

An *array* is a group of similar-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one dimensional array declaration is

```
type var-name[ ];
```

Here, *type* declares the base type of the array. For example, the following declares an array named **month** with the type “array of int”:

```
int month [];
```

Although this declaration establishes the fact that **month** is an array variable, no array actually exists. In fact, the value of **month** is set to **null**, which represents an array with no value. To link **month** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month**. **new** is a special operator that allocates memory. The general form of **new** as it applies to one-dimensional arrays appears as follows:

```
array-var = new type[size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero.

This example allocates a 12-element array of integers and links them to **month**

```
month = new int[12];
```

After this statement executes, **month** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Another way to declare an array in single step is

```
type arr-name=new type[size];
```

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

For example, to store the number of days in each month, we do as follows

```
// An improved version of the previous program.
```

```
class AutoArray
{
public static void main(String args[])
{
int month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
System.out.println("April has " + month[3] + " days.");
}
}
```

When you run this program, in the output it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month[3]** or 30.

Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers.

```
// Average an array of values.
```

```
class Average
{
public static void main(String args[])
{
double nums[] = { 10.1, 11.2, 12.3, 13.4, 14.5 };
double result = 0;
```

```

int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}

```

Output: Average is:12.3

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

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

This allocates a 4 by 5 array and assigns it to **twoD**.

program :

// Demonstrate a two-dimensional array.

```

class TwoDArray
{
public static void main(String args[])
{
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++)
{
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++)
{
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
}

```

Output:

```

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19

```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. We can allocate the second dimension manually.

```
int twoD[][] = new int[4][];
```

```
twoD[0] = new int[5];
```

```
twoD[1] = new int[5];
```

```
twoD[2] = new int[5];
```

```
twoD[3] = new int[5];
```

we can create a two dimensional array in which the sizes of the second dimension are unequal.

// Manually allocate differing size second dimensions.


```

class TwoDAgain
{
public static void main(String args[])
{
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++)
{
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++)
{
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}

```

Output:

```

0
1 2
3 4 5
6 7 8

```

We can create a three-dimensional array where first index specifies the number of tables, second one number of rows and the third number of columns.

// Demonstrate a three-dimensional array.

```

class threeDMatrix
{
public static void main(String args[])
{
int threeD[][][] = new int[3][4][5];
int i, j, k;
for(i=0; i<3; i++)
for(j=0; j<4; j++)
for(k=0; k<5; k++)
threeD[i][j][k] = i * j * k;
for(i=0; i<3; i++)
{
for(j=0; j<4; j++)
{
for(k=0; k<5; k++)
System.out.print(threeD[i][j][k] + " ");
System.out.println();
}
}
}

```



```

}
}
}

```

Output:

```

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

```

```

0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12

```

```

0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24

```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

`type[] var-name;`

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

`int al[] = new int[3];`

`int[] a2 = new int[3];`

The following declarations are also equivalent:

`char twod1[][] = new char[3][4];`

`char[][] twod2 = new char[3][4];`

Operators:

Operators are special symbols that perform specific operations on one, two, or three *operands*, and then return a result. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence	
Operators	Precedence
Postfix	EXPR++, EXPR--
Unary	++EXPR --EXPR +EXPR -EXPR ~ !

multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	
logical AND	&&
logical OR	
ternary	? :
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=

In general-purpose programming, certain operators tend to appear more frequently than theirs; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>".

Expressions:

An **EXPRESSION** is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the language that evaluates to a single value.

```
int a = 0;
arr[0] = 100;
System.out.println("Element 1 at index 0: " + arr[0]);
```

```
int result = 1 + 2; // result is now 3
if(value1 == value2)
    System.out.println("value1 == value2");
```

The data type of the value returned by an expression depends on the elements used in the expression. The expression `a = 0` returns an `int` because the assignment operator returns a value of the same data type as its left-hand operand; in this case, cadence is an `int`. As you can see from the other expressions, an expression can return other types of values as well, such as `boolean` or `String`.

For example, the following expression gives different results, depending on whether you perform the addition or the division operation first:

$x + y / 100$ // ambiguous

You can specify exactly how an expression will be evaluated using balanced parenthesis rewrite the expression as

$(x + y) / 100$ // unambiguous, recommended

If you don't explicitly indicate the order for the operations to be performed, the order is determined by the precedence assigned to the operators in use within the expression. Operators that have a higher precedence get evaluated first. For example, the division operator has a higher precedence than does the addition operator. Therefore, the following two statements are equivalent:

$x + y / 100$

$x + (y / 100)$ // unambiguous, recommended

When writing compound expressions, be explicit and indicate with parentheses which operators should be evaluated first.

Control Statements:

IF Statement

The general form of the **if** statement:

if (condition) statement1;

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a *block*). The *condition* is any expression that returns a **boolean** value.

If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* is executed.

IF –ELSE Statement

The general form of the **if** statement:

if (condition) statement1;

else statement2;

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false.

void applyBrakes()

```
{
    if (isMoving) {
        currentSpeed--;
    } else {
        System.err.println("The bicycle has already stopped!");
    }
}
```

```
}  
}
```

The switch Statement

Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types*.

Program: displays the name of the month, based on the value of month, using the switch statement.

```
class SwitchDemo  
{  
    public static void main(String[] args) {  
  
        int month = 8;  
        switch (month) {  
            case 1: System.out.println("January");  
                    break;  
            case 2: System.out.println("February");  
                    break;  
            case 3: System.out.println("March");  
                    break;  
            case 4: System.out.println("April");  
                    break;  
            case 5: System.out.println("May");  
                    break;  
            case 6: System.out.println("June");  
                    break;  
            case 7: System.out.println("July");  
                    break;  
            case 8: System.out.println("August");  
                    break;  
            case 9: System.out.println("September");  
                    break;  
            case 10: System.out.println("October");  
                    break;  
            case 11: System.out.println("November");  
                    break;  
            case 12: System.out.println("December");  
                    break;  
            default: System.out.println("Invalid month.");  
                    break;  
        }  
    }  
}
```

Output:"August"

The body of a switch statement is known as a *switch block*. Any statement immediately contained by the switch block may be labeled with one or more case or default labels. The switch statement evaluates its expression and executes the appropriate case.

The while and do-while Statements

The while statement continually executes a block of statements while a particular condition is true.

Its syntax can be expressed as:

```
while (expression) {  
    statement(s)  
}
```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the STATEMENT(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following program:

```
class WhileDemo  
{  
    public static void main(String[] args){  
        int count = 1;  
        while (count < 11) {  
            System.out.println("Count is: " + count);  
            count++;  
        }  
    }  
}
```

do-while statement

Its syntax can be expressed as:

```
do {  
    statement(s)  
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once.

Program:

```
class DoWhileDemo  
{  
    public static void main(String[] args){  
        int count = 1;  
        do {  
            System.out.println("Count is: " + count);  
            count++;  
        } while (count <= 11);  
    }  
}
```

```
}  
}
```

The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```
for (INITIALIZATION; TERMINATION; INCREMENT) {  
    STATEMENT(S)  
}
```

When using the for statement, we need to remember that

- The INITIALIZATION expression initializes the loop; it's executed once, as the loop begins.
- When the TERMINATION expression evaluates to false, the loop terminates.
- The INCREMENT expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

Type Conversion and Casting:

We can assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an int value to a long variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from double to byte.

But it is possible for conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are satisfied:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.

Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.

Casting Incompatible Types

The automatic type conversions are helpful, they will not fulfil all needs. For example, if we want to assign an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This kind of conversion is sometimes called a *narrowing conversion*, since you are explicitly making the value narrower so that it will fit

into the target type. To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.

It has this general form:

(target-type) value

Here, *target-type* specifies the desired type to convert the specified value to.

Example:

```
int a;  
byte b;  
// ...  
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As integers do not have fractional components so, when a floating-point value is assigned to an integer type, the fractional component is lost.

Program:

```
class Conversion  
{  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
        System.out.println("\nConversion of int to byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
        System.out.println("\nConversion of double to int.");  
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
        System.out.println("\nConversion of double to byte.");  
        b = (byte) d;  
        System.out.println("d and b " + d + " " + b);  
    }  
}
```

Output:

```
Conversion of int to byte.  
i and b 257 1  
Conversion of double to int.  
d and i 323.142 323  
Conversion of double to byte.  
d and b 323.142 67  
byte b = 50;  
b = (byte)(b * 2);
```

The Type Promotion Rules:

In addition to the elevation of bytes and shorts to int, Java defines several *type promotion rules* that apply to expressions. They are as follows. First, all byte and short values are promoted to int. Then, if one operand is a long, the whole expression is promoted to long. If one operand is a float, the entire expression is promoted to float. If any of the operands is double, the result is double.

The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:

```

class Promote {
public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
}
}
double result = (f * b) + (i / c) - (d * s);

```

In the first sub expression, $f * b$, b is promoted to a float and the result of the sub expression is float. Next, in the sub expression i / c , c is promoted to int, and the result is of type int. Then, in $d * s$, the value of s is promoted to double, and the type of the sub expression is double. Finally, these three intermediate values, float, int, and double, are considered. The outcome of float plus an int is a float. Then the resultant float minus the last double is promoted to double, which is the type for the final result of the expression.

Simple Java Program:

```

class Example
{
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}

```

Here public is an access modifier, which means this method can be accessed by any one out side the class.

Static allows the main () method to be called without initiating any instance for the class.

Void tells the compiler that main() doesnot return any type.

String args[] declares a parameter named args, which is an array of instances of class string.

args takes the arguments for a command line.

Classes and Objects:

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an *instance* of the *class of objects* known as bicycles. A *class* is the blueprint from which individual objects are created.

Declaring Member Variables

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

A class is declared by use of the **class** keyword

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

The data, or variables, defined within a **class** are called *instance variables*. The code is contained within *methods*. Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

Example:

Class demo

```
{
int x=1;;
int y=2;
float z=3;
void display()
{
System.out.println("values of x, y and z are:"+x+" "+y+" "+z);
}
}
```

Declaring Objects

when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that

is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

In the above programs to declare an object of type demo:

```
Demo d1 = new demo();
```

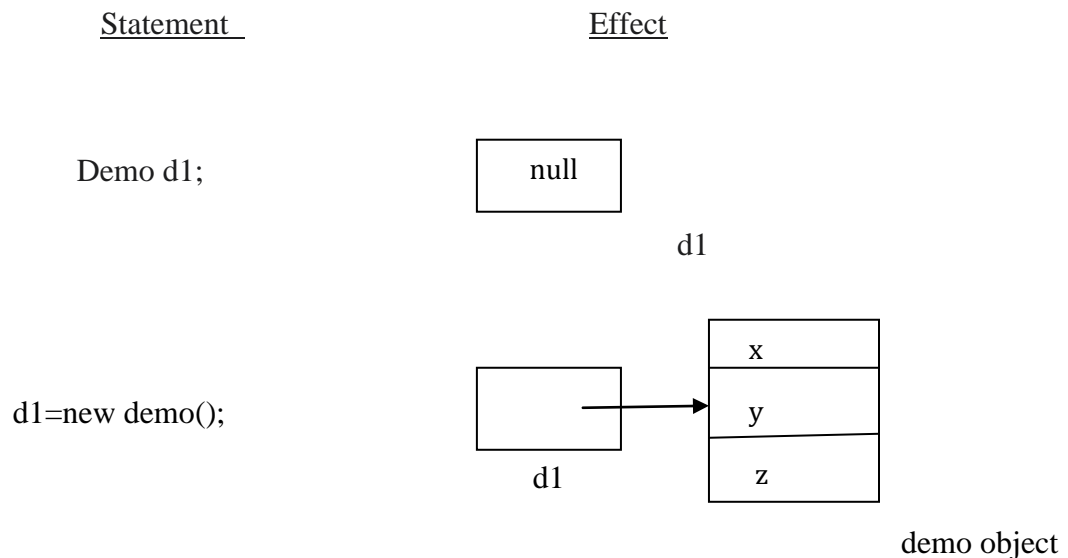
This statement combines the two steps just described. It can be rewritten like this to

show each step more clearly:

```
demo d1; // declare reference to object
```

```
d1 = new demo(); // allocate a demo object
```

The first line declares **d1** as a reference to an object of type demo. After this line executes, d1 contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use d1 at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **d1**. After the second line executes; you can use d1 as if it were a demo object. But in reality, d1 simply holds the memory address of the actual demo object. The effect of these two lines of code is depicted in Figure.



Constructors:

A class contains constructors that are invoked to create objects from the class blueprint.

Constructor declarations look like method declarations—except that they use the name of the class and have no return type. A constructor initializes an object immediately upon creation.

Constructors can be default or parameterized constructors.

A default constructor is called when an instance is created for a class.

Example

Class demo

```
{
int x;
int y;
float z;
demo()
{
X=1;
Y=2;
```

```

Z=3;
}
void display()
{
System.out.println("values of x, y and z are:"+x+" "+y+" "+z);
}
}
Class demomain
{
Public static void main(String args[])
{
demo d1=new demo(); // this is a call for the above default constructor
d1.display();
}
}

```

Parameterized constructor:

```

Class demo
{
int x;
int y;
float z;
demo(int x1,int y1,int z1)
{
x=x1;
y=y1;
z=z1;
}
void display()
{
System.out.println("values of x, y and z are:"+x+" "+y+" "+z);
}
}
Class demomain
{
Public static void main(String args[])
{
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
}
}

```

This Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

Example:

```
Class demo
{
int x;
int y;
float z;
demo(int x,int y,int z)
{
    this.x=x;
    this.y=y;
    this.z=z;
}
void display()
{
System.out.println("values of x, y and z are:"+x+" "+y+" "+z);
}
}
Class demomain
{
Public static void main(String args[])
{
demo d1=new demo(1,2,3); // this is a call for the above parameterized constructor
d1.display();
}
}
```

Output:

Values of x, y and z are:1 2 3

To differentiate between the local and instance variables we have used this keyword `int` in the constructor.

Garbage Collection:

Since objects are dynamically allocated by using the **new** operator, objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it

handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.

The finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization*. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the finalize() method you will specify those actions that must be performed before an object is destroyed. The garbage

collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the `finalize()` method on the object.

The `finalize()` method has this general form:

```
protected void finalize()  
{  
    // finalization code here  
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize()** by code defined outside its class.

Overloading Methods:

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists .

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, `drawString`, `drawInteger`, `drawFloat`, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named `draw`, each of which has a different parameter list.

```
public class DataArtist {  
    ...  
    public void draw(String s) {  
        ...  
    }  
    public void draw(int i) {  
        ...  
    }  
    public void draw(double f) {  
        ...  
    }  
    public void draw(int i, double f) {  
        ...  
    }  
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, `draw(String s)` and `draw(int i)` are distinct and unique methods because they require different argument types. You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart. The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Overloading Constructors:

We can overload constructor methods

```
class Box {  
    double width;  
    double height;  
    double depth;
```

```
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

As you can see, the **Box()** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box()** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box()** requires three arguments.

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
```

```
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons
{
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
```

```
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
//get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Parameter Passing:

In general, there are two ways that a computer language can pass an argument to a subroutine.

The first way is *call-by-value*. In this method copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

The second way an argument can be passed is *call-by-reference*. In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

Java uses both approaches, depending upon what is passed.

In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
// Simple types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}
class CallByValue
{
public static void main(String args[])
{
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " + a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " + a + " " + b);
}
}
```

output :

```
a and b before call: 15 20
a and b after call: 15 20
```

we can see, the operations that occur inside **meth()** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When we pass an object to a method, the situation changes dramatically, because objects are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument.

Example:

// Objects are passed by reference.

```
class Test {
    int a, b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    // pass an object
    void meth(Test o)
    {
        o.a *= 2;
        o.b /= 2;
    }
}
class CallByRef
{
    public static void main(String args[])
    {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
    }
}
```

output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

in this case, the actions inside **meth()** have affected the object used as an argument.

Recursion:

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

The classic example of recursion is the computation of the factorial of a number.

The factorial of a number N is the product of all the whole numbers between 1 and N .

For example, 3 factorial is $1 \times 2 \times 3$, or 6. Here is how a factorial can be computed by use of a recursive method:

// A simple example of recursion.

```
class Factorial
{
```



```
// this is a recursive function
int fact(int n)
{
    int result;
    if(n==1) return 1;
    result = fact(n-1) * n;
    return result;
}
}
class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();
        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

Output:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

String Handling:

In Java a *string* is a sequence of characters. But, unlike many other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient.

when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, there is a companion class to **String** called **StringBuffer**, whose objects contain strings that can be modified after they are created.

Both the **String** and **StringBuffer** classes are defined in **java.lang**. Thus, they are available to all programs automatically. Both are declared **final**, which means that neither of these classes may be subclassed.

The String Constructors:

The **String** class supports several constructors. To create an empty **String**, you call the default constructor.

For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it. Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

```
String(char chars[ ])
```

Example:

```
char chars[] = { 'a', 'b', 'c' };
```

```
String s = new String(chars);
```

This constructor initializes **s** with the string “abc”.

You can specify a subrange of a character array as an initializer using the following constructor:

```
String(char chars[ ], int startIndex, int numChars)
```

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use.

Example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
```

```
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

```
String(String strObj)
```

Here, *strObj* is a **String** object.

String Length:

The length of a string is the number of characters that it contains. To obtain this value, call the **length()** method.

```
int length( )
```

Example:

```
char chars[] = { 'a', 'b', 'c' };
```

```
String s = new String(chars);
```

```
System.out.println(s.length());
```

It prints 3 as the output since the string as 3 characters.

```
charAt( )
```

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt()** method. It has this general form:

```
char charAt(int where)
```

```
getChars( )
```

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

```
equals()
```

To compare two strings for equality, use **equals()**. It has this general form:

```
boolean equals(Object str)
```

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase()**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

```
boolean equalsIgnoreCase(String str)
```

Here, *str* is the **String** object being compared with the invoking **String** object.

```
compareTo()
```

to know whether two strings are identical. For sorting applications, you need to know which is *less than*, *equal to*, or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo()** serves this purpose. It has this general form:

```
int compareTo(String str)
```

Here, *str* is the **String** being compared with the invoking **String**.

```
indexOf()
```

The **String** class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf()** Searches for the first occurrence of a character or substring.
- **lastIndexOf()** Searches for the last occurrence of a character or substring.

```
substring()
```

we can extract a substring using **substring()**. It has two forms. The first is

```
String substring(int startIndex)
```

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring()** allows you to specify both the beginning and ending index of the substring:

```
String substring(int startIndex, int endIndex)
```

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

```
replace()
```

The **replace()** method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

```
String replace(char original, char replacement)
```

StringBuffer Constructors:

StringBuffer defines these three constructors:

```
StringBuffer()
```

```
StringBuffer(int size)
```

```
StringBuffer(String str)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more

characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place.

```
length() and capacity()
```

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms:

```
int length()
```

`int capacity()`

`ensureCapacity()`

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity()** to set the size of the buffer. **ensureCapacity()** has this general form:

`void ensureCapacity(int capacity)`

Here, *capacity* specifies the size of the buffer

`append()`

The **append()** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has overloaded versions for all the built-in types and for **Object**. Here are a few of its forms:

`StringBuffer append(String str)`

`StringBuffer append(int num)`

`StringBuffer append(Object obj)`

String.valueOf() is called for each parameter to obtain its string representation.

`insert()`

The **insert()** method inserts one string into another.

`StringBuffer insert(int index, String str)`

`StringBuffer insert(int index, char ch)`

`StringBuffer insert(int index, Object obj)`

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object

`reverse()`

You can reverse the characters within a **StringBuffer** object using **reverse()**, shown here:

`StringBuffer reverse()`

This method returns the reversed object on which it was called

`delete()` and `deleteCharAt()`

to delete characters using the methods **delete()** and **deleteCharAt()**. These methods are shown here:

`StringBuffer delete(int startIndex, int endIndex)`

`StringBuffer deleteCharAt(int loc)`

The **delete()** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt()** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object

`replace()`

to replaces one set of characters with another set inside a **StringBuffer** object. Its signature is shown here:

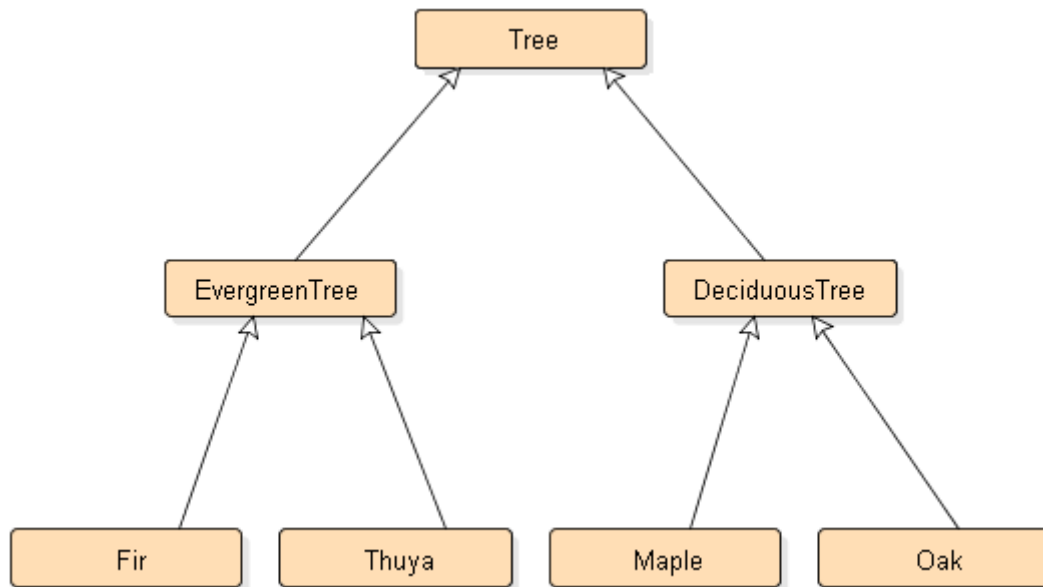
`StringBuffer replace(int startIndex, int endIndex, String str)`

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced.

Inheritance

Hierarchical Abstractions

Example:



- Abstraction consists of eliminating unnecessary detail and concentrating on essential features
- The concept of an evergreen tree is an abstraction
- Fir trees, Thuya trees, Pine trees, ... have features that are common to all evergreen trees
- The concept of a deciduous tree is an abstraction
- Maple trees, Oak trees, Apple trees, ... have features that are common to all deciduous trees

Inheritance is a Java language feature, used to model hierarchical abstraction

Inheritance means taking a class (called the base class) and defining a new class (called the subclass) by specializing the state and behaviors of the base class

Subclasses specialize the behaviors of their base class

- The subclasses inherit the state and behaviors of their base class
- They can have additional state and behaviors

Inheritance is mainly used for code reusability and to reduce the complexity of the program.

Base Class:

A class that is being inherited is known as Base class

For ex:

Class x

```
{  
Void method ();  
}  
class y extends x  
{  
}
```

In the above example the class x is known as the base class.

When we create an object for the base class then that object is known as base class object.

Sub class:

The class that is inheriting the base class Is known as sub class.

In the example above since class y is inheriting the class x, class y is known as the subclass to class x.

Forms of inheritance:

1. Single Level Inheritance:

When one base class is being inherited by one sub class then that kind of inheritance is known as single level inheritance.

2. Multi Level Inheritance:

When a sub class is in turn being inherited then that kind of inheritance is known as multi level inheritance.

3. Hierarchical Inheritance:

When a base class is being inherited by one or more sub class then that kind of inheritance is known as hierarchical inheritance.

A sub class uses the keyword “extends” to inherit a base class.

Example for Single Inheritance

```
class x
{
int a;
void display()
{
a=0;
System.out.println(a);
}
}
class y extends x
{
int b;
void show()
{
B=1;
System.out.println(b);
}
}
class show_main
{
Public static void main(String args[])
{
y y1=new y();
y1.display();
y1.show();
}
}
```

Output:

0

1

Since the class y is inheriting class x, it is able to access the members of class x.

Hence the method display() can be invoked by the instance of the class y.

Example for multilevel inheritance:

```
class x
{
int a;
void display()
{
a=0;
System.out.println(a);
}
}
class y extends x
{
int b;
void show()
{
B=1;
System.out.println(b);
}
}
class z extends y
{
Int c;
void show1()
{
c=2;
System.out.println(c);
}
}
class show_main
{
Public static void main(String args[])
{
z z1=new z();
z1.display();
z1.show();
}
}
```

Output

0
1
2

Since class z is inheriting class y which is in turn a sub class of the class x, indirectly z can access the members of class x.

Hence the instance of class z can access the display () method in class x, the show () method in class y.

Problems:

In java multiple level inheritance is not possible easily. We have to make use of a concept called interfaces to achieve it.

Access Specifiers:

The different access specifiers used are

1. Public
2. Private
3. Protected
4. Default
5. Privateprotected

1. Private members can be accessed only within the class in which they are declared.
2. Protected members can be accessed inside the class in which they are declared and also in the sub classes in the same package and sub classes in the other packages.
3. Default members are accessed by the methods in their own class and in the sub classes of the same package.
4. Public members can be accessed anywhere.

Super Keyword:

Whenever a sub class needs to refer to its immediate super class, we can use the super keyword.

Super has two general forms

1. The first calls the super class constructor.
2. The second is used to access a member of the super class that has been hidden by a member of a sub class

Syntax:

A sub class can call a constructor defined by its super class by use of the following form of super.

super(arg-list);

here arg-list specifies any arguments needed by the constructor in the super class .

The second form of super acts like a “this” keyword. The difference between “this” and “super” is that “this” is used to refer the current object where as the super is used to refer to the super class.

The usage has the following general form:

super.member;

Example:

Class x

```
{
int a;
x()
{
a=0;
}
void display()
{
System.out.println(a);
}
}
class y extends x
{
int b;

y()
{
```



```

    super();
    b=1;
    }
    Void display()
    {
    Super.display();
    System.out.println(b);
    }
    }
    class super_main
    {
    Public static void main(String args[])
    {
    y y1=new y();
    y1.display();
    }
    }

```

Using final with inheritance:

The keyword final has three uses.

1. First it can be used to create the equivalent of a named constant.
2. To prevent overriding.
3. To prevent inheritance.

Using final to prevent overriding:

To disallow a method from being overridden, specify final as a modifier at the start of the declaration.

Methods declared as final cannot be overridden.

Syntax:

final <return type> <method name> (argument list);

Using final with inheritance:

Some times we may want to prevent a class from being inherited.

In order to do this we must precede the class declaration with final.

Declaring a class as final implicitly declares all its methods as final.

Example:

```

final class A
{
.....//members
}

```

Polymorphism- dynamic binding:

Dynamic binding is a binding in which the class association is not made until the object is created at execution time. It is also called as late binding.

In java, base class reference can be assigned objects of sub class. When methods of sub class object are called through base class's reference. The mapping / binding or function calls to respective function takes place after running the program, atleast possible moment. This kind of binding is known as "late binding" or "dynamic binding" or "runtime polymorphism".

Eg.

```
class A
```

```

{
public void display()
{ }
}
class B extends A
{
public void display()
{ }
}
class C extends A
{
public void display()
{ }
}
class DispatchDemo
{
psvm (String args[])
{
A ob1= new A();
B ob2 = new B();
C ob3 = new C();
A r;
r=ob1;
r.display();
r=ob2;
r.display();
r=ob3;
r.display();
}
}

```

In the above programs, the statement `r.display()`, calls the `display()` method and based on the object. That has been assigned to base class reference. i.e. `r`. But the mapping / binding as to which method is to be invoked is done only at run time.

Method Overriding:

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

// Method overriding.

```

class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}}
class B extends A {

```

```

int k;
B(int a, int b, int c) {
super(a, b);
k = c;
} // display k – this overrides show() in A
void show() {
System.out.println("k: " + k);
}}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show(); // this calls show() in B
}}

```

The output produced by this program is shown here:

k: 3

When **show()** is invoked on an object of type **B**, the version of **show()** defined within **B** is used. That is, the version of **show()** inside **B** overrides the version declared in **A**.

Abstract classes and methods:

We can require that some methods be overridden by sub classes by specifying the abstract type modifier.

These methods are sometimes referred to as sub classer responsibility as they have no implementation specified in the super class.

Thus a sub class must override them.

To declare an abstract method we have:

abstract type name (parameter list);

Any class that contains one or more abstract methods must also be declared abstract..

Such types of classes are known as abstract classes.

Abstract classes can contain both abstract and non-abstract methods.

Let us consider the following example:

```

abstract class A
{
abstract void callme();
void call()
{
System.out.println("HELLO");
}}
class B extends A
{
Void callme()
{
System.out.println("GOOD MORNING");
}}
class abstractdemo
{
Public static void main(String args[]){
B b=new B();
b.callme();
b.call();
}}

```

Output:

GOOD MORNING
HELLO

UNIT-II

Packages and Interfaces

Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts:

- A single package statement (optional)
- Any number of import statements (optional)
- a single public class declaration (required)
- Any number of classes private to the package (optional)

Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

Defining a Package:

Creating a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored.

If you omit the **package** statement, the class names are put into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

Creating a Package:

The general form of the **package** statement:

```
package pkg;
```

Here, *pkg* is the name of the package.

For example, the following statement creates a package called **MyPackage**.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system.

For example, a package declared as

```
package java.awt.image;
```

needs to be stored in **java/awt/image**, **java\awt\image**, or **java:awt:image** on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH:

Java run-time system know where to look for packages that you create? The answer has two parts.

- First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
- Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

For example, consider the following package specification.

```
package MyPack;
```

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**. The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in.

Create the package directories below your current development directory, put the **.class** files into the appropriate directories and then execute the programs from the development directory.

Example:

```
// A simple package
package MyPack;
class Balance
{
    String name;
    double bal;
    Balance(String n, double b)
    {
        name = n;
        bal = b;
    }
    void show()
    {
        if(bal<0)
            System.out.print("--> ");
        System.out.println(name + ": $" + bal);
    }
}
class AccountBalance
{
    public static void main(String args[])
    {
        Balance current[] = new Balance[3];
        current[0] = new Balance("K. J. Fielding", 123.23);
        current[1] = new Balance("Will Tell", 157.02);
        current[2] = new Balance("Tom Jackson", -12.33);
        for(int i=0; i<3; i++)
            current[i].show();
    }
}
```

Save this file as **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, we should be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately.

AccountBalance is now part of the package **MyPack**. This means that it cannot be executed by itself.

That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- ✚ Subclasses in the same package
- ✚ Non-subclasses in the same package
- ✚ Subclasses in different packages
- ✚ Classes that are neither in the same package nor subclasses

Table: Class member access

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different Package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

Example:

This is file **Protection.java**:

```
package p1;
public class Protection
{
int n = 1;
```

```

private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection()
{
    System.out.println("base constructor");
    System.out.println("n = " + n);
    System.out.println("n_pri = " + n_pri);
    System.out.println("n_pro = " + n_pro);
    System.out.println("n_pub = " + n_pub);
}
}

```

This is file **Derived.java**:

```

package p1;
class Derived extends Protection
{
    Derived()
    {
        System.out.println("derived constructor");
        System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
    }
}

```

This is file **SamePackage.java**:

```

package p1;
class SamePackage
{
    SamePackage()
    {
        Protection p = new Protection();
        System.out.println("same package constructor");
        System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
    }
}

```

This is file **Protection2.java**:

```

package p2;
class Protection2 extends p1.Protection
{
    Protection2()
    {
        System.out.println("derived other package constructor");
        // class or package only
        // System.out.println("n = " + n);
    }
}

```



```
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
This is file OtherPackage.java:
package p2;
class OtherPackage
{
OtherPackage()
{
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
{
public static void main(String args[])
{
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo
{
public static void main(String args[])
{
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

Importing Packages:

Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is convenient.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

This is the general form of the **import** statement:

```
import pkg1[.pkg2].(classname|*);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*), which indicates that the Java compiler should import the entire package.

This code fragment shows both forms in use:

```
import java.util.Date;
import java.io.*;
```

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use.

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.

For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;
/* Now, the Balance class, its constructor, and its
show() method are public. This means that they can
be used by non-subclass code outside their package.
*/
public class Balance
{
String name;
double bal;
public Balance(String n, double b)
{
name = n;
bal = b;
}
public void show()
{
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
```

```
}
```

As you can see, the **Balance** class is now **public**. Also, its constructor and its **show()** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance
{
public static void main(String args[])
{
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented.

Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. Each class can determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

❖ Interfaces are designed to support dynamic method resolution at run time.

Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and not extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

Defining an Interface:

An interface is defined much like a class.

This is the general form of an interface:

```
access interface name
{
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
```

```
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here, access is either public or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. name is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon

after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly final and static, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly public if the interface, itself, is declared as public.

An example of an interface definition. It declares a simple interface which contains one method called callback() that takes a single integer parameter.

```
interface Callback
{
void callback(int param);
}
```

Implementing Interfaces:

Once an interface has been defined, one or more classes can implement that interface.

To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface.

The general form of a class that implements the interface:

```
access class classname [extends superclass][implements interface [,interface...]]
{
// class-body
}
```

Here, access is either public or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.

Example: class that implements the Callback interface shown earlier.

```
class Client implements Callback
{
public void callback(int p)
{
System.out.println("callback called with " + p);
}
}
```

Notice that callback() is declared using the public access specifier. When you implement an interface method, it must be declared as public.

For example, the following version of Client implements callback() and adds the method nonIfaceMeth():

```
class Client implements Callback
{
```

```

public void callback(int p)
{
    System.out.println("callback called with " + p);
}
void nonIfaceMeth()
{
    System.out.println("Classes that implement interfaces"+"may also define other members, too.");
}
}

```

Accessing Implementations Through Interface References:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee.” This process is similar to using a superclass reference to access a subclass object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the callback() method via an interface reference variable:

```

class TestIface
{
    public static void main(String args[])
    {
        Callback c = new Client();
        c.callback(42);
    }
}

```

Output:

callback called with 42

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback() method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

Thus, c could not be used to access nonIfaceMeth() since it is defined by Client but not Callback.

The preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

```

// Another implementation of Callback.
class AnotherClient implements Callback
{
    public void callback(int p)
    {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
    }
}

```

```

class TestIface2
{
public static void main(String args[])
{
    Callback c = new Client();
    AnotherClient ob = new AnotherClient();
    c.callback(42);
    c = ob; // c now refers to AnotherClient object
    c.callback(42);
}
}

```


Output:

callback called with 42

Another version of callback

p squared is 1764

As you can see, the version of callback() that is called is determined by the type of object that c refers to at run time.

 If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

For example:

abstract class Incomplete implements Callback

```

{
int a, b;
void show()
{
    System.out.println(a + " " + b);
}
}

```

Here, the class Incomplete does not implement callback() and must be declared as abstract. Any class that inherits Incomplete must implement callback() or be declared abstract itself.

Applying Interfaces:

We define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations.

Example:

// Define an integer stack interface.

```

interface IntStack
{
    void push(int item); // store an item
    int pop(); // retrieve an item
}

```

The following program creates a class called FixedStack that implements a fixed-length version of an integer stack:

Example:

// An implementation of IntStack that uses fixed storage.

```

class FixedStack implements IntStack
{
    private int stck[];

```

```

private int tos;
FixedStack(int size)
{
    stck = new int[size];
    tos = -1;
}
// Push an item onto the stack
public void push(int item)
{
    if(tos==stck.length-1) // use length member
        System.out.println("Stack is full.");
    else
        stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
{
    if(tos < 0)
    {
        System.out.println("Stack underflow.");
        return 0;
    }
    else
        return stck[tos--];
}
}
class IFTest
{
    public static void main(String args[])
    {
        FixedStack mystack1 = new FixedStack(5);
        FixedStack mystack2 = new FixedStack(8);
        // push some numbers onto the stack
        for(int i=0; i<5; i++) mystack1.push(i);
        for(int i=0; i<8; i++) mystack2.push(i);
        // pop those numbers off the stack
        System.out.println("Stack in mystack1:");
        for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
        System.out.println("Stack in mystack2:");
        for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
    }
}

```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

Example:

```

// Implement a "growable" stack.
class DynStack implements IntStack

```

```
{
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size)
{
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item)
{
// if stack is full, allocate a larger stack
if(tos==stck.length-1)
{
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++)
temp[i] = stck[i];
stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
{
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2
{
public static void main(String args[])
{
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```



```
}
```

The following class uses both the FixedStack and DynStack implementations. It does so through an interface reference. This means that calls to push() and pop() are resolved at run time rather than at compile time.

```
/* Create an interface variable and access stacks through it.
```

```
*/
```

```
class IFTest3
{
public static void main(String args[])
{
    IntStack mystack; // create an interface reference variable
    DynStack ds = new DynStack(5);
    FixedStack fs = new FixedStack(8);
    mystack = ds; // load dynamic stack
    // push some numbers onto the stack
    for(int i=0; i<12; i++)
        mystack.push(i);
    mystack = fs; // load fixed stack
    for(int i=0; i<8; i++)
        mystack.push(i);
    mystack = ds;
    System.out.println("Values in dynamic stack:");
    for(int i=0; i<12; i++)
        System.out.println(mystack.pop());
    mystack = fs;
    System.out.println("Values in fixed stack:");
    for(int i=0; i<8; i++)
        System.out.println(mystack.pop());
    }
}
```

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push() and pop() defined by the DynStack implementation. When it refers to fs, it uses the versions of push() and pop() defined by FixedStack.

These determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces:

Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When we include that interface in a class (that is, when you “implement” the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything. It is as if that class were importing the constant variables into the class name space as final variables.

Example:

```
import java.util.Random;
interface SharedConstants
```

```
{
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants
{
Random rand = new Random();
int ask()
{
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants
{
static void answer(int result)
{
switch(result)
{
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
```

```

}
public static void main(String args[])
{
    Question q = new Question();
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
    answer(q.ask());
}
}

```

Output:

Later

Soon

No

Yes

This program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods which allow you to obtain random numbers in the form required by your program. In this example, the method nextDouble() is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly.

Interfaces Can Be Extended:

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

Example:

```

// One interface can extend another.
interface A
{
    void meth1();
    void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A
{
    void meth3();
}
// this class must implement all of A and B
class MyClass implements B
{
    public void meth1()
    {
        System.out.println("Implement meth1().");
    }
    public void meth2()

```

```

{
System.out.println("Implement meth2().");
}
public void meth3()
{
System.out.println("Implement meth3().");
}
}
class IFExtend
{
public static void main(String arg[])
{
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}

```

Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

EXPLORING JAVA.IO PACKAGE:

java.io, which provides support for I/O operations Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. A stream is a logical entity that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ.

Some of the I/O classes defined by **java.io** are:

FileWriter, BufferedOutputStream, FilterInputStream, BufferedReader, FilterOutputStream, BufferedWriter, FilterReader, DataInputStream, RandomAccessFile
DataOutputStream.

File:

most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Files are a primary source and destination for data within many programs. Files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list()** method.

The following constructors can be used to create **File** objects:

```
File(String directoryPath)  
File(String directoryPath, String filename)  
File(File dirObj, String filename)  
File(URI uriObj)
```

Here, *directoryPath* is the path name of the file, *filename* is the name of the file, *dirObj* is a **File** object that specifies a directory, and *uriObj* is a **URI** object that describes a file.

The following example demonstrates several of the **File** methods:

```
// Demonstrate File.  
import java.io.File;  
class FileDemo  
{  
    static void p(String s)  
    {  
        System.out.println(s);  
    }  
    public static void main(String args[])  
    {  
        File f1 = new File("/java/COPYRIGHT");  
        p("File Name: " + f1.getName());  
        p("Path: " + f1.getPath());  
        p("Abs Path: " + f1.getAbsolutePath());  
        p("Parent: " + f1.getParent());  
        p(f1.exists() ? "exists" : "does not exist");  
        p(f1.canWrite() ? "is writeable" : "is not writeable");  
        p(f1.canRead() ? "is readable" : "is not readable");  
        p("is " + (f1.isDirectory() ? "" : "not") + " a directory");  
        p(f1.isFile() ? "is normal file" : "might be a named pipe");  
        p(f1.isAbsolute() ? "is absolute" : "is not absolute");  
        p("File last modified: " + f1.lastModified());  
        p("File size: " + f1.length() + " Bytes");  
    }  
}
```

When you run this program, you will see something similar to the following:

```
File Name: COPYRIGHT  
Path: /java/COPYRIGHT  
Abs Path: /java/COPYRIGHT  
Parent: /java  
exists  
is writeable  
is readable  
is not a directory  
is normal file  
is absolute  
File last modified: 812465204000  
File size: 695 Bytes
```

The Stream Classes:

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and **OutputStream** are designed for byte streams.

Reader and **Writer** are designed for character streams.

The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

Byte Stream Classes:

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an **IOException** on error conditions.

Some of the methods in this class are:

- `int available()`: Returns the number of bytes of input currently available for reading.
- `void close()`: Closes the input source. Further read attempts will generate an **IOException**.
- `int read()`: Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
- `int read(byte buffer[])`: Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. -1 is returned when the end of the file is encountered.

OutputStream

OutputStream is an abstract class that defines streaming byte output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors.

Some of the methods in this class are:

- `void close()`: Closes the output stream. Further write attempts will generate an **IOException**.
- `void write(int b)`: Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write()** with expressions without having to cast them back to **byte**.
- `void write(byte buffer[])`: Writes a complete array of bytes to an output stream.

1. FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Its two most common constructors are shown here:

`FileInputStream(String filepath)`

`FileInputStream(File fileObj)`

Either can throw a **FileNotFoundException**. Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

2. FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. Its most commonly used constructors are shown here:

`FileOutputStream(String filePath)`

`FileOutputStream(File fileObj)`

`FileOutputStream(String filePath, boolean append)`

`FileOutputStream(File fileObj, boolean append)`

They can throw a `FileNotFoundException` or a `SecurityException`. Here, *filePath* is the full path name of a file, and *fileObj* is a `File` object that describes the file. If *append* is true, the file is opened in append mode

The Character Streams:

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the “write once, run anywhere” philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes.

Reader

Reader is an abstract class that defines Java’s model of streaming character input. All of the methods in this class will throw an **IOException** on error conditions. Table 17-3 provides a synopsis of the methods in **Reader**.

Writer

Writer is an abstract class that defines streaming character output. All of the methods in this class return a **void** value and throw an **IOException** in the case of errors.

Table 17-4 shows a synopsis of the methods in **Writer**.

1. FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

`FileReader(String filePath)`

`FileReader(File fileObj)`

2. FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

`FileWriter(String filePath)`

`FileWriter(String filePath, boolean append)`

`FileWriter(File fileObj)`

`FileWriter(File fileObj, boolean append)`

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file.

Exception handling

Exception-Handling Fundamentals:

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw.

Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method returns is put in a finally block.

general form of an exception-handling block:

```
try {  
    // block of code to monitor for errors  
}  
catch (ExceptionType1 exOb)  
{  
    // exception handler for ExceptionType1  
}  
catch (ExceptionType2 exOb)  
{  
    // exception handler for ExceptionType2  
}  
// ...  
  
Finally  
{  
    // block of code to be executed before try block ends  
}
```

Here, ExceptionType is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

Exception Types:

All exception types are subclasses of the built-in class `Throwable`. Thus, `Throwable` is at the top of the exception class hierarchy. Immediately below `Throwable` are two subclasses that partition exceptions into two distinct branches. One branch is headed by `Exception`. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of `Exception`, called `RuntimeException`. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by `Error`, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type `Error` are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions:

program includes an expression that intentionally causes a divide-by-zero error.

```
class Exc0
{
public static void main(String args[])
{
int d = 0;
int a = 42 / d;
}
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Output:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace. Also, notice that the type of the exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

Using try and catch:

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. It provides two benefits:

- 1) it allows you to fix the error.
- 2) it prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2
```

```

{
public static void main(String args[])
{
int d, a;
try
{
// monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{
// catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
output:
Division by zero.
After catch statement.

```

Notice that the call to **println()** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not “called,” so execution never “returns” to the **try** block from a **catch**. Thus, the line “This will not be printed.” is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement. The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

Example:

```

// Handle an exception and move on.
import java.util.Random;
class HandleError
{
public static void main(String args[])
{
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++)
{
try
{
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
}
catch (ArithmeticException e)
{
System.out.println("Division by zero.");
}
}
}

```

```

a = 0; // set a to zero and continue
}
System.out.println("a: " + a);
}
}
}

```

Throwable overrides the **toString()** method (defined by **Object**) so that it returns a string containing a description of the exception. we can display this description in a **println()** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```

catch (ArithmeticException e)
{
System.out.println("Exception: " + e);
a = 0; // set a to zero and continue
}

```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

The following example traps two different exception types:

```

// Demonstrate multiple catch statements.
class MultiCatch
{
public static void main(String args[])
{
Try
{
int a = args.length;
System.out.println("a = " + a);
int b = 42 / a;
int c[] = { 1 };
c[42] = 99;
}
catch(ArithmeticException e)
{
System.out.println("Divide by 0: " + e);
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}

```

This program will cause a division-by-zero exception if it is started with no commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

output generated by running it both ways:

```
C:\>java MultiCatch
```

```
a = 0
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
```

```
C:\>java MultiCatch TestArg
```

```
a = 1
```

```
Array index oob: java.lang.ArrayIndexOutOfBoundsException
```

```
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.
```

```
A subclass must come before its superclass in a series of catch statements. If not, unreachable code will be created and a compile-time error will result.
```

```
*/
```

```
class SuperSubCatch
```

```
{
```

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

```
{
```

```
Try
```

```
{
```

```
int a = 0;
```

```
int b = 42 / a;
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
System.out.println("Generic Exception catch.");
```

```
}
```

```
/* This catch is never reached because
```

```
ArithmeticException is a subclass of Exception. */
```

```
catch(ArithmeticException e) { // ERROR - unreachable
```

```
System.out.println("This is never reached.");
```

```
}
```

```
}
```

```
}
```

If you try to compile this program, you will receive an error message stating that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

Nested try Statements:

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statements:

Example of nested try statements:

```
class NestTry
{
public static void main(String args[])
{
Try
{
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

this program nests one **try** block within another. The program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program by one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the inner **try** block.

Here are sample runs that illustrate each case:

C:\>java NestTry

Divide by 0: java.lang.ArithmeticException: / by zero

```
C:\>java NestTry One
```

```
a = 1
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
C:\>java NestTry One Two
```

```
a = 2
```

```
Array index out-of-bounds:
```

```
java.lang.ArrayIndexOutOfBoundsException
```

Nesting of **try** statements can occur in less obvious ways when method calls are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method.

Here is the previous program recoded so that the nested **try** block is moved inside the method **nesttry()**:

```
/* Try statements can be implicitly nested via
calls to methods. */
class MethNestTry {
static void nesttry(int a) {
try { // nested try block
/* If one command-line arg is used,
then a divide-by-zero exception
will be generated by the following code. */
if(a==1) a = a/(a-a); // division by zero
/* If two command-line args are used,
then generate an out-of-bounds exception. */
if(a==2) {
int c[] = { 1 };
c[42] = 99; // generate an out-of-bounds exception
}
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index out-of-bounds: " + e);
}
}
public static void main(String args[]) {
try {
int a = args.length;
/* If no command-line args are present,
the following statement will generate
a divide-by-zero exception. */
int b = 42 / a;
System.out.println("a = " + a);
nesttry(a);
} catch(ArithmeticException e) {
System.out.println("Divide by 0: " + e);
}
}
}
```

Output:

```
throw
```

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement.

The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator.

The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
```

```
class ThrowDemo
{
    static void demoproc()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

This program gets two chances to deal with the same error. First, **main()** sets up an exception context and then calls **demoproc()**. The **demoproc()** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown.

output:

```
Caught inside demoproc.
```

```
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects.

Pay close attention to this line:

```
throw new NullPointerException("demo");
```


Here, **new** is used to construct an instance of **NullPointerException**. All of Java's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

Throws:

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.

All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

type method-name(parameter-list) throws exception-list

```
{
// body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

finally:

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency.

finally creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
} finally {
System.out.println("procA's finally");
}
}
}
// Return from within a try block.
```



```

static void procB() {
try {
System.out.println("inside procB");
return;
} finally {
System.out.println("procB's finally");
}
}
// Execute a try block normally.
static void procC() {
try {
System.out.println("inside procC");
} finally {
System.out.println("procC's finally");
}
}
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}

```

In this example, **procA()** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB()**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB()** returns. In **procC()**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

*If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.*

output:

```

inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

```

Java's Built-in Exceptions:

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table. Table lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself.

These are called *checked exceptions*. Java defines several other types of exceptions that relate to its various class libraries.

Java's Unchecked RuntimeException Subclasses:

Exception Meaning

- 1.ArithmeticException Arithmetic error, such as divide-by-zero.
- 2.ArrayIndexOutOfBoundsException Array index is out-of-bounds.
- 3.ArrayStoreException Assignment to an array element of an incompatible type.
- 4.ClassCastException Invalid cast.
- 5.IllegalArgumentException Illegal argument used to invoke a method.
- 6.IllegalMonitorStateException Illegal monitor operation, such as waiting on an unlocked thread.
- 7.IllegalStateException Environment or application is in incorrect state.
- 8.IllegalThreadStateException Requested operation not compatible with current thread state.
- 9.IndexOutOfBoundsException Some type of index is out-of-bounds.
- 10.NegativeArraySizeException Array created with a negative size.

Java's Checked Exceptions Defined in java.lang:

Exception Meaning

- 1.ClassNotFoundException Class not found.
- 2.CloneNotSupportedException Attempt to clone an object that does not implement the **Cloneable** interface.
- 3.IllegalAccessException Access to a class is denied.
- 4.InstantiationException Attempt to create an object of an abstract class or interface.
- 5.InterruptedOperationException One thread has been interrupted by another thread.
- 6.NoSuchFieldException A requested field does not exist.
- 7.NoSuchMethodException A requested method does not exist.
- 8.NullPointerException Invalid use of a null reference.
- 9.NumberFormatException Invalid conversion of a string to a numeric format.
- 10.SecurityException Attempt to violate security.
- 11.StringIndexOutOfBoundsException Attempt to index outside the bounds of a string.
- 12.UnsupportedOperationException An unsupported operation was encountered.

Creating Your Own Exception Subclasses:

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString()** method, allowing the description of the exception to be displayed using **println()**.

// This program creates a custom exception type.

```
class MyException extends Exception
{
private int detail;
MyException(int a)
```

```

    {
    detail = a;
    }
    public String toString()
    {
    return "MyException[" + detail + "]";
    }
    }
    class ExceptionDemo
    {
    static void compute(int a) throws MyException
    {
    System.out.println("Called compute(" + a + ")");

    if(a > 10)
    throw new MyException(a);
    System.out.println("Normal exit");
    }
    public static void main(String args[])
    {
    try {
    compute(1);
    compute(20);
    } catch (MyException e)
    {
    System.out.println("Caught " + e);
    }}}

```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: it has only a constructor plus an overloaded **toString()** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute()** that throws a **MyException** object. The exception is thrown when **compute()**'s integer parameter is greater than 10. The **main()** method sets up an exception handler for **MyException**, then calls **compute()** with a legal value (less than 10) and an illegal one to show both paths through the code.

Output:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]

MULTITHREADING

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a *thread*, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

There are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two.

- ✚ process-based multitasking is the more familiar form. A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- ✚ In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the “big picture,” and thread-based multitasking handles the details.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a traditional, single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

The Java Thread Model:

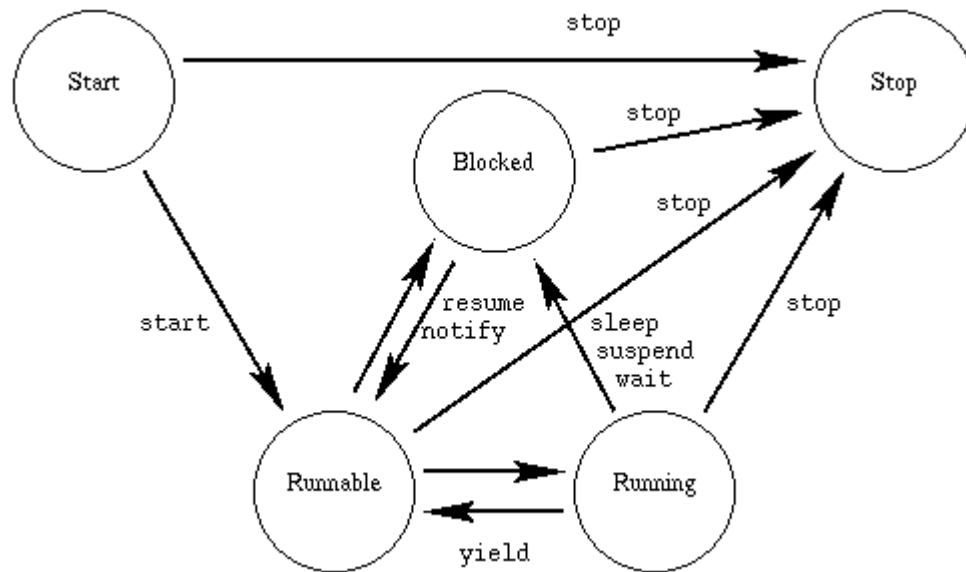
The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java’s multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Threads exist in several states. A thread can be *running*. It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended*, which temporarily suspends its

activity. A suspended thread can then be *resumed*, allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.



Thread Priorities:

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

- *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- *A thread can be pre-empted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

Synchronization:

Because multithreading introduces an asynchronous behavior to the programs, there must be a way for you to enforce synchronicity when you need it. For example, if we want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, we must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor*. The monitor is a

control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Messaging:

we divide the program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

The Thread Class and the Runnable Interface:

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since we can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.

The **Thread** class defines several methods that help manage threads.

Method Meaning

- 1.getName Obtain a thread's name.
- 2.getPriority Obtain a thread's priority.
- 3.isAlive Determine if a thread is still running.
- 4.join Wait for a thread to terminate.
- 5.run Entry point for the thread.
- 6.sleep Suspend a thread for a period of time.
- 7.start Start a thread by calling its run method.

The Main Thread:

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**.

Its general form is shown here:

```
static Thread currentThread()
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

example:

```
// Controlling the main Thread.  
class CurrentThreadDemo  
{  
public static void main(String args[])  
{
```



```

Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
Try
{
for(int n = 5; n > 0; n--)
{
System.out.println(n);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println("Main thread interrupted");
}
}
}

```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop.

The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently.

output:

```

Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1

```

Notice the output produced when **t** is used as an argument to **println()**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment and is not discussed in detail here.

After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

The **sleep()** method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of milliseconds to suspend is specified in *milliseconds*. This method may throw an **InterruptedException**.

The **sleep()** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException

This second form is useful only in environments that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName()**. You can obtain the name of a thread by calling **getName()** (but note that this procedure is not shown in the program). These methods are members of the **Thread** class and are declared like this:

final void setName(String *threadName*)

final String getName()

Here, *threadName* specifies the name of the thread.

Creating a Thread:

We create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this:

public void run()

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After we create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** executes a call to **run()**.

The **start()** method is shown here:

void start()

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        try {
```



```

for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
}
System.out.println("Exiting child thread.");
}
}
}
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
}
System.out.println("Main thread exiting.");
}
}
}

```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. After calling **start()**, **NewThread**'s constructor returns to **main()**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
```

```
Main Thread: 5
```

```
Child Thread: 5
```

```
Child Thread: 4
```

```
Main Thread: 4
```

```
Child Thread: 3
```

```
Child Thread: 2
```

```
Main Thread: 3
```

```
Child Thread: 1
```

```
Exiting child thread.
```

```
Main Thread: 2
```

```
Main Thread: 1
```

```
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may “hang.” The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to

terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread:

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run()** method, which is the entry point for the new thread. It must also call **start()** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }
    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super()** inside **NewThread**. This invokes the following form of the **Thread** constructor:

```
public Thread(String threadName)
```

Here, *threadName* specifies the name of the thread.

Creating Multiple Threads:

We have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
    String name; // name of thread
    Thread t;
    NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
    }
    // This is the entry point for thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
    }
}

class MultiThreadDemo {
    public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
            // wait for other threads to end
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
        }
        System.out.println("Main thread exiting.");
    }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
```

Three: 4

One: 3

Three: 3

Two: 3

One: 2

Three: 2

Two: 2

One: 1

Three: 1

Two: 1

One exiting.

Two exiting.

Three exiting.

Main thread exiting.

once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main()**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive() and join():

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep()** within **main()**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive()** on the thread. This method is defined by **Thread**, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive()** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive()** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join()**, shown here:

```
final void join() throws InterruptedException
```

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it.

Additional forms of **join()** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Thread Priorities:

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.

threads of equal priority should get equal access to the CPU. threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying

on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally, so that other threads can run.

- To set a thread's priority, use the **setPriority()** method, which is a member of **Thread**.

This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **final** variables within **Thread**.

- You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here:

```
final int getPriority()
```

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex*. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run()** method. The thread is started immediately. The **run()** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

Example:

// This program is not synchronized.

```
class Callme {
void call(String msg) {
```

```

System.out.print "[" + msg);
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}
class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here is the output produced by this program:

```

Hello[Synchronized[World]
]
]

```

As you can see, by calling **sleep()**, the **call()** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep()** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must *serialize* access to **call()**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call()**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
    synchronized void call(String msg) {
```

```
...
```

This prevents other threads from entering **call()** while another thread is using it. After **synchronized** has been added to **call()**, the output of the program is as follows:

```
[Hello]
```

```
[Synchronized]
```

```
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {
    // statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run()** method:

```
// This program uses a synchronized block.
```

```
class Callme {
    void call(String msg) {
        System.out.print("[ " + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
```



```

public Caller(Callme targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
}
// synchronize calls to call()
public void run() {
    synchronized(target) { // synchronized block
        target.call(msg);
    }
}
}
class Synch1 {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Here, the **call()** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller's run()** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication:

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. multithreading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.

For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a

synchronized context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.

- **notify()** wakes up the first thread that called **wait()** on the same object.

- **notifyAll()** wakes up all the threads that called **wait()** on the same object.

The highest priority thread will run first.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException
```

```
final void notify()
```

```
final void notifyAll()
```

Additional forms of **wait()** exist that allow you to specify a period of time to wait.

The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

// An incorrect implementation of a producer and consumer.

```
class Q {
    int n;
    synchronized int get() {
        System.out.println("Got: " + n);
        return n;
    }
    synchronized void put(int n) {
        this.n = n;
        System.out.println("Put: " + n);
    }
}

class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}

class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
```

```

}
}
}
class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}

```

Although the **put()** and **get()** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```

Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7

```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them. The proper way to write this program in Java is to use **wait()** and **notify()** to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.

```

class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {

```

```

wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}

```

Inside **get()**, **wait()** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get()** resumes. After the data has been obtained, **get()** calls **notify()**. This tells **Producer** that it is okay to put more data in the queue. Inside **put()**, **wait()** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify()** is called. This tells the **Consumer** that it should now remove it.

Output:

Put: 1

Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5

EXPLORING JAVA.UTIL PACKAGE:

The java.util package contains one of Java's most powerful subsystems: collections.

1. StringTokenizer

The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens*, which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner*. **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using **StringTokenizer**.

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, “,:;” sets the delimiters to a comma, semicolon, and colon.

The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.

The **StringTokenizer** constructors are shown here:

`StringTokenizer(String str)`

`StringTokenizer(String str, String delimiters)`

`StringTokenizer(String str, String delimiters, boolean delimAsToken)`

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken()** method is used to extract consecutive tokens. The **hasMoreTokens()** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements()** and **nextElement()** methods are also implemented, and they act the same as **hasMoreTokens()** and **nextToken()**, respectively.

2. Date

The **Date** class encapsulates the current date and time.

Date supports the following constructors:

Date()

Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

3. Random

The **Random** class is a generator of pseudorandom numbers. These are called *pseudorandom* numbers because they are simply uniformly distributed sequences. **Random** defines the following constructors:

Random()

Random(long *seed*)

The first version creates a number generator that uses the current time as the starting, or *seed*, value. The second form allows you to specify a seed value manually. If you initialize a Random object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another Random object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. The easiest way to do this is to use the current time to seed a Random object. This approach reduces the possibility of getting repeated sequences.

Some of the methods in this class are:

- i. boolean nextBoolean(): Returns the next boolean random number.
- ii. void nextBytes(byte *vals*[]): Fills *vals* with randomly generated values.
- iii. double nextDouble(): Returns the next double random number.
- iv. float nextFloat(): Returns the next float random number.
- v. double nextGaussian(): Returns the next Gaussian random number.
- vi. int nextInt(): Returns the next int random number.
- vii. int nextInt(int *n*): Returns next int random number within the range zero to *n*.
- viii. long nextLong(): Returns the next long random number.

UNIT-IV

COLLECTIONS IN JAVA

Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used **Vector** was different from the way that you used **Properties**.

The collections framework was designed to meet several goals, such as –

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

Towards this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList**, **HashSet**, and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following –

- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

THE COLLECTION INTERFACES

The collections framework defines several interfaces. This section provides an overview of each interface –

Sr.No.	Interface & Description
--------	-------------------------

- | | |
|---|--|
| 1 | The Collection Interface |
|---|--|

This enables you to work with groups of objects; it is at the top of the collections hierarchy.

2 [The List Interface](#)

This extends **Collection** and an instance of List stores an ordered collection of elements.

3 [The Set](#)

This extends Collection to handle sets, which must contain unique elements.

4 [The SortedSet](#)

This extends Set to handle sorted sets.

5 [The Map](#)

This maps unique keys to values.

6 [The Map.Entry](#)

This describes an element (a key/value pair) in a map. This is an inner class of Map.

7 [The SortedMap](#)

This extends Map so that the keys are maintained in an ascending order.

[The Enumeration](#)

8 This is legacy interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by Iterator.

THE COLLECTION CLASSES

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table –

Sr.No.	Class & Description
1	AbstractCollection Implements most of the Collection interface.
2	AbstractList Extends AbstractCollection and implements most of the List interface.
3	AbstractSequentialList Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.

4 [LinkedList](#)

Implements a linked list by extending `AbstractSequentialList`.

5 [ArrayList](#)

Implements a dynamic array by extending `AbstractList`.

6 **AbstractSet**

Extends `AbstractCollection` and implements most of the `Set` interface.

7 [HashSet](#)

Extends `AbstractSet` for use with a hash table.

8 [LinkedHashSet](#)

Extends `HashSet` to allow insertion-order iterations.

9 [TreeSet](#)

Implements a set stored in a tree. Extends `AbstractSet`.

10 **AbstractMap**

Implements most of the `Map` interface.

11 [HashMap](#)

Extends `AbstractMap` to use a hash table.

12 [TreeMap](#)

Extends `AbstractMap` to use a tree.

13 [WeakHashMap](#)

Extends `AbstractMap` to use a hash table with weak keys.

14 [LinkedHashMap](#)

Extends `HashMap` to allow insertion-order iterations.

15 [IdentityHashMap](#)

Extends `AbstractMap` and uses reference equality when comparing documents.

The *AbstractCollection*, *AbstractSet*, *AbstractList*, *AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by `java.util` have been discussed in the previous chapter –

Sr.No.	Class & Description
--------	---------------------

1 [Vector](#)

This implements a dynamic array. It is similar to `ArrayList`, but with some differences.

2 [Stack](#)

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

3 [Dictionary](#)

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

4 [Hashtable](#)

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

5 [Properties](#)

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

6 [BitSet](#)

A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

THE COLLECTION ALGORITHMS

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections define three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

Sr.No.	Algorithm & Description
--------	-------------------------

1 [The Collection Algorithms](#)

Here is a list of all the algorithm implementation.

HOW TO USE AN ITERATOR ?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list and the modification of elements.

Sr.No.	Iterator Method & Description
--------	-------------------------------

[Using Java Iterator](#)

1

Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

JAVA ARRAYLIST CLASS

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

HIERARCHY OF ARRAYLIST CLASS

As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order.

ARRAYLIST CLASS DECLARATION

Let's see the old non-generic example of creating java collection.

1. `ArrayList al=new ArrayList();`//creating old non-generic arraylist

Let's see the new generic example of creating java collection.

1. `ArrayList<String> al=new ArrayList<String>();`//creating new generic arraylist

In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*.

JAVA ARRAYLIST EXAMPLE

1. `import java.util.*;`
2. `class TestCollection1{`
3. `public static void main(String args[]){`

```
4.  ArrayList<String> list=new ArrayList<String>();//Creating arraylist
5.  list.add("Ravi");//Adding object in arraylist
6.  list.add("Vijay");
7.  list.add("Ravi");
8.  list.add("Ajay");
9.  //Traversing list through Iterator
10. Iterator itr=list.iterator();
11. while(itr.hasNext()){
12.  System.out.println(itr.next());
13. }
14. }
15. }
```

[Test it Now](#)

```
Ravi
Vijay
Ravi
Ajay
```

TWO WAYS TO ITERATE THE ELEMENTS OF COLLECTION IN JAVA

There are two ways to traverse collection elements:

1. By Iterator interface.
2. By for-each loop.

In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop.

ITERATING COLLECTION THROUGH FOR-EACH LOOP

```
1.  import java.util.*;
2.  class TestCollection2{
3.  public static void main(String args[]){
4.  ArrayList<String> al=new ArrayList<String>();
5.  al.add("Ravi");
6.  al.add("Vijay");
7.  al.add("Ravi");
8.  al.add("Ajay");
9.  for(String obj:al)
10.  System.out.println(obj);
11. }
12. }
```

[Test it Now](#)

```
Ravi
Vijay
Ravi
Ajay
```

USER-DEFINED CLASS OBJECTS IN JAVA ARRAYLIST

Let's see an example where we are storing Student class object in array list.

```
1. class Student{
2.   int rollno;
3.   String name;
4.   int age;
5.   Student(int rollno,String name,int age){
6.     this.rollno=rollno;
7.     this.name=name;
8.     this.age=age;
9.   }
10. }

1. import java.util.*;
2. public class TestCollection3{
3.   public static void main(String args[]){
4.     //Creating user-defined class objects
5.     Student s1=new Student(101,"Sonoo",23);
6.     Student s2=new Student(102,"Ravi",21);
7.     Student s3=new Student(103,"Hanumat",25);
8.     //creating arraylist
9.     ArrayList<Student> al=new ArrayList<Student>();
10.    al.add(s1);//adding Student class object
11.    al.add(s2);
12.    al.add(s3);
13.    //Getting Iterator
14.    Iterator itr=al.iterator();
15.    //traversing elements of ArrayList object
16.    while(itr.hasNext()){
17.      Student st=(Student)itr.next();
18.      System.out.println(st.rollno+" "+st.name+" "+st.age);
19.    }
20.  }
21. }
```

Test it Now

```
101 Sonoo 23
102 Ravi 21
103 Hanumat 25
```

EXAMPLE OF ADDALL(COLLECTION C) METHOD

```
1. import java.util.*;
2. class TestCollection4{
3.   public static void main(String args[]){
4.     ArrayList<String> al=new ArrayList<String>();
5.     al.add("Ravi");
6.     al.add("Vijay");
7.     al.add("Ajay");
8.     ArrayList<String> al2=new ArrayList<String>();
9.     al2.add("Sonoo");
10.    al2.add("Hanumat");
11.    al.addAll(al2);//adding second list in first list
```

```
12. Iterator itr=al.iterator();
13. while(itr.hasNext()){
14.     System.out.println(itr.next());
15. }
16. }
17. }
```

[Test it Now](#)

```
Ravi
Vijay
Ajay
Sonoo
Hanumat
```

EXAMPLE OF REMOVEALL() METHOD

```
1. import java.util.*;
2. class TestCollection5{
3.     public static void main(String args[]){
4.         ArrayList<String> al=new ArrayList<String>();
5.         al.add("Ravi");
6.         al.add("Vijay");
7.         al.add("Ajay");
8.         ArrayList<String> al2=new ArrayList<String>();
9.         al2.add("Ravi");
10.        al2.add("Hanumat");
11.        al.removeAll(al2);
12.        System.out.println("iterating the elements after removing the elements of al2...");
13.        Iterator itr=al.iterator();
14.        while(itr.hasNext()){
15.            System.out.println(itr.next());
16.        }
17.
18.    }
19. }
```

[Test it Now](#)

```
iterating the elements after removing the elements of al2...
Vijay
Ajay
```

EXAMPLE OF RETAINALL() METHOD

```
1. import java.util.*;
2. class TestCollection6{
3.     public static void main(String args[]){
4.         ArrayList<String> al=new ArrayList<String>();
5.         al.add("Ravi");
6.         al.add("Vijay");
7.         al.add("Ajay");
8.         ArrayList<String> al2=new ArrayList<String>();
```

```
9.  al2.add("Ravi");
10. al2.add("Hanumat");
11. al.retainAll(al2);
12. System.out.println("iterating the elements after retaining the elements of al2...");
13. Iterator itr=al.iterator();
14. while(itr.hasNext()){
15.  System.out.println(itr.next());
16. }
17. }
18. }
```

[Test it Now](#)

```
iterating the elements after retaining the elements of al2...
Ravi
```

JAVA ARRAYLIST EXAMPLE: BOOK

Let's see an ArrayList example where we are adding books to list and printing all the books.

```
1.  import java.util.*;
2.  class Book {
3.  int id;
4.  String name,author,publisher;
5.  int quantity;
6.  public Book(int id, String name, String author, String publisher, int quantity) {
7.    this.id = id;
8.    this.name = name;
9.    this.author = author;
10.   this.publisher = publisher;
11.   this.quantity = quantity;
12. }
13. }
14. public class ArrayListExample {
15. public static void main(String[] args) {
16.  //Creating list of Books
17.  List<Book> list=new ArrayList<Book>();
18.  //Creating Books
19.  Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.  Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hil
   l",4);
21.  Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.  //Adding Books to list
23.  list.add(b1);
24.  list.add(b2);
25.  list.add(b3);
26.  //Traversing list
27.  for(Book b:list){
28.    System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29.  }
30. }
31. }
```

[Test it Now](#)

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

JAVA LINKEDLIST CLASS

Java LinkedList class uses doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to be occurred.
- Java LinkedList class can be used as list, stack or queue.

HIERARCHY OF LINKEDLIST CLASS

As shown in above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

DOUBLY LINKED LIST

In case of doubly linked list, we can add or remove elements from both side.

LINKEDLIST CLASS DECLARATION

Let's see the declaration for java.util.LinkedList class.

1. `public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable, Serializable`

CONSTRUCTORS OF JAVA LINKEDLIST

Constructor	Description
LinkedList()	It is used to construct an empty list.

LinkedList(Collection c) It is used to construct a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

METHODS OF JAVA LINKEDLIST

Method	Description
void add(int index, Object element)	It is used to insert the specified element at the specified position index in a list.
void addFirst(Object o)	It is used to insert the given element at the beginning of a list.
void addLast(Object o)	It is used to append the given element to the end of a list.
int size()	It is used to return the number of elements in a list
boolean add(Object o)	It is used to append the specified element to the end of a list.
boolean contains(Object o)	It is used to return true if the list contains a specified element.
boolean remove(Object o)	It is used to remove the first occurrence of the specified element in a list.
Object getFirst()	It is used to return the first element in a list.
Object getLast()	It is used to return the last element in a list.
int indexOf(Object o)	It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element.
int lastIndexOf(Object o)	It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element.

JAVA LINKEDLIST EXAMPLE

```
import java.util.*;
public class TestCollection7{
    public static void main(String args[]){

        LinkedList<String> al=new LinkedList<String>();
        al.add("Ravi");
        al.add("Vijay");
        al.add("Ravi");
        al.add("Ajay");

        Iterator<String> itr=al.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```



```

    }
  }
}

```

[Test it Now](#)

```

Output:Ravi
       Vijay
       Ravi
       Ajay

```

JAVA LINKEDLIST EXAMPLE: BOOK

```

1.  import java.util.*;
2.  class Book {
3.  int id;
4.  String name,author,publisher;
5.  int quantity;
6.  public Book(int id, String name, String author, String publisher, int quantity) {
7.      this.id = id;
8.      this.name = name;
9.      this.author = author;
10.     this.publisher = publisher;
11.     this.quantity = quantity;
12. }
13. }
14. public class LinkedListExample {
15. public static void main(String[] args) {
16.     //Creating list of Books
17.     List<Book> list=new LinkedList<Book>();
18.     //Creating Books
19.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20.     Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hil
l",4);
21.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
22.     //Adding Books to list
23.     list.add(b1);
24.     list.add(b2);
25.     list.add(b3);
26.     //Traversing list
27.     for(Book b:list){
28.         System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29.     }
30. }
31. }

```

Output:

```

101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6

```

JAVA HASHSET CLASS

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing**.
- HashSet contains unique elements only.

DIFFERENCE BETWEEN LIST AND SET

List can contain duplicate elements whereas Set contains unique elements only.

HIERARCHY OF HASHSET CLASS

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

HASHSET CLASS DECLARATION

Let's see the declaration for java.util.HashSet class.

1. `public class HashSet<E> extends AbstractSet<E> implements Set<E>, Cloneable, Serializable`

CONSTRUCTORS OF JAVA HASHSET CLASS:

Constructor	Description
HashSet()	It is used to construct a default HashSet.
HashSet(Collection c)	It is used to initialize the hash set by using the elements of the collection c.
HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.

METHODS OF JAVA HASHSET CLASS:

Method	Description
<code>void clear()</code>	It is used to remove all of the elements from this set.
<code>boolean contains(Object o)</code>	It is used to return true if this set contains the specified element.
<code>boolean add(Object o)</code>	It is used to adds the specified element to this set if it is not already present.
<code>boolean isEmpty()</code>	It is used to return true if this set contains no elements.
<code>boolean remove(Object o)</code>	It is used to remove the specified element from this set if it is present.
<code>Object clone()</code>	It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned.
<code>Iterator iterator()</code>	It is used to return an iterator over the elements in this set.
<code>int size()</code>	It is used to return the number of elements in this set.

JAVA HASHSET EXAMPLE

```

1. import java.util.*;
2. class TestCollection9{
3.     public static void main(String args[]){
4.         //Creating HashSet and adding elements
5.         HashSet<String> set=new HashSet<String>();
6.         set.add("Ravi");
7.         set.add("Vijay");
8.         set.add("Ravi");
9.         set.add("Ajay");
10.        //Traversing elements
11.        Iterator<String> itr=set.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }

```

[Test it Now](#)

```

Ajay
Vijay
Ravi

```

JAVA HASHSET EXAMPLE: BOOK

Let's see a HashSet example where we are adding books to set and printing all the books.

```

1. import java.util.*;
II B.tech II Semester(CSE)

```

```
2. class Book {
3.   int id;
4.   String name,author,publisher;
5.   int quantity;
6.   public Book(int id, String name, String author, String publisher, int quantity) {
7.     this.id = id;
8.     this.name = name;
9.     this.author = author;
10.    this.publisher = publisher;
11.    this.quantity = quantity;
12.  }
13. }
14. public class HashSetExample {
15.   public static void main(String[] args) {
16.     HashSet<Book> set=new HashSet<Book>();
17.     //Creating Books
18.     Book b1=new Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
19.     Book b2=new Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hil
l",4);
20.     Book b3=new Book(103,"Operating System","Galvin","Wiley",6);
21.     //Adding Books to HashSet
22.     set.add(b1);
23.     set.add(b2);
24.     set.add(b3);
25.     //Traversing HashSet
26.     for(Book b:set){
27.       System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
28.     }
29.   }
30. }
```

Output:

```
101 Let us C Yashwant Kanetkar BPB 8
102 Data Communications & Networking Forouzan Mc Graw Hill 4
103 Operating System Galvin Wiley 6
```

JAVA TREESSET CLASS

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

- Contains unique elements only like HashSet.
- Access and retrieval times are quiet fast.

- Maintains ascending order.

HIERARCHY OF TREESET CLASS

As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

TREESET CLASS DECLARATION

Let's see the declaration for java.util.TreeSet class.

1. `public class TreeSet<E> extends AbstractSet<E> implements NavigableSet<E>, Cloneable, Serializable`

CONSTRUCTORS OF JAVA TREESET CLASS

Constructor	Description
<code>TreeSet()</code>	It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.
<code>TreeSet(Collection c)</code>	It is used to build a new tree set that contains the elements of the collection c.
<code>TreeSet(Comparator comp)</code>	It is used to construct an empty tree set that will be sorted according to given comparator.
<code>TreeSet(SortedSet ss)</code>	It is used to build a TreeSet that contains the elements of the given SortedSet.

METHODS OF JAVA TREESET CLASS

Method	Description
<code>boolean addAll(Collection c)</code>	It is used to add all of the elements in the specified collection to this set.
<code>boolean contains(Object o)</code>	It is used to return true if this set contains the specified element.
<code>boolean isEmpty()</code>	It is used to return true if this set contains no elements.
<code>boolean remove(Object o)</code>	It is used to remove the specified element from this set if it is present.
<code>void add(Object o)</code>	It is used to add the specified element to this set if it is not already present.

<code>void clear()</code>	It is used to remove all of the elements from this set.
<code>Object clone()</code>	It is used to return a shallow copy of this TreeSet instance.
<code>Object first()</code>	It is used to return the first (lowest) element currently in this sorted set.
<code>Object last()</code>	It is used to return the last (highest) element currently in this sorted set.
<code>int size()</code>	It is used to return the number of elements in this set.

JAVA TREESSET EXAMPLE

```
1. import java.util.*;
2. class TestCollection11{
3.     public static void main(String args[]){
4.         //Creating and adding elements
5.         TreeSet<String> al=new TreeSet<String>();
6.         al.add("Ravi");
7.         al.add("Vijay");
8.         al.add("Ravi");
9.         al.add("Ajay");
10.        //Traversing elements
11.        Iterator<String> itr=al.iterator();
12.        while(itr.hasNext()){
13.            System.out.println(itr.next());
14.        }
15.    }
16. }
```

[Test it Now](#)

Output:

```
Ajay
Ravi
Vijay
```

JAVA TREESSET EXAMPLE: BOOK

Let's see a TreeSet example where we are adding books to set and printing all the books. The elements in TreeSet must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in TreeSet, you need to implement Comparable interface.

```
1. import java.util.*;
2. class Book implements Comparable<Book>{
3.     int id;
4.     String name,author,publisher;
5.     int quantity;
6.     public Book(int id, String name, String author, String publisher, int quantity) {
II B.tech II Semester(CSE)                110                Object Oriented Programming
```

```
7.    this.id = id;
8.    this.name = name;
9.    this.author = author;
10.   this.publisher = publisher;
11.   this.quantity = quantity;
12. }
13. public int compareTo(Book b) {
14.     if(id>b.id){
15.         return 1;
16.     }else if(id<b.id){
17.         return -1;
18.     }else{
19.         return 0;
20.     }
21. }
22. }
23. public class TreeSetExample {
24.     public static void main(String[] args) {
25.         Set<Book> set=new TreeSet<Book>();
26.         //Creating Books
27.         Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28.         Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
29.         Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hil
l",4);
30.         //Adding Books to TreeSet
31.         set.add(b1);
32.         set.add(b2);
33.         set.add(b3);
34.         //Traversing TreeSet
35.         for(Book b:set){
36.             System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37.         }
38.     }
39. }
```

Output:

```
101 Data Communications & Networking Forouzan Mc Graw Hill 4
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6
```

JAVA QUEUE INTERFACE

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

QUEUE INTERFACE DECLARATION

```
1. public interface Queue<E> extends Collection<E>
```

METHODS OF JAVA QUEUE INTERFACE

Method	Description
boolean add(object)	It is used to insert the specified element into this queue and return true upon success.
boolean offer(object)	It is used to insert the specified element into this queue.
Object remove()	It is used to retrieves and removes the head of this queue.
Object poll()	It is used to retrieves and removes the head of this queue, or returns null if this queue is empty.
Object element()	It is used to retrieves, but does not remove, the head of this queue.
Object peek()	It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

PRIORITYQUEUE CLASS

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner. It inherits AbstractQueue class.

PRIORITYQUEUE CLASS DECLARATION

Let's see the declaration for java.util.PriorityQueue class.

1. public class PriorityQueue<E> extends AbstractQueue<E> implements Serializable

JAVA PRIORITYQUEUE EXAMPLE

```

1. import java.util.*;
2. class TestCollection12{
3.     public static void main(String args[]){
4.         PriorityQueue<String> queue=new PriorityQueue<String>();
5.         queue.add("Amit");
6.         queue.add("Vijay");
7.         queue.add("Karan");
8.         queue.add("Jai");
9.         queue.add("Rahul");
10.        System.out.println("head:"+queue.element());
11.        System.out.println("head:"+queue.peek());
12.        System.out.println("iterating the queue elements:");
13.        Iterator itr=queue.iterator();
14.        while(itr.hasNext()){
15.            System.out.println(itr.next());
16.        }

```



```

17. queue.remove();
18. queue.poll();
19. System.out.println("after removing two elements:");
20. Iterator<String> itr2=queue.iterator();
21. while(itr2.hasNext()){
22. System.out.println(itr2.next());
23. }
24. }
25. }

```

[Test it Now](#)

```

Output:head:Amit
      head:Amit
      iterating the queue elements:
      Amit
      Jai
      Karan
      Vijay
      Rahul
      after removing two elements:
      Karan
      Rahul
      Vijay

```

JAVA PRIORITYQUEUE EXAMPLE: BOOK

Let's see a PriorityQueue example where we are adding books to queue and printing all the books. The elements in PriorityQueue must be of Comparable type. String and Wrapper classes are Comparable by default. To add user-defined objects in PriorityQueue, you need to implement Comparable interface.

```

1. import java.util.*;
2. class Book implements Comparable<Book>{
3. int id;
4. String name,author,publisher;
5. int quantity;
6. public Book(int id, String name, String author, String publisher, int quantity) {
7.     this.id = id;
8.     this.name = name;
9.     this.author = author;
10.    this.publisher = publisher;
11.    this.quantity = quantity;
12. }
13. public int compareTo(Book b) {
14.    if(id>b.id){
15.        return 1;
16.    }else if(id<b.id){
17.        return -1;
18.    }else{
19.        return 0;
20.    }
21. }
22. }
23. public class LinkedListExample {
24. public static void main(String[] args) {

```

```

25. Queue<Book> queue=new PriorityQueue<Book>();
26. //Creating Books
27. Book b1=new Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=new Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=new Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hil
    l",4);
30. //Adding Books to the queue
31. queue.add(b1);
32. queue.add(b2);
33. queue.add(b3);
34. System.out.println("Traversing the queue elements:");
35. //Traversing queue elements
36. for(Book b:queue){
37.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
38. }
39. queue.remove();
40. System.out.println("After removing one book record:");
41. for(Book b:queue){
42.     System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
43. }
44. }
45. }

```

Output:

```

Traversing the queue elements:
101 Data Communications & Networking Forouzan Mc Graw Hill 4
233 Operating System Galvin Wiley 6
121 Let us C Yashwant Kanetkar BPB 8
After removing one book record:
121 Let us C Yashwant Kanetkar BPB 8
233 Operating System Galvin Wiley 6

```

JAVA DEQUE INTERFACE

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for "**double ended queue**".

DEQUE INTERFACE DECLARATION

1. public interface Deque<E> extends Queue<E>

METHODS OF JAVA DEQUE INTERFACE

Method	Description
boolean add(object)	It is used to insert the specified element into this deque and return true upon success.

boolean offer(object)	It is used to insert the specified element into this deque.
Object remove()	It is used to retrieves and removes the head of this deque.
Object poll()	It is used to retrieves and removes the head of this deque, or returns null if this deque is empty.
Object element()	It is used to retrieves, but does not remove, the head of this deque.
Object peek()	It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty.

ARRAYDEQUE CLASS

The ArrayDeque class provides the facility of using deque and resizable-array. It inherits AbstractCollection class and implements the Deque interface.

The important points about ArrayDeque class are:

- Unlike Queue, we can add or remove elements from both sides.
- Null elements are not allowed in the ArrayDeque.
- ArrayDeque is not thread safe, in the absence of external synchronization.
- ArrayDeque has no capacity restrictions.
- ArrayDeque is faster than LinkedList and Stack.

ARRAYDEQUE HIERARCHY

The hierarchy of ArrayDeque class is given in the figure displayed at the right side of the page.

ARRAYDEQUE CLASS DECLARATION

Let's see the declaration for java.util.ArrayDeque class.

1. public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable

JAVA ARRAYDEQUE EXAMPLE

1. import java.util.*;
2. public class ArrayDequeExample {
3. public static void main(String[] args) {
4. //Creating Deque and adding elements
5. Deque<String> deque = new ArrayDeque<String>();

```
6.  deque.add("Ravi");
7.  deque.add("Vijay");
8.  deque.add("Ajay");
9.  //Traversing elements
10. for (String str : deque) {
11.  System.out.println(str);
12. }
13. }
14. }
```

Output:

```
Ravi
Vijay
Ajay
```

JAVA ARRAYDEQUE EXAMPLE: OFFERFIRST() AND POLLLAST()

```
1.  import java.util.*;
2.  public class DequeExample {
3.  public static void main(String[] args) {
4.    Deque<String> deque=new ArrayDeque<String>();
5.    deque.offer("arvind");
6.    deque.offer("vimal");
7.    deque.add("mukul");
8.    deque.offerFirst("jai");
9.    System.out.println("After offerFirst Traversal...");
10.   for(String s:deque){
11.     System.out.println(s);
12.   }
13.   //deque.poll();
14.   //deque.pollFirst();//it is same as poll()
15.   deque.pollLast();
16.   System.out.println("After pollLast() Traversal...");
17.   for(String s:deque){
18.     System.out.println(s);
19.   }
20. }
21. }
```

Output:

```
After offerFirst Traversal...
jai
arvind
vimal
mukul
After pollLast() Traversal...
jai
arvind
vimal
```

Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. The easiest way to do this is to employ an iterator, which is an object that implements either the `Iterator` or the `ListIterator` interface.

`Iterator` enables you to cycle through a collection, obtaining or removing elements. `ListIterator` extends `Iterator` to allow bidirectional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an `iterator()` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps –

- Obtain an iterator to the start of the collection by calling the collection's `iterator()` method.
- Set up a loop that makes a call to `hasNext()`. Have the loop iterate as long as `hasNext()` returns true.
- Within the loop, obtain each element by calling `next()`.

For collections that implement `List`, you can also obtain an iterator by calling `ListIterator`.

THE METHODS DECLARED BY ITERATOR

Sr.No.	Method & Description
1	boolean hasNext() Returns true if there are more elements. Otherwise, returns false.
2	Object next() Returns the next element. Throws <code>NoSuchElementException</code> if there is not a next element.
3	void remove() Removes the current element. Throws <code>IllegalStateException</code> if an attempt is made to call <code>remove()</code> that is not preceded by a call to <code>next()</code> .

THE METHODS DECLARED BY LISTITERATOR

Sr.No.	Method & Description
1	void add(Object obj) Inserts <code>obj</code> into the list in front of the element that will be returned by the next call to <code>next()</code> .

2 **boolean hasNext()**

Returns true if there is a next element. Otherwise, returns false.

3 **boolean hasPrevious()**

Returns true if there is a previous element. Otherwise, returns false.

4 **Object next()**

Returns the next element. A NoSuchElementException is thrown if there is not a next element.

5 **int nextIndex()**

Returns the index of the next element. If there is not a next element, returns the size of the list.

6 **Object previous()**

Returns the previous element. A NoSuchElementException is thrown if there is not a previous element.

7 **int previousIndex()**

Returns the index of the previous element. If there is not a previous element, returns -1.

8 **void remove()**

Removes the current element from the list. An IllegalStateException is thrown if remove() is called before next() or previous() is invoked.

9 **void set(Object obj)**

Assigns obj to the current element. This is the element last returned by a call to either next() or previous().

EXAMPLE

Here is an example demonstrating both Iterator and ListIterator. It uses an ArrayList object, but the general principles apply to any type of collection.

Of course, ListIterator is available only to those collections that implement the List interface.

```
import java.util.*;
public class IteratorDemo {

    public static void main(String args[]) {
        // Create an array list
        ArrayList al = new ArrayList();

        // add elements to the array list
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
    }
}
```

```
// Use iterator to display contents of al
System.out.print("Original contents of al: ");
Iterator itr = al.iterator();

while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
System.out.println();

// Modify objects being iterated
ListIterator litr = al.listIterator();

while(litr.hasNext()) {
    Object element = litr.next();
    litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();

while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
System.out.println();

// Now, display the list backwards
System.out.print("Modified list backwards: ");

while(litr.hasPrevious()) {
    Object element = litr.previous();
    System.out.print(element + " ");
}
System.out.println();
}
```

This will produce the following result –

OUTPUT

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

JAVA COMPARATOR INTERFACE

Java Comparator interface is used to order the objects of user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element).

It provides multiple sorting sequence i.e. you can sort the elements on the basis of any data member, for example rollno, name, age or anything else.

COMPARE() METHOD

public int compare(Object obj1, Object obj2): compares the first object with second object.

COLLECTIONS CLASS

Collections class provides static methods for sorting the elements of collection. If collection elements are of Set or Map, we can use TreeSet or TreeMap. But we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements also.

METHOD OF COLLECTIONS CLASS FOR SORTING LIST ELEMENTS

public void sort(List list, Comparator c): is used to sort the elements of List by the given Comparator.

JAVA COMPARATOR EXAMPLE (NON-GENERIC OLD STYLE)

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

1. Student.java
2. AgeComparator.java
3. NameComparator.java
4. Simple.java

Student.java

This class contains three fields rollno, name and age and a parameterized constructor.

1. class Student{
2. int rollno;
3. String name;
4. int age;
5. Student(int rollno,String name,int age){
6. this.rollno=rollno;
7. this.name=name;
8. this.age=age;
9. }
10. }

AgeComparator.java

This class defines comparison logic based on the age. If age of first object is greater than the second, we are returning positive value, it can be any one such as 1, 2, 10 etc. If age of first object is less than the second object, we are returning negative value, it can be any negative value and if age of both objects are equal, we are returning 0.

```
1. import java.util.*;
2. class AgeComparator implements Comparator{
3.     public int compare(Object o1, Object o2){
4.         Student s1=(Student)o1;
5.         Student s2=(Student)o2;
6.
7.         if(s1.age==s2.age)
8.             return 0;
9.         else if(s1.age>s2.age)
10.            return 1;
11.        else
12.            return -1;
13.    }
14. }
```

NameComparator.java

This class provides comparison logic based on the name. In such case, we are using the compareTo() method of String class, which internally provides the comparison logic.

```
1. import java.util.*;
2. class NameComparator implements Comparator{
3.     public int compare(Object o1, Object o2){
4.         Student s1=(Student)o1;
5.         Student s2=(Student)o2;
6.
7.         return s1.name.compareTo(s2.name);
8.     }
9. }
```

Simple.java

In this class, we are printing the objects values by sorting on the basis of name and age.

```
1. import java.util.*;
2. import java.io.*;
3.
4. class Simple{
5.     public static void main(String args[]){
6.
7.         ArrayList al=new ArrayList();
8.         al.add(new Student(101,"Vijay",23));
9.         al.add(new Student(106,"Ajay",27));
10.        al.add(new Student(105,"Jai",21));
11.
12.        System.out.println("Sorting by Name...");
13.
14.        Collections.sort(al,new NameComparator());
```

```

15. Iterator itr=al.iterator();
16. while(itr.hasNext()){
17. Student st=(Student)itr.next();
18. System.out.println(st.rollno+" "+st.name+" "+st.age);
19. }
20.
21. System.out.println("sorting by age...");
22.
23. Collections.sort(al,new AgeComparator());
24. Iterator itr2=al.iterator();
25. while(itr2.hasNext()){
26. Student st=(Student)itr2.next();
27. System.out.println(st.rollno+" "+st.name+" "+st.age);
28. }
29.
30.
31. }
32. }

```

```

Sorting by Name...
106 Ajay 27
105 Jai 21
101 Vijay 23

```

```

Sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27

```

Stack:

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines the following methods –

Sr.No.	Method & Description
	boolean empty()
1	Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
	Object peek()
2	Returns the element on the top of the stack, but does not remove it.

3 Object pop()

Returns the element on the top of the stack, removing it in the process.

4 Object push(Object element)

Pushes the element onto the stack. Element is also returned.

int search(Object element)**5**

Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

EXAMPLE

The following program illustrates several of the methods supported by this collection –

```
import java.util.*;
public class StackDemo {

    static void showpush(Stack st, int a) {
        st.push(new Integer(a));
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }

    static void showpop(Stack st) {
        System.out.print("pop -> ");
        Integer a = (Integer) st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }

    public static void main(String args[]) {
        Stack st = new Stack();
        System.out.println("stack: " + st);
        showpush(st, 42);
        showpush(st, 66);
        showpush(st, 99);
        showpop(st);
        showpop(st);
        showpop(st);
        try {
            showpop(st);
        } catch (EmptyStackException e) {
            System.out.println("empty stack");
        }
    }
}
```

This will produce the following result –

OUTPUT

```
stack: [ ]
push(42)
stack: [42]
push(66)
```

```

stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack

```

Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences –

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.

Following is the list of constructors provided by the vector class.

Sr.No.	Constructor & Description
1	Vector() This constructor creates a default vector, which has an initial size of 10. Vector(int size)
2	This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size. Vector(int size, int incr)
3	This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward. Vector(Collection c)
4	This constructor creates a vector that contains the elements of collection c.

Apart from the methods inherited from its parent classes, Vector defines the following methods –

Sr.No.	Method & Description
1	void add(int index, Object element) Inserts the specified element at the specified position in this Vector.

2 boolean add(Object o)

Appends the specified element to the end of this Vector.

3 boolean addAll(Collection c)

Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.

4 boolean addAll(int index, Collection c)

Inserts all of the elements in in the specified Collection into this Vector at the specified position.

5 void addElement(Object obj)

Adds the specified component to the end of this vector, increasing its size by one.

6 int capacity()

Returns the current capacity of this vector.

7 void clear()

Removes all of the elements from this vector.

8 Object clone()

Returns a clone of this vector.

9 boolean contains(Object elem)

Tests if the specified object is a component in this vector.

10 boolean containsAll(Collection c)

Returns true if this vector contains all of the elements in the specified Collection.

11 void copyInto(Object[] anArray)

Copies the components of this vector into the specified array.

12 Object elementAt(int index)

Returns the component at the specified index.

13 Enumeration elements()

Returns an enumeration of the components of this vector.

14 void ensureCapacity(int minCapacity)

Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.

15 boolean equals(Object o)

Compares the specified Object with this vector for equality.

16 Object firstElement()

Returns the first component (the item at index 0) of this vector.

17 Object get(int index)

Returns the element at the specified position in this vector.

18 int hashCode()

Returns the hash code value for this vector.

19 int indexOf(Object elem)

Searches for the first occurrence of the given argument, testing for equality using the equals method.

20 int indexOf(Object elem, int index)

Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals method.

21 void insertElementAt(Object obj, int index)

Inserts the specified object as a component in this vector at the specified index.

22 boolean isEmpty()

Tests if this vector has no components.

23 Object lastElement()

Returns the last component of the vector.

24 int lastIndexOf(Object elem)

Returns the index of the last occurrence of the specified object in this vector.

25 int lastIndexOf(Object elem, int index)

Searches backwards for the specified object, starting from the specified index, and returns an index to it.

26 Object remove(int index)

Removes the element at the specified position in this vector.

27 boolean remove(Object o)

Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.

28 boolean removeAll(Collection c)

Removes from this vector all of its elements that are contained in the specified Collection.

29 void removeAllElements()

Removes all components from this vector and sets its size to zero.

30 boolean removeElement(Object obj)

Removes the first (lowest-indexed) occurrence of the argument from this vector.

- 31 **void removeElementAt(int index)**
removeElementAt(int index).
- 32 **protected void removeRange(int fromIndex, int toIndex)**
Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
- 33 **boolean retainAll(Collection c)**
Retains only the elements in this vector that are contained in the specified Collection.
- 34 **Object set(int index, Object element)**
Replaces the element at the specified position in this vector with the specified element.
- 35 **void setElementAt(Object obj, int index)**
Sets the component at the specified index of this vector to be the specified object.
- 36 **void setSize(int newSize)**
Sets the size of this vector.
- 37 **int size()**
Returns the number of components in this vector.
- 38 **List subList(int fromIndex, int toIndex)**
Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
- 39 **Object[] toArray()**
Returns an array containing all of the elements in this vector in the correct order.
- 40 **Object[] toArray(Object[] a)**
Returns an array containing all of the elements in this vector in the correct order; the runtime type of the returned array is that of the specified array.
- 41 **String toString()**
Returns a string representation of this vector, containing the String representation of each element.
- 42 **void trimToSize()**
Trims the capacity of this vector to be the vector's current size.

EXAMPLE

The following program illustrates several of the methods supported by this collection –

```
import java.util.*;  
public class VectorDemo {
```

```
public static void main(String args[]) {
    // initial size is 3, increment is 2
    Vector v = new Vector(3, 2);
    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " + v.capacity());

    v.addElement(new Integer(1));
    v.addElement(new Integer(2));
    v.addElement(new Integer(3));
    v.addElement(new Integer(4));
    System.out.println("Capacity after four additions: " + v.capacity());

    v.addElement(new Double(5.45));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Double(6.08));
    v.addElement(new Integer(7));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Float(9.4));
    v.addElement(new Integer(10));
    System.out.println("Current capacity: " + v.capacity());

    v.addElement(new Integer(11));
    v.addElement(new Integer(12));
    System.out.println("First element: " + (Integer)v.firstElement());
    System.out.println("Last element: " + (Integer)v.lastElement());

    if(v.contains(new Integer(3)))
        System.out.println("Vector contains 3.");

    // enumerate the elements in the vector.
    Enumeration vEnum = v.elements();
    System.out.println("\nElements in vector:");

    while(vEnum.hasMoreElements())
        System.out.print(vEnum.nextElement() + " ");
    System.out.println();
}
```

This will produce the following result –

OUTPUT

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```


BIT SET

The BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits. This is a legacy class but it has been completely re-engineered in Java 2, version 1.4.

The BitSet defines the following two constructors.

Sr.No.	Constructor & Description
1	BitSet() This constructor creates a default object. BitSet(int size)
2	This constructor allows you to specify its initial size, i.e., the number of bits that it can hold. All bits are initialized to zero.

BitSet implements the Cloneable interface and defines the methods listed in the following table

—

Sr.No.	Method & Description
1	void and(BitSet bitSet) ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.
2	void andNot(BitSet bitSet) For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	int cardinality() Returns the number of set bits in the invoking object.
4	void clear() Zeros all bits.
5	void clear(int index) Zeros the bit specified by index.
6	void clear(int startIndex, int endIndex) Zeros the bits from startIndex to endIndex.
7	Object clone() Duplicates the invoking BitSet object.
8	boolean equals(Object bitSet) Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise,

- the method returns false.
- 9 **void flip(int index)**
Reverses the bit specified by the index.
- 10 **void flip(int startIndex, int endIndex)**
Reverses the bits from startIndex to endIndex.
- 11 **boolean get(int index)**
Returns the current state of the bit at the specified index.
- 12 **BitSet get(int startIndex, int endIndex)**
Returns a BitSet that consists of the bits from startIndex to endIndex. The invoking object is not changed.
- 13 **int hashCode()**
Returns the hash code for the invoking object.
- 14 **boolean intersects(BitSet bitSet)**
Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
- 15 **boolean isEmpty()**
Returns true if all bits in the invoking object are zero.
- 16 **int length()**
Returns the number of bits required to hold the contents of the invoking BitSet. This value is determined by the location of the last 1 bit.
- 17 **int nextClearBit(int startIndex)**
Returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex.
- 18 **int nextSetBit(int startIndex)**
Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, -1 is returned.
- 19 **void or(BitSet bitSet)**
ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
- 20 **void set(int index)**
Sets the bit specified by index.
- 21 **void set(int index, boolean v)**
Sets the bit specified by index to the value passed in v. True sets the bit, false clears the bit.

- 22 **void set(int startIndex, int endIndex)**
Sets the bits from startIndex to endIndex.
- 23 **void set(int startIndex, int endIndex, boolean v)**
Sets the bits from startIndex to endIndex, to the value passed in v. true sets the bits, false clears the bits.
- 24 **int size()**
Returns the number of bits in the invoking BitSet object.
- 25 **String toString()**
Returns the string equivalent of the invoking BitSet object.
- 26 **void xor(BitSet bitSet)**
XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.

EXAMPLE

The following program illustrates several of the methods supported by this data structure –

```
import java.util.BitSet;
public class BitSetDemo {

    public static void main(String args[]) {
        BitSet bits1 = new BitSet(16);
        BitSet bits2 = new BitSet(16);

        // set some bits
        for(int i = 0; i < 16; i++) {
            if((i % 2) == 0) bits1.set(i);
            if((i % 5) != 0) bits2.set(i);
        }

        System.out.println("Initial pattern in bits1: ");
        System.out.println(bits1);
        System.out.println("\nInitial pattern in bits2: ");
        System.out.println(bits2);

        // AND bits
        bits2.and(bits1);
        System.out.println("\nbits2 AND bits1: ");
        System.out.println(bits2);

        // OR bits
        bits2.or(bits1);
        System.out.println("\nbits2 OR bits1: ");
        System.out.println(bits2);

        // XOR bits
        bits2.xor(bits1);
        System.out.println("\nbits2 XOR bits1: ");
        System.out.println(bits2);
    }
}
```

```
}
```

This will produce the following result –

OUTPUT

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

JAVA CALENDAR CLASS

Java Calendar class is an abstract class that provides methods for converting date between a specific instant in time and a set of calendar fields such as MONTH, YEAR, HOUR, etc. It inherits Object class and implements the Comparable interface.

JAVA CALENDAR CLASS DECLARATION

Let's see the declaration of java.util.Calendar class.

1. public abstract class Calendar extends Object
 2. implements Serializable, Cloneable, Comparable<Calendar>
-

METHODS OF JAVA CALENDAR

Method	Description
abstract void add(int field, int amount)	It is used to add or subtract the specified amount of time to the given calendar field, based on the calendar's rules.
int get(int field)	It is used to return the value of the given calendar field.
static Calendar getInstance()	It is used to get a calendar using the default time zone and locale.
abstract int getMaximum(int field)	It is used to return the maximum value for the given calendar field of this Calendar instance.
abstract int getMinimum(int field)	It is used to return the minimum value for the given calendar field of this Calendar instance.

field)	this Calendar instance.
void set(int field, int value)	It is used to set the given calendar field to the given value.
void setTime(Date date)	It is used to set this Calendar's time with the given Date.
Date getTime()	It is used to return a Date object representing this Calendar's time value.

JAVA CALENDAR CLASS EXAMPLE

```

1. import java.util.Calendar;
2. public class CalendarExample {
3.     public static void main(String[] args) {
4.         Calendar calendar = Calendar.getInstance();
5.         System.out.println("The current date is : " + calendar.getTime());
6.         calendar.add(Calendar.DATE, -15);
7.         System.out.println("15 days ago: " + calendar.getTime());
8.         calendar.add(Calendar.MONTH, 4);
9.         System.out.println("4 months later: " + calendar.getTime());
10.        calendar.add(Calendar.YEAR, 2);
11.        System.out.println("2 years later: " + calendar.getTime());
12.    }
13. }

```

Output:

```

The current date is : Thu Jan 19 18:47:02 IST 2017
15 days ago: Wed Jan 04 18:47:02 IST 2017
4 months later: Thu May 04 18:47:02 IST 2017
2 years later: Sat May 04 18:47:02 IST 2019

```

DATE

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors as shown in the following table.

Sr.No.	Constructor & Description
1	Date() This constructor initializes the object with the current date and time.
2	Date(long millisec) This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Following are the methods of the date class.

Sr.No.	Method & Description
1	boolean after(Date date) Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.
2	boolean before(Date date) Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.
3	Object clone() Duplicates the invoking Date object.
4	int compareTo(Date date) Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later than date.
5	int compareTo(Object obj) Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.
6	boolean equals(Object date) Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false.
7	long getTime() Returns the number of milliseconds that have elapsed since January 1, 1970.
8	int hashCode() Returns a hash code for the invoking object.
9	void setTime(long time) Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970.
10	String toString() Converts the invoking Date object into a string and returns the result.

GETTING CURRENT DATE AND TIME

This is a very easy method to get current date and time in Java. You can use a simple Date object with *toString()* method to print the current date and time as follows –

EXAMPLE

```
import java.util.Date;
public class DateDemo {

    public static void main(String args[]) {
        // Instantiate a Date object
        Date date = new Date();

        // display time and date using toString()
        System.out.println(date.toString());
    }
}
```

This will produce the following result –

OUTPUT

on May 04 09:51:52 CDT 2009

UNIT-V

AWT

AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table lists some of the many AWT classes.

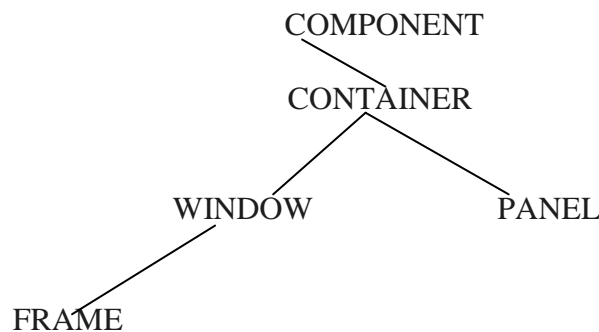
<u>Class</u>	<u>Description</u>
AWTEvent	Encapsulates AWT events.
BorderLayout	The border layout manager. Border layouts use Five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent Fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Event	Encapsulates events.
FileDialog	Creates a window from which a file can be selected.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GridBagLayout	The grid bag layout manager. Grid bag layout Displays components subject to the constraints specified by GridBagConstraints .
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Image	Encapsulates graphical images.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
Menu	Creates a pull-down menu.

MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
Scrollbar	Creates a scroll bar control.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for TextArea and TextField .
TextField	Creates a single-line edit control.
Window	Creates a window with no frame, no menu bar, and no title.

Window Fundamentals:

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding

the class hierarchy for **Panel** and **Frame**.



Component:

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container:

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

Panel:

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is

drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add()** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation()**, **setSize()**, or **setBounds()** methods defined by **Component**.

Window:

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**.

Frame:

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas:

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw

Working with Frame Windows:

After the applet, the type of window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for applications. As mentioned, it creates a standard-style window.

Here are two of **Frame**'s constructors:

Frame()

Frame(String title)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

There are several methods you will use when working with **Frame** windows. They are examined here.

1. Setting the Window's Dimensions

The **setSize()** method is used to set the dimensions of the window. Its signature is shown here:

void setSize(int newWidth, int newHeight)

void setSize(Dimension newSize)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels.

The **getSize()** method is used to obtain the current size of a window. Its signature is shown here:

Dimension getSize()

This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

2. Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call **setVisible()**. Its signature is shown here:

```
void setVisible(boolean visibleFlag)
```

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

3. Setting a Window's Title

You can change the title in a frame window using **setTitle()**, which has this general form:

```
void setTitle(String newTitle)
```

Here, *newTitle* is the new title for the window.

4. Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing()** method of the **WindowListener** interface. Inside **windowClosing()**, you must remove the window from the screen.

Creating a Frame Window in an Applet:

While it is possible to simply create a window by creating an instance of **Frame**, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets you override **Frame**'s methods and event handling.

Creating a new frame window from within an applet is actually quite easy. First, create a subclass of **Frame**. Next, override any of the standard window methods, such as **init()**, **start()**, **stop()**, and **paint()**. Finally, implement the **windowClosing()** method of the **WindowListener** interface, calling **setVisible(false)** when the window is closed. Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling **setVisible()**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **setSize()** method.

The following applet creates a subclass of **Frame** called **SampleFrame**. A window of this subclass is instantiated within the **init()** method of **AppletFrame**. Notice that **SampleFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in **title**. This example overrides the applet window's **start()** and **stop()** methods so that they show and hide the child window, respectively.

This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

// Create a child frame window from within an applet.

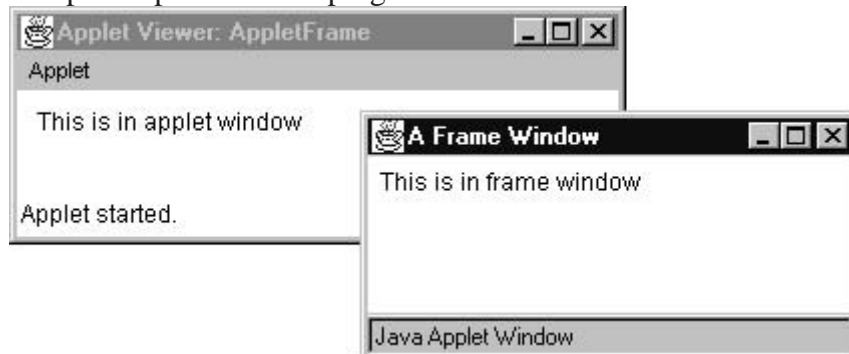
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AppletFrame" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame {
    SampleFrame(String title) {
        super(title);
    }
    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
```

```

addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString("This is in frame window", 10, 40);
}
}
class MyWindowAdapter extends WindowAdapter {
SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame) {
this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we) {
sampleFrame.setVisible(false);
}
}
// Create frame window.
public class AppletFrame extends Applet {
Frame f;
public void init() {
f = new SampleFrame("A Frame Window");
f.setSize(250, 250);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
public void paint(Graphics g) {
g.drawString("This is in applet window", 10, 20);
}
}

```

Sample output from this program is shown here:



Event Handling

Applets are event-driven programs. Thus, event handling is at the core of successful applet programming. Most events to which your applet will respond are generated by the user. These events are passed to your applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the **java.awt.event** package.

Two Event Handling Mechanisms:

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates

those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Java also allows you to process events without using the delegation event model. This can be done by extending an AWT component.

Events:

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Events may also occur that are not directly caused by interactions with a user interface.

For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources:

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

Here is the general form:

```
public void addTypeListener(TypeListener el)
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener()**. The method that registers a mouse motion listener is called **addMouseMotionListener()**.

When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

```
public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event.

The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener()**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners:

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes:

The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events.

Its one constructor is shown here:

```
EventObject(Object src)
```

Here, *src* is the object that generates this event.

EventObject contains two methods: **getSource()** and **toString()**. The **getSource()** method returns the source of the event.

Its general form is shown here:

```
Object getSource()
```

toString() returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID()** method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID()
```

■ **EventObject** is a superclass of all events.

■ **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines several types of events that are generated by various user interface elements.

Main Event Classes in java.awt.event:

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or

	becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionEvent Class:

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

`ActionEvent(Object src, int type, String cmd)`

`ActionEvent(Object src, int type, String cmd, int modifiers)`

`ActionEvent(Object src, int type, String cmd, long when, int modifiers)`

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand()** method, shown here:

`String getActionCommand()`

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers()** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.

Its form is shown here:

Event Class Description

`int getModifiers()`

the method **getWhen()** that returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen()** method is shown here.

`long getWhen()`

The AdjustmentEvent Class:

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT The user clicked inside the scroll bar to decrease its value.

BLOCK_INCREMENT The user clicked inside the scroll bar to increase its value.

TRACK The slider was dragged.

UNIT_DECREMENT The button at the end of the scroll bar was clicked to decrease its value.

UNIT_INCREMENT The button at the end of the scroll bar was clicked to increase its value.

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

`AdjustmentEvent(Adjustable src, int id, int type, int data)`

Here, *src* is a reference to the object that generated this event. The *id* equals **ADJUSTMENT_VALUE_CHANGED**. The type of the event is specified by *type*, and its associated data is *data*.

The **getAdjustable()** method returns the object that generated the event. Its form is shown here:

`Adjustable getAdjustable()`

The type of the adjustment event may be obtained by the **getAdjustmentType()** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

`int getAdjustmentType()`

the amount of the adjustment can be obtained from the **getValue()** method, shown here:

`int getValue()`

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class:

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN The component was hidden.

COMPONENT_MOVED The component was moved.

COMPONENT_RESIZED The component was resized.

COMPONENT_SHOWN The component became visible.

ComponentEvent has this constructor:

`ComponentEvent(Component src, int type)`

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**.

The **getComponent()** method returns the component that generated the event. It is shown here:

`Component getComponent()`

The ContainerEvent Class:

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of **ComponentEvent** and has this constructor:

`ContainerEvent(Component src, int type, Component comp)`

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*.

You can obtain a reference to the container that generated this event by using the **getContainer()** method, shown here:

`Container getContainer()`

The **getChild()** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild()

The FocusEvent Class:

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**. **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field.

If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the *opposite component*, is passed in *other*. Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus. You can determine the other component by calling **getOppositeComponent()**, shown here.

Component getOppositeComponent()

The opposite component is returned. The **isTemporary()** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary()

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

The InputEvent Class:

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**. **InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers.

ALT_MASK BUTTON2_MASK META_MASK

ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK

BUTTON1_MASK CTRL_MASK

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, Java 2, version 1.4 added the following extended modifier values.

ALT_DOWN_MASK ALT_GRAPH_DOWN_MASK BUTTON1_DOWN_MASK

BUTTON2_DOWN_MASK BUTTON3_DOWN_MASK CTRL_DOWN_MASK

META_DOWN_MASK SHIFT_DOWN_MASK

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown()**, **isAltGraphDown()**, **isControlDown()**, **isMetaDown()**, and **isShiftDown()** methods. The forms of these methods are shown here:

boolean isAltDown()

boolean isAltGraphDown()

boolean isControlDown()

boolean isMetaDown()

boolean isShiftDown()

You can obtain a value that contains all of the original modifier flags by calling the **getModifiers()** method. It is shown here:

int getModifiers()

You can obtain the extended modifiers by called **getModifiersEx()**, which is shown here.

int getModifiersEx()

The ItemEvent Class:

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. There are two types of item events, which are identified by the following integer constants:

DESELECTED The user deselected an item.

SELECTED The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

ItemEvent has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

The **getItem()** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

Object getItem()

The **getItemSelectable()** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

ItemSelectable getItemSelectable()

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.

The **getStateChange()** method returns the state change (i.e., **SELECTED** or **DESELECTED**) for the event. It is shown here:

int getStateChange()

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing the SHIFT key does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ENTER VK_ESCAPE VK_CANCEL VK_UP

VK_DOWN VK_LEFT VK_RIGHT VK_PAGE_DOWN

VK_PAGE_UP VK_SHIFT VK_ALT VK_CONTROL

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

KeyEvent is a subclass of **InputEvent**. Here are two of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*)

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred.

The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**. The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar()**, which returns the character that was entered, and **getKeyCode()**, which returns the key code. Their general forms are shown here:

```
char getKeyChar()
```

```
int getKeyCode()
```

If no valid character is available, then **getKeyChar()** returns **CHAR_UNDEFINED**.

When a **KEY_TYPED** event occurs, **getKeyCode()** returns **VK_UNDEFINED**.

The MouseEvent Class:

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.

MOUSE_DRAGGED The user dragged the mouse.

MOUSE_ENTERED The mouse entered a component.

MOUSE_EXITED The mouse exited from a component.

MOUSE_MOVED The mouse moved.

MOUSE_PRESSED The mouse was pressed.

MOUSE_RELEASED The mouse was released.

MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors.

```
MouseEvent(Component src, int type, long when, int modifiers,
int x, int y, int clicks, boolean triggersPopup)
```

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. Java 2, version 1.4 adds a second constructor which also allows the button that caused the event to be specified.

The most commonly used methods in this class are **getX()** and **getY()**. These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here:

```
int getX()
```

```
int getY()
```

Alternatively, you can use the **getPoint()** method to obtain the coordinates of the mouse. It is shown here:

```
Point getPoint()
```

It returns a **Point** object that contains the X, Y coordinates in its integer members: **x** and **y**. The **translatePoint()** method changes the location of the event. Its form is shown here:

```
void translatePoint(int x, int y)
```

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount()** method obtains the number of mouse clicks for this event. Its signature is shown here:

```
int getClickCount()
```

The **isPopupTrigger()** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```
boolean isPopupTrigger()
```

Java 2, version 1.4 added the **getButton()** method, shown here.

```
int getButton()
```

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **MouseEvent**.

NOBUTTON BUTTON1 BUTTON2 BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.

The MouseWheelEvent Class:

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent** and was added by Java 2, version 1.4. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants.

WHEEL_BLOCK_SCROLL A page-up or page-down scroll event occurred.

WHEEL_UNIT_SCROLL A line-up or line-down scroll event occurred.

MouseWheelEvent defines the following constructor.

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation()**, shown here.

int getWheelRotation()

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

To obtain the type of scroll, call **getScrollType()**, shown next.

int getScrollType()

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.

If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount()**. It is shown here.

int getScrollAmount()

The **TextEvent** Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.

The one constructor for this class is shown here:

TextEvent(Object *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the **TextEvent** class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class:

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED The window was activated.

WINDOW_CLOSED The window has been closed.

WINDOW_CLOSING The user requested that the window be closed.

WINDOW_DEACTIVATED The window was deactivated.

WINDOW_DEICONIFIED The window was deiconified.

WINDOW_GAINED_FOCUS The window gained input focus.

WINDOW_ICONIFIED The window was iconified.

WINDOW_LOST_FOCUS The window lost input focus.

WINDOW_OPENED The window was opened.

WINDOW_STATE_CHANGED The state of the window changed.

(Added by Java 2, version 1.4.)

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is `WindowEvent(Window src, int type)`

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

Java 2, version 1.4 adds the next three constructors.

`WindowEvent(Window src, int type, Window other)`

`WindowEvent(Window src, int type, int fromState, int toState)`

`WindowEvent(Window src, int type, Window other, int fromState, int toState)`

Here, *other* specifies the opposite window when a focus event occurs. The *fromState* specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs. The most commonly used method in this class is **getWindow()**. It returns the **Window** object that generated the event. Its general form is shown here:

`Window getWindow()`

Java 2, version 1.4, adds methods that return the opposite window (when a focus event has occurred), the previous window state, and the current window state. These methods are shown here:

`Window getOppositeWindow()`

`int getOldState()`

`int getNewState()`

Sources of Events

Event Source Description:

Event Source Examples:

Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces:

the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface Description:

ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

1) The ActionListener Interface:

This interface defines the **actionPerformed()** method that is invoked when an action event occurs. Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

2) The AdjustmentListener Interface:

This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs. Its general form is shown here:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

3) The ComponentListener Interface:

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

```
void componentResized(ComponentEvent ce)
```

```
void componentMoved(ComponentEvent ce)
```

```
void componentShown(ComponentEvent ce)
```

```
void componentHidden(ComponentEvent ce)
```

*The AWT processes the resize and move events. The **componentResized()** and **componentMoved()** methods are provided for notification purposes only.*

4) The ContainerListener Interface:

This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked. Their general forms are shown here:

```
void componentAdded(ContainerEvent ce)
```

void componentRemoved(ContainerEvent *ce*)

5) The FocusListener Interface:

This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. When a component loses keyboard focus, **focusLost()** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)

void focusLost(FocusEvent *fe*)

6) The ItemListener Interface:

This interface defines the **itemStateChanged()** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

7) The KeyListener Interface:

This interface defines three methods. The **keyPressed()** and **keyReleased()** methods are invoked when a key is pressed and released, respectively. The **keyTyped()** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)

void keyReleased(KeyEvent *ke*)

void keyTyped(KeyEvent *ke*)

8) The MouseListener Interface:

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked()** is invoked. When the mouse enters a component, the **mouseEntered()** method is called. When it leaves, **mouseExited()** is called. The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

9) The MouseMotionListener Interface:

This interface defines two methods. The **mouseDragged()** method is called multiple times as the mouse is dragged. The **mouseMoved()** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

10) The MouseWheelListener Interface:

This interface defines the **mouseWheelMoved()** method that is invoked when the mouse wheel is moved. Its general form is shown here.

void mouseWheelMoved(MouseWheelEvent *mwe*)

MouseWheelListener was added by Java 2, version 1.4.

11) The TextListener Interface:

This interface defines the **textChanged()** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

12) The WindowFocusListener Interface:

This interface defines two methods: **windowGainedFocus()** and **windowLostFocus()**. These are called when a window gains or losses input focus. Their general forms are shown here.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

WindowFocusListener was added by Java 2, version 1.4.

13) The WindowListener Interface:

This interface defines seven methods. The **windowActivated()** and **windowDeactivated()** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified()** method is called. When a window is deiconified, the **windowDeiconified()** method is called. When a window is opened or closed, the **windowOpened()** or **windowClosed()** methods are called, respectively. The **windowClosing()** method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Using the Delegation Event Model:

Applet programming using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle the two most commonly used event generators: the mouse and keyboard.

Handling Mouse Events:

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**, but we won't be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area. As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint()** to display output at the point of these occurrences.

// Demonstrate the mouse event handlers.

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



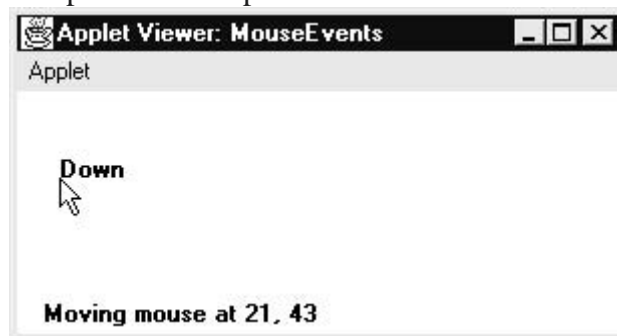
```
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener(this);
addMouseMotionListener(this);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
```

```

public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// show status
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}

```

Sample output from this program is shown here:



Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events.

This works because **Component**, which supplies the **addMouseListener()** and **addMouseMotionListener()** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets. Inside **init()**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener()** and **addMouseMotionListener()**, which, as mentioned, are members of **Component**. They are shown here:

```

void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)

```

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events:

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface. Before looking at an example, it is useful to review how key events are generated.

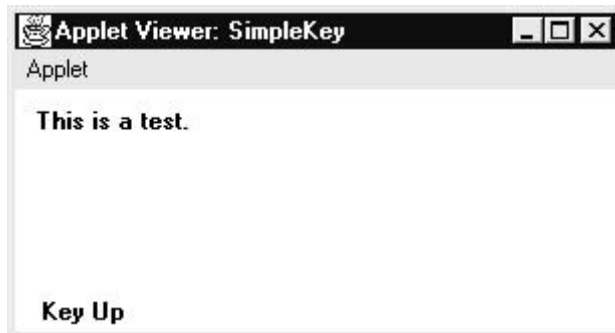
When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed()** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased()** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped()** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed()** handler.

There is one other requirement that your program must meet before it can process keyboard events: it must request input focus. To do this, call **requestFocus()**, which is defined by **Component**. If you don't, then your program will not receive any keyboard events.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener
{
    String msg = "";
    int X = 10, Y = 20; // output coordinates
    public void init() {
        addKeyListener(this);
        requestFocus(); // request input focus
    }
    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
    }
    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
    }
    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }
    // Display keystrokes.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

Sample output is shown here:



If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed()** handler. They are not available through **keyTyped()**. To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class KeyEvents extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
requestFocus(); // request input focus
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
int key = ke.getKeyCode();
switch(key) {
case KeyEvent.VK_F1:
msg += "<F1>";
break;
case KeyEvent.VK_F2:
msg += "<F2>";
break;
case KeyEvent.VK_F3:
msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;

```

```

case KeyEvent.VK_RIGHT:
    msg += "<Right Arrow>";
    break;
}
repaint();
}
public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
    g.drawString(msg, X, Y);
}
}

```

Sample output is shown here:



The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

Adapter Classes:

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested. For example, the **MouseMotionAdapter** class has two methods, **mouseDragged()** and **mouseMoved()**. The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and implement **mouseDragged()**. The empty implementation of **mouseMoved()** would handle the mouse motion events for you.

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init()** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and

registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. **MyMouseAdapter** implements the **mouseClicked()** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** implements the **mouseDragged()** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Note that both of our event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the **showStatus()** method.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}
class MyMouseAdapter extends MouseAdapter {
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
        adapterDemo.showStatus("Mouse clicked");
    }
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
    }
}
```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

Inner Classes:

the basics of inner classes were explained. Here you will see why they are important. Recall that an *inner class* is a class defined within other class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init()** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener()** method.

Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed()** method. When the mouse is pressed, it invokes the **showStatus()** method of the applet through the stored applet reference. In other words, **showStatus()** is invoked relative to the applet reference stored by **MyMouseAdapter**.

// This applet does NOT use an inner class.

```
import java.applet.*;
```

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

```
/*
```

```
<applet code="MousePressedDemo" width=200 height=100>
```

```
</applet>
```

```
*/
```

```
public class MousePressedDemo extends Applet {
```

```
public void init() {
```

```
addMouseListener(new MyMouseAdapter(this));
```

```
}
```

```
}
```

```
class MyMouseAdapter extends MouseAdapter {
```

```
MousePressedDemo mousePressedDemo;
```

```
public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
```

```
this.mousePressedDemo = mousePressedDemo;
```

```
}
```

```
public void mousePressed(MouseEvent me) {
```

```
mousePressedDemo.showStatus("Mouse Pressed.");
```

```
}
```

```
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed()** method can call the **showStatus()** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter()** a reference to the invoking object.

// Inner class demo.

```
import java.applet.*;
```

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

```
/*
```

```
<applet code="InnerClassDemo" width=200 height=100>
```

```
</applet>
```

```
*/
```

```
public class InnerClassDemo extends Applet {
```



```

public void init() {
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
}
}
}

```

Anonymous Inner Classes:

An *anonymous* inner class is one that is not assigned a name. Consider the applet shown in the following listing. As before, its goal is to display the string “Mouse Pressed” in the status bar of the applet viewer or browser when the mouse is pressed.

// Anonymous inner class demo.

```
import java.applet.*;
```

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

```
/*
```

```
<applet code="AnonymousInnerClassDemo" width=200 height=100>
```

```
</applet>
```

```
*/
```

```
public class AnonymousInnerClassDemo extends Applet {
```

```
public void init() {
```

```
addMouseListener(new MouseAdapter() {
```

```
public void mousePressed(MouseEvent me) {
```

```
showStatus("Mouse Pressed");
```

```
}
```

```
});
```

```
}
```

```
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init()** method calls the **addMouseListener()** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let’s analyze this expression carefully. The syntax **new MouseAdapter() { ... }** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus()** method directly.

JAVA SWING TUTORIAL

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is USED TO CREATE WINDOW-BASED APPLICATIONS. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

DIFFERENCE BETWEEN AWT AND SWING

There are many differences between java awt and swing that are given below.

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent .	Java swing components are platform-independent .
2)	AWT components are heavyweight .	Swing components are lightweight .
3)	AWT doesn't support pluggable look and feel .	Swing supports pluggable look and feel .
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC .

WHAT IS JFC

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

Do You Know

- How to create runnable jar file in java?
- How to display image on a button in swing?
- How to change the component color by choosing a color from ColorChooser ?
- How to display the digital watch in swing tutorial ?
- How to create a notepad in swing?
- How to create puzzle game and pic puzzle game in swing ?
- How to create tic tac toe game in swing ?

HIERARCHY OF JAVA SWING CLASSES

The hierarchy of java swing API is given below.

COMMONLY USED METHODS OF COMPONENT CLASS

The methods of Component class are widely used in java swing that are given below.

Method	Description
<code>public void add(Component c)</code>	add a component on another component.
<code>public void setSize(int width,int height)</code>	sets size of the component.
<code>public void setLayout(LayoutManager m)</code>	sets the layout manager for the component.
<code>public void setVisible(boolean b)</code>	sets the visibility of the component. It is by default false.

JAVA SWING EXAMPLES

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

We can write the code of swing inside the `main()`, constructor or any other method.

SIMPLE JAVA SWING EXAMPLE

Let's see a simple swing example where we are creating one button and adding it on the `JFrame` object inside the `main()` method.

File: `FirstSwingExample.java`

```

1. import javax.swing.*;
2. public class FirstSwingExample {
3.     public static void main(String[] args) {
4.         JFrame f=new JFrame();//creating instance of JFrame
5.
6.         JButton b=new JButton("click");//creating instance of JButton
7.         b.setBounds(130,100,100, 40);//x axis, y axis, width, height
8.
9.         f.add(b);//adding button in JFrame
10.
11.        f.setSize(400,500);//400 width and 500 height
12.        f.setLayout(null);//using no layout managers
13.        f.setVisible(true);//making the frame visible
14.    }
15. }
```

EXAMPLE OF SWING BY ASSOCIATION INSIDE CONSTRUCTOR

We can also write all the codes of creating JFrame, JButton and method call inside the java constructor.

File: Simple.java

```
1. import javax.swing.*;
2. public class Simple {
3.     JFrame f;
4.     Simple(){
5.         f=new JFrame();//creating instance of JFrame
6.
7.         JButton b=new JButton("click");//creating instance of JButton
8.         b.setBounds(130,100,100, 40);
9.
10.        f.add(b);//adding button in JFrame
11.
12.        f.setSize(400,500);//400 width and 500 height
13.        f.setLayout(null);//using no layout managers
14.        f.setVisible(true);//making the frame visible
15.    }
16.
17.    public static void main(String[] args) {
18.        new Simple();
19.    }
20. }
```

The `setBounds(int xaxis, int yaxis, int width, int height)` is used in the above example that sets the position of the button.

SIMPLE EXAMPLE OF SWING BY INHERITANCE

We can also inherit the JFrame class, so there is no need to create the instance of JFrame class explicitly.

File: Simple2.java

```
1. import javax.swing.*;
2. public class Simple2 extends JFrame{//inheriting JFrame
3.     JFrame f;
4.     Simple2(){
5.         JButton b=new JButton("click");//create button
6.         b.setBounds(130,100,100, 40);
7.
8.         add(b);//adding button on frame
```

```

9. setSize(400,500);
10. setLayout(null);
11. setVisible(true);
12. }
13. public static void main(String[] args) {
14. new Simple2();
15. }}

```

Handling Events in a Frame Window:

Since **Frame** is a subclass of **Component**, it inherits all the capabilities defined by **Component**. This means that you can use and manage a frame window that you create just like you manage your applet's main window. For example, you can override **paint()** to display output, call **repaint()** when you need to restore the window, and override all event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.

```

// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="WindowEvents" width=300 height=50>
</applet>
*/
// Create a subclass of Frame.
class SampleFrame extends Frame
implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX=10, mouseY=40;
int movX=0, movY=0;
SampleFrame(String title) {
super(title);
// register this object to receive its own mouse events
addMouseListener(this);
addMouseMotionListener(this);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
}
// Handle mouse entered.
public void mouseEntered(MouseEvent evtObj) {
// save coordinates

```

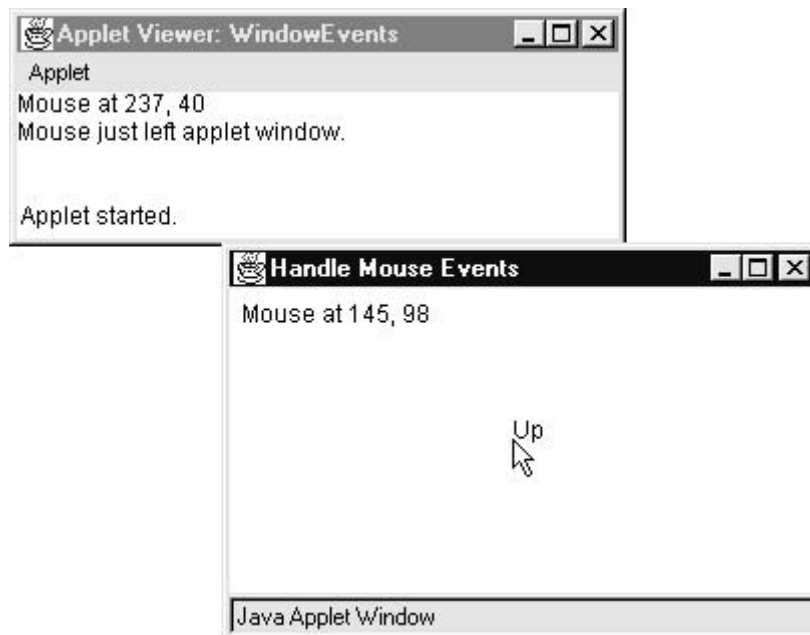
```
mouseX = 10;
mouseY = 54;
msg = "Mouse just entered child.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent evtObj) {
// save coordinates
mouseX = 10;
mouseY = 54;
msg = "Mouse just left child window.";
repaint();
}
// Handle mouse pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle mouse released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
movX = me.getX();
movY = me.getY();
msg = "*";
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// save coordinates
movX = me.getX();
movY = me.getY();
repaint(0, 0, 100, 60);
}
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
}
}
```

```
class MyWindowAdapter extends WindowAdapter {
    SampleFrame sampleFrame;
    public MyWindowAdapter(SampleFrame sampleFrame) {
        this.sampleFrame = sampleFrame;
    }
    public void windowClosing(WindowEvent we) {
        sampleFrame.setVisible(false);
    }
}
// Applet window.
public class WindowEvents extends Applet
    implements MouseListener, MouseMotionListener {
    SampleFrame f;
    String msg = "";
    int mouseX=0, mouseY=10;
    int movX=0, movY=0;
    // Create a frame window.
    public void init() {
        f = new SampleFrame("Handle Mouse Events");
        f.setSize(300, 200);
        f.setVisible(true);
        // register this object to receive its own mouse events
        addMouseListener(this);
        addMouseMotionListener(this);
    }
    // Remove frame window when stopping applet.
    public void stop() {
        f.setVisible(false);
    }
    // Show frame window when starting applet.
    public void start() {
        f.setVisible(true);
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me) {
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 24;
        msg = "Mouse just entered applet window.";
        repaint();
    }
    // Handle mouse exited.
    public void mouseExited(MouseEvent me) {
        // save coordinates
        mouseX = 0;
        mouseY = 24;
        msg = "Mouse just left applet window.";
        repaint();
    }
}
```

```
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
movX = me.getX();
movY = me.getY();
msg = "*";
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {

// save coordinates
movX = me.getX();
movY = me.getY();
repaint(0, 0, 100, 20);
}
// Display msg in applet window.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
}
}
```

Sample output from this program is shown here:



Creating a Windowed Program:

Although creating applets is the most common use for Java's AWT, it is possible to create stand-alone AWT-based applications, too. To do this, simply create an instance of the window or windows you need inside **main()**. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
// Create a frame window.
public class AppWindow extends Frame {
    String keymsg = "This is a test.";
    String mousemsg = "";
    int mouseX=30, mouseY=30;
    public AppWindow() {
        addKeyListener(new MyKeyAdapter(this));
        addMouseListener(new MyMouseAdapter(this));
        addWindowListener(new MyWindowAdapter());
    }
    public void paint(Graphics g) {
        g.drawString(keymsg, 10, 40);
        g.drawString(mousemsg, mouseX, mouseY);
    }
// Create the window.
    public static void main(String args[]) {
        AppWindow appwin = new AppWindow();
        appwin.setSize(new Dimension(300, 200));
        appwin.setTitle("An AWT-Based Application");
        appwin.setVisible(true);
    }
}
class MyKeyAdapter extends KeyAdapter {
```

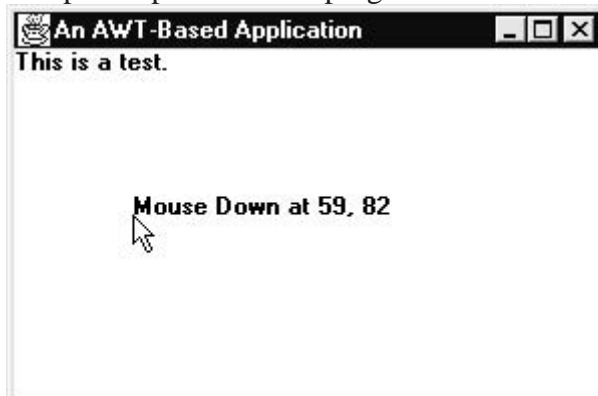


```

AppWindow appWindow;
public MyKeyAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void keyTyped(KeyEvent ke) {
appWindow.keymsg += ke.getKeyChar();
appWindow.repaint();
};
}
class MyMouseAdapter extends MouseAdapter {
AppWindow appWindow;
public MyMouseAdapter(AppWindow appWindow) {
this.appWindow = appWindow;
}
public void mousePressed(MouseEvent me) {
appWindow.mouseX = me.getX();
appWindow.mouseY = me.getY();
appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
", " + appWindow.mouseY;
appWindow.repaint();
}
}
class MyWindowAdapter extends WindowAdapter {
public void windowClosing(WindowEvent we) {
System.exit(0);
}
}

```

Sample output from this program is shown here:



Once created, a frame window takes on a life of its own. Notice that **main()** ends with the call to **appwin.setVisible(true)**. However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use **main()** to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs used earlier.

user interface components:

Controls are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. A menu bar is always positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls. While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. We introduces the various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization.

Control Fundamentals:

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of **Component**.

Adding and Removing Controls:

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add()**, which is defined by **Container**. The **add()** method has several forms.

Component add(Component *compObj*)

Here, *compObj* is an instance of the control that you want to add. A reference to *compObj* is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove()**. This method is also defined by **Container**.

It has this general form:

void remove(Component *obj*)

Here, *obj* is a reference to the control you want to remove. You can remove all controls by calling **removeAll()**.

Responding to Controls

Except for labels, which are passive controls, all controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 20, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

1.Labels:

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user.

Label defines the following constructors:

Label()

Label(String *str*)

Label(String *str*, int *how*)

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**. You can set or change the text in a label by using the **setText()** method. You can obtain the current label by calling **getText()**. These methods are shown here:

```
void setText(String str)
```

```
String getText( )
```

For **setText()**, *str* specifies the new label. For **getText()**, the current label is returned. You can set the alignment of the string within the label by calling **setAlignment()**. To obtain the current alignment, call **getAlignment()**. The methods are as follows:

```
void setAlignment(int how)
```

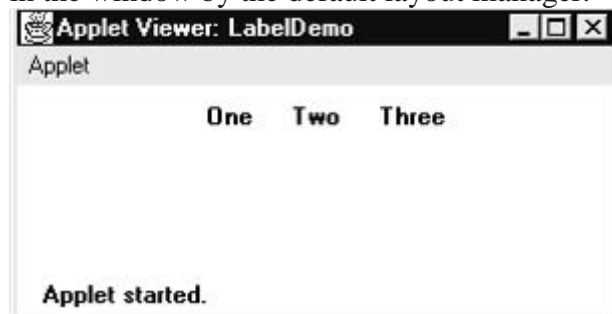
```
int getAlignment( )
```

Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet {
public void init() {
Label one = new Label("One");
Label two = new Label("Two");
Label three = new Label("Three");
// add labels to applet window
add(one);
add(two);
add(three);
}
}
```

Following is the window created by the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager.



2.Buttons:

The most widely used control is the push button. A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

```
Button( )
```

```
Button(String str)
```

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel()**. You can retrieve its label by calling **getLabel()**. These methods are as follows:

```
void setLabel(String str)
```

String getLabel()

Here, *str* becomes the new label for the button.

3.Check Boxes:

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:

Checkbox()

Checkbox(String *str*)

Checkbox(String *str*, boolean *on*)

Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*)

Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*)

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box. To retrieve the current state of a check box, call **getState()**. To set its state, call **setState()**. You can obtain the current label associated with a check box by calling **getLabel()**. To set the label, call **setLabel()**. These methods are as follows:

boolean getState()

void setState(boolean *on*)

String getLabel()

void setLabel(String *str*)

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

4.CheckboxGroup:

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons*, because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group. You can determine which check box in a group is currently selected by calling **getSelectedCheckbox()**. You can set a check box by calling **setSelectedCheckbox()**.

These methods are as follows:

Checkbox getSelectedCheckbox()

void setSelectedCheckbox(Checkbox *which*)

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

5.Choice Controls:

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** only defines the default constructor, which creates an empty list.

To add a selection to the list, call **add()**. It has this general form:

```
void add(String name)
```

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add()** occur.

To determine which item is currently selected, you may call either **getSelectedItem()** or **getSelectedIndex()**. These methods are shown here:

```
String getSelectedItem()
```

```
int getSelectedIndex()
```

The **getSelectedItem()** method returns a string containing the name of the item. **getSelectedIndex()** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount()**. You can set the currently selected item using the **select()** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

```
int getItemCount()
```

```
void select(int index)
```

```
void select(String name)
```

Given an index, you can obtain the name associated with the item at that index by calling **getItem()**, which has this general form:

```
String getItem(int index)
```

Here, *index* specifies the index of the desired item.

6.Lists:

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

```
List()
```

```
List(int numRows)
```

```
List(int numRows, boolean multipleSelect)
```

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time.

If it is **false**, then only one item may be selected.

To add a selection to the list, call **add()**. It has the following two forms:

```
void add(String name)
```

```
void add(String name, int index)
```

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify -1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem()** or **getSelectedIndex()**. These methods are shown here:

```
String getSelectedItem()
```

```
int getSelectedIndex()
```

The **getSelectedItem()** method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, **null** is returned. **getSelectedIndex()** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, **-1** is returned. For lists that allow multiple selection, you must use either **getSelectedItems()** or **getSelectedIndexes()**, shown here, to determine the current selections:

```
String[ ] getSelectedItems()
```

```
int[ ] getSelectedIndexes()
```

getSelectedItems() returns an array containing the names of the currently selected items.

getSelectedIndexes() returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount()**. You can set the currently selected item by using the **select()** method with a zero-based integer index.

These methods are shown here:

```
int getItemCount()
```

```
void select(int index)
```

Given an index, you can obtain the name associated with the item at that index by calling **getItem()**, which has this general form:

```
String getItem(int index)
```

Here, *index* specifies the index of the desired item

7.Managing Scroll Bars:

Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down.

Scroll bars are encapsulated by the **Scrollbar** class.

Scrollbar defines the following constructors:

```
Scrollbar()
```

```
Scrollbar(int style)
```

```
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
```

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*. If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues()**, shown here, before it can be used:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue()**. It returns the current setting. To set the current value, call **setValue()**. These methods are as follows:

```
int getValue()
```

```
void setValue(int newValue)
```


Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value. You can also retrieve the minimum and maximum values via **getMinimum()** and **getMaximum()**, shown here:

```
int getMinimum()
```

```
int getMaximum()
```

They return the requested quantity. By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement()**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement()**. These methods are shown here:

```
void setUnitIncrement(int newIncr)
```

```
void setBlockIncrement(int newIncr)
```

8.Using a TextField:

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

```
TextField()
```

```
TextField(int numChars)
```

```
TextField(String str)
```

```
TextField(String str, int numChars)
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width. **TextField** (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText()**. To set the text, call **setText()**. These methods are as follows:

```
String getText()
```

```
void setText(String str)
```

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select()**. Your program can obtain the currently selected text by calling **getSelectedText()**. These methods are shown here:

```
String getSelectedText()
```

```
void select(int startIndex, int endIndex)
```

getSelectedText() returns the selected text. The **select()** method selects the characters beginning at *startIndex* and ending at *endIndex*-1. You can control whether the contents of a text field may be modified by the user by calling **setEditable()**. You can determine editability by calling **isEditable()**. These methods are shown here:

```
boolean isEditable()
```

```
void setEditable(boolean canEdit)
```

isEditable() returns **true** if the text may be changed and **false** if not. In **setEditable()**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar()**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet()** method. You can retrieve the echo character by calling the **getEchoChar()** method.

These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed.

9.Using a TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

```
TextArea( )
TextArea(int numLines, int numChars)
TextArea(String str)
TextArea(String str, int numLines, int numChars)
TextArea(String str, int numLines, int numChars, int sBars)
```

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

```
SCROLLBARS_BOTH SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY SCROLLBARS_VERTICAL_ONLY
```

TextArea is a subclass of **TextComponent**. Therefore, it supports the **getText()**, **setText()**, **getSelectedText()**, **select()**, **isEditable()**, and **setEditable()** methods. **TextArea** adds the following methods:

```
void append(String str)
void insert(String str, int index)
void replaceRange(String str, int startIndex, int endIndex)
```

The **append()** method appends the string specified by *str* to the end of the current text. **insert()** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange()**. It replaces the characters from *startIndex* to *endIndex*-1, with the replacement text passed in *str*.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Text areas only generate got-focus and lost-focus events. Normally, your program simply obtains the current text when it is needed.

Understanding Layout Managers:

All of the components that we have shown so far have been positioned by the default layout manager. a layout manager automatically arranges your controls within a window by using some type of algorithm.

If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized.

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout()** method. If no call to **setLayout()** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout()** method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```


Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds()** method defined by **Component**. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize()** and **preferredLayoutSize()** methods. Each component that is being managed by a layout manager contains the **getPreferredSize()** and **getMinimumSize()** methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise. Java has several predefined **LayoutManager** classes.

1.FlowLayout:

FlowLayout is the default layout manager. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout()

FlowLayout(int how)

FlowLayout(int how, int horz, int vert)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

These values specify left, center, and right alignment, respectively. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

2.BorderLayout:

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

BorderLayout()

BorderLayout(int horz, int vert)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. **BorderLayout** defines the following constants that specify the regions:

BorderLayout.CENTER BorderLayout.SOUTH

BorderLayout.EAST BorderLayout.WEST

BorderLayout.NORTH

When adding components, you will use these constants with the following form of **add()**, which is defined by **Container**:

void add(Component compObj, Object region);

Here, *compObj* is the component to be added, and *region* specifies where the component will be added.

3.GridLayout:

GridLayout lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

```
GridLayout( )
```

```
GridLayout(int numRows, int numColumns )
```

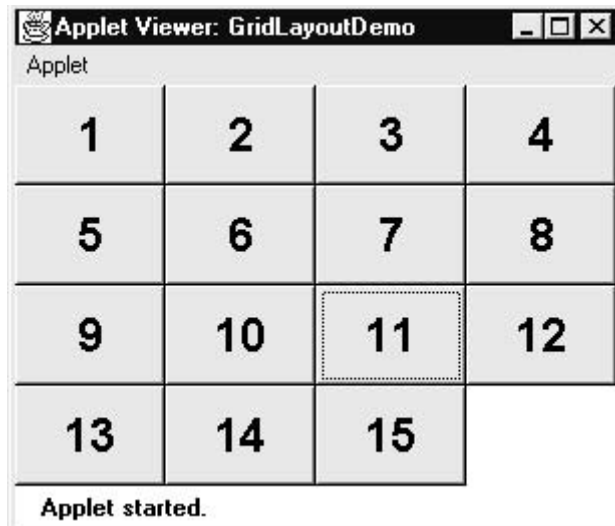
```
GridLayout(int numRows, int numColumns, int horz, int vert)
```

The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
public class GridLayoutDemo extends Applet {
    static final int n = 4;
    public void init() {
        setLayout(new GridLayout(n, n));
        setFont(new Font("SansSerif", Font.BOLD, 24));
        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                int k = i * n + j;
                if(k > 0)
                    add(new Button("" + k));
            }
        }
    }
}
```

Following is the output generated by the **GridLayoutDemo**



4.CardLayout:

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

CardLayout provides these two constructors:

`CardLayout()`

`CardLayout(int horz, int vert)`

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the main applet panel. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck. When card panels are added to a panel, they are usually given a name. Thus, most

of the time, you will use this form of **add()** when adding cards to a panel:

`void add(Component panelObj, Object name);`

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelObj*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

`void first(Container deck)`

`void last(Container deck)`

`void next(Container deck)`

`void previous(Container deck)`

`void show(Container deck, String cardName)`

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first()** causes the first card in the deck to be shown. To show the last card, call **last()**. To show the next card, call **next()**. To show the previous card, call

previous(). Both **next()** and **previous()** automatically cycle back to the top or bottom of the deck, respectively. The **show()** method displays the card whose name is passed in *cardName*.

Menu Bars and Menus:

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in Java by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu()

Menu(String optionName)

Menu(String optionName, boolean removable)

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu. Individual menu items are of type **MenuItem**. It defines these constructors:

MenuItem()

MenuItem(String itemName)

MenuItem(String itemName, MenuShortcut keyAccel)

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item. You can disable or enable a menu item by using the **setEnabled()** method. Its form is shown here:

void setEnabled(boolean enabledFlag)

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled. You can determine an item's status by calling **isEnabled()**. This method is shown here:

boolean isEnabled()

isEnabled() returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**. You can change the name of a menu item by calling **setLabel()**. You can retrieve the current name by using **getLabel()**. These methods are as follows:

void setLabel(String newName)

String getLabel()

Here, *newName* becomes the new name of the invoking menu item. **getLabel()** returns the current name. You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

CheckboxMenuItem()

CheckboxMenuItem(String itemName)

CheckboxMenuItem(String itemName, boolean on)

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared. You can obtain the status of a checkable item by calling **getState()**. You can set it to a known state by using **setState()**. These methods are shown here:

boolean getState()

`void setState(boolean checked)`

If the item is checked, **getState()** returns **true**. Otherwise, it returns **false**. To check an item, pass **true** to **setState()**. To clear an item, pass **false**. Once you have created a menu item, you must add the item to a **Menu** object by using **add()**, which has the following general form:

`MenuItem add(MenuItem item)`

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add()** take place. The *item* is returned. Once you have added all items to a **Menu** object, you can add that object to the menubar by using this version of **add()** defined by **MenuBar**:

`Menu add(Menu menu)`

Here, *menu* is the menu being added. The *menu* is returned.

Menus only generate events when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated.

Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and **ItemListener** interfaces in order to handle these menu events.

The **getItem()** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

`Object getItem()`

Dialog Boxes:

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input. They are similar to frame windows, except that dialog boxes are always child windows of a top-level window. Also, dialog boxes don't have menu bars. In other respects, dialog boxes function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. Dialog boxes are of type **Dialog**.

Two commonly used constructors are shown here:

`Dialog(Frame parentWindow, boolean mode)`

`Dialog(Frame parentWindow, String title, boolean mode)`

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application. When the dialog box is closed, **dispose()** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

FileDialog:

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type **FileDialog**. This causes a file dialog box to be displayed. Usually, this is the standard file dialog box provided by the operating system. **FileDialog** provides these constructors:

`FileDialog(Frame parent, String boxName)`

`FileDialog(Frame parent, String boxName, int how)`

`FileDialog(Frame parent)`

Here, *parent* is the owner of the dialog box, and *boxName* is the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is

FileDialog.LOAD, then the box is selecting a file for reading. If *how* is **FileDialog.SAVE**, the box is selecting a file for writing. The third constructor creates a dialog box for selecting a file for reading. **FileDialog()** provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

```
String getDirectory( )
```

```
String getFile( )
```

These methods return the directory and the filename, respectively.

Working with Graphics:

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

- It is passed to an applet when one of its various methods, such as **paint()** or **update()**, is called.

- It is returned by the **getGraphics()** method of **Component**.

The **Graphics** class defines a number of drawing functions. Each shape can be drawn edge-only or filled. Objects are drawn and filled in the currently selected graphics color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. Let's take a look at several of the drawing methods.

Drawing Lines

Lines are drawn by means of the **drawLine()** method, shown here:

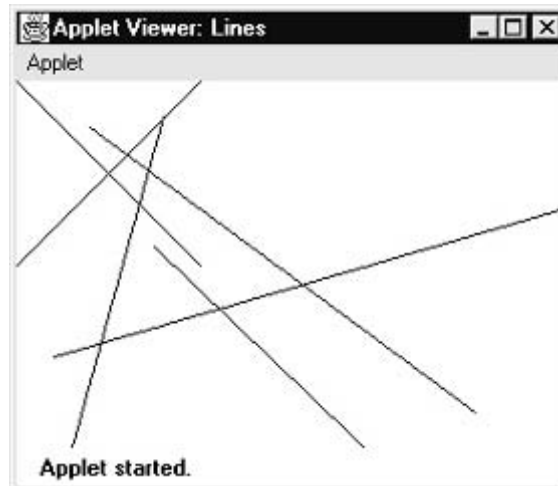
```
void drawLine(int startX, int startY, int endX, int endY)
```

drawLine() displays a line in the current drawing color that begins at *startX,startY* and ends at *endX,endY*.

The following applet draws several lines:

```
// Draw lines
import java.awt.*;
import java.applet.*;
/*
<applet code="Lines" width=300 height=200>
</applet>
*/
public class Lines extends Applet {
public void paint(Graphics g) {
g.drawLine(0, 0, 100, 100);
g.drawLine(0, 100, 100, 0);
g.drawLine(40, 25, 250, 180);
g.drawLine(75, 90, 400, 400);
g.drawLine(20, 150, 400, 40);
g.drawLine(5, 290, 80, 19);
}
}
```

Sample output from this program is shown here:



Drawing Rectangles

The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively. They are shown here:

```
void drawRect(int top, int left, int width, int height)
```

```
void fillRect(int top, int left, int width, int height)
```

The upper-left corner of the rectangle is at *top, left*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect()** or **fillRoundRect()**, both shown here:

```
void drawRoundRect(int top, int left, int width, int height,
```

```
int xDiam, int yDiam) void fillRoundRect(int top, int left, int width, int height,
```

```
int xDiam, int yDiam)
```

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *top, left*. The dimensions of the rectangle are specified by *width* and *height*. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

Drawing Ellipses and Circles

To draw an ellipse, use **drawOval()**. To fill an ellipse, use **fillOval()**. These methods are shown here:

```
void drawOval(int top, int left, int width, int height)
```

```
void fillOval(int top, int left, int width, int height)
```

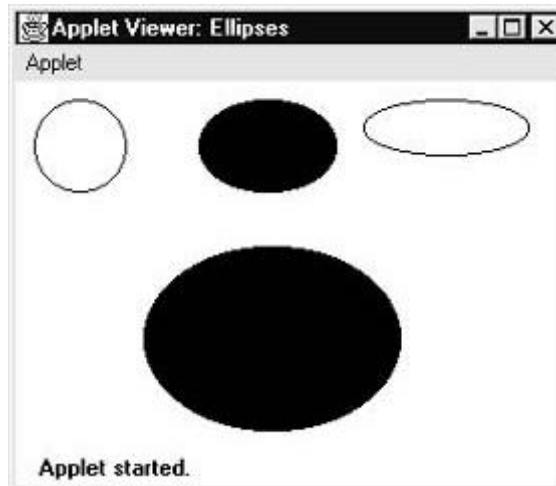
The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *top, left* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle.

The following program draws several ellipses:

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/*
<applet code="Ellipses" width=300 height=200>
</applet>
*/
public class Ellipses extends Applet {
public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
```

```
}  
}
```

Sample output from this program is shown here



UNIT-8 Applets & swings

Applet Basics:

The **Applet** class is contained in the **java.applet** package. All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must also import **java.awt**. Recall that AWT stands for the Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for that window. Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer. Execution of an applet does not begin at **main()**. Actually, few applets even have **main()** methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by **System.out.println(**

). Rather, it is handled with various AWT methods, such as **drawString()**, which outputs a string to a specified X,Y location. Input is also handled differently than in an application.

Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target.

example of such a comment:

```
/*  
<applet code="MyApplet" width=200 height=60>  
</applet>  
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier.

differences between applets and applications: An applet runs under the control of a browser, whereas an application runs stand-alone, with the support of a virtual machine. As such, an applet is subjected to more stringent security restrictions in terms of file and network access, whereas an application can have free reign over these resources.

Applets are great for creating dynamic and interactive web applications, but the true power of Java lies in writing full blown applications. With the limitation of disk and network access, it would be difficult to write commercial applications (though through the user of server based file systems, not impossible). However, a Java application has full network and local file system access, and its potential is limited only by the creativity of its developers.

OTHER DIFFERENCES BETWEEN APPLETS AND APPLICATIONS

- No main() method: Applets do not need a main() method to exist anywhere because they are running inside another program (the browser) and are thus not a stand-alone program.
- No parameterized constructor: The browser calls the default, unparameterized constructor of an Applet, so parameterized constructors are not generally useful. Applets can be called from the web page with parameters however. See the discussion of the getParameters() method below.
- The init() method: An applet runs this special method AFTER THE CONSTRUCTOR IS RUN. This method is used instead of the constructor to initialize the applet.
- Applets are stay resident: Applets stay in the computer's memory until the browser itself closes. That is, even if the browser changes to a new web page, the applet

stays around. THE APPLET'S INIT() METHOD IS ONLY RUN ONCE, NO MATTER HOW MANY TIMES THE BROWSER REVISITS THE PAGE WITH THE APPLET.

- The start() method: This method of an applet is called whenever the applet is shown in the browser. IT IS CALLED AFTER INIT() AND EVERY TIME THE PAGE CONTAINING THE APPLET IS REVISITED. This method is useful for reinitializing the applet.
- The stop() method: This method of an applet IS CALLED WHENEVER THE BROWSER LEAVES THE WEB PAGE CONTAINING THE APPLET. It is useful for performing any clean-up needed by the applet to prepare it for a "dormant" state when the browser is elsewhere. In particular, it is important to shutdown any extra threads that are running and to free up other resources.
- The destroy() method: This method of an Applet is called when the browser itself exits to free up the applet's resources. It is not usually called by the applet itself.
- The getParameter() method: This method of an Applet can be called from within the applet to obtain the parameters, in String form, that were passed to the applet by the web browser. The name of the desired parameter is the input value of this method, and a string value is returned.
- The getAppletInfo() method: This method of an Applet is called by the browser, say from a Javascript script, to obtain information about the applet.
- The getParameterInfo() method: This method of an Applet is called by the browser, say from a Javascript script, to obtain information about the parameters needed by an applet.

life cycle of an applet: All applets have the following four methods:

```
public void init();  
public void start();  
public void stop();  
public void destroy();
```

They have these methods because their superclass, java.applet.Applet, has these methods.

In the superclass, these are simply do-nothing methods. For example,

```
public void init() { }
```

Subclasses may override these methods to accomplish certain tasks at certain times. For instance, the init() method is a good place to read parameters that were passed to the applet via <PARAM> tags because it's called exactly once when the applet starts up. However, they do not have to override them. Since they're declared in the superclass, the Web browser can invoke them when it needs to without knowing in advance whether the method is implemented in the superclass or the subclass. This is a good example of polymorphism.

INIT(), START(), STOP(), AND DESTROY()

The `init()` method is called exactly once in an applet's life, when the applet is first loaded. It's normally used to read `PARAM` tags, start downloading any other images or media files you need, and set up the user interface. Most applets have `init()` methods.

The `start()` method is called at least once in an applet's life, when the applet is started or restarted. In some cases it may be called more than once. Many applets you write will not have explicit `start()` methods and will merely inherit one from their superclass. A `start()` method is often used to start any threads the applet will need while it runs.

The `stop()` method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's `start()` method will be called if at some later point the browser returns to the page containing the applet. In some cases the `stop()` method may be called multiple times in an applet's life. Many applets you write will not have explicit `stop()` methods and will merely inherit one from their superclass. Your applet should use the `stop()` method to pause any running threads. When your applet is stopped, it **SHOULD** not use any CPU cycles.

The `destroy()` method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final clean-up. For example, an applet that stores state on the server might send some data back to the server before it's terminated. many applets will not have explicit `destroy()` methods and just inherit one from their superclass.

For example, in a video applet, the `init()` method might draw the controls and start loading the video file. The `start()` method would wait until the file was loaded, and then start playing it. The `stop()` method would pause the video, but not rewind it. If the `start()` method were called again, the video would pick up where it left off; it would not start over from the beginning. However, if `destroy()` were called and then `init()`, the video would start over from the beginning.

In the JDK's appletviewer, selecting the Restart menu item calls `stop()` and then `start()`. Selecting the Reload menu item calls `stop()`, `destroy()`, and `init()`, in that order. (Normally the byte codes will also be reloaded and the HTML file reread though Netscape has a problem with this.)The applet `start()` and `stop()` methods are not related to the similarly named methods in the `java.lang.Thread` class.

Your own code may occasionally invoke `start()` and `stop()`. For example, it's customary to stop playing an animation when the user clicks the mouse in the applet and restart it when they click the mouse again.

Your own code can also invoke `init()` and `destroy()`, but this is normally a bad idea. Only the environment should call `init()` and `destroy()`.

Types of applets: There are two types of applets.

- 1.local applets:these applets are run within the browser.
- 2.remote applets:these applets are run on the internet.

Difference is that local applet operate in single machine browser which is not connected in network,while remote applet operate over internet via network.

Local and Remote Applets:

One of Java's major strengths is that you can use the language to create dynamic content for your Web pages. That is, thanks to Java applets, your Web pages are no longer limited to the tricks you can perform with HTML. Now your Web pages can do just about anything you want them to. All you need to do is write the appropriate applets.

But writing Java applets is only half the story. How your Web page's users obtain and run the applets is equally as important. It's up to you to not only write the applet (or use someone else's applet), but also to provide users access to the applet. Basically, your Web pages can contain two types of applets: local and remote. In this section, you learn the difference between these applet types, which are named after the location at which they are stored.

LOCAL APPLETS

A local applet is one that is stored on your own computer system. When your Web page must find a local applet, it doesn't need to retrieve information from the Internet—in fact, your browser doesn't even need to be connected to the Internet at that time. A local applet is specified by a path name and a file name.

Specifying a Local Applet.

```
<applet  
  
    codebase="tictactoe"  
  
    code="TicTacToe.class"  
  
    width=120  
  
    height=120>  
  
</applet>
```

the codebase attribute specifies a path name on your system for the local applet, whereas the code attribute specifies the name of the byte-code file that contains the applet's code. The path specified in the codebase attribute is relative to the folder containing the HTML document that references the applet.

REMOTE APPLETS

A remote applet is one that is located on another computer system. This computer system may be located in the building next door or it may be on the other side of the world—it makes no difference to your Java-compatible browser. No matter where the remote applet is located, it's downloaded onto your computer via the Internet. Your browser must, of course, be connected to the Internet at the time it needs to display the remote applet.

To reference a remote applet in your Web page, you must know the applet's URL (where it's located on the Web) and any attributes and parameters that you need to supply in order to display the applet correctly. If you didn't write the applet, you'll need to find the document that

describes the applet's attributes and parameters. This document is usually written by the applet's author. Listing shows how to compose an HTML <applet> tag that accesses a remote applet.

Listing: Specifying a Remote Applet.

```
<applet

    codebase="http://www.myconnect.com/applets/"

    code="TicTacToe.class"

    width=120

    height=120>

</applet>
```

Creating Applets:

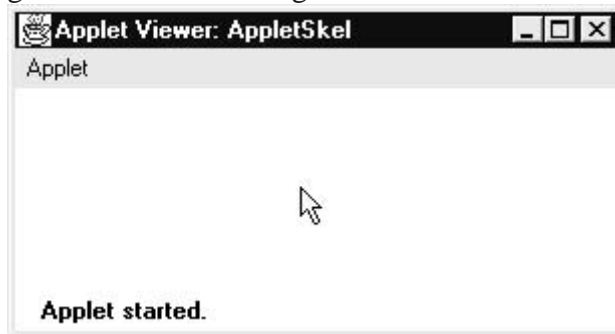
An Applet Skeleton

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—**init()**, **start()**, **stop()**, and **destroy()**—are defined by **Applet**. Another, **paint()**, is defined by the AWT **Component** class. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet {
// Called first.
public void init() {
// initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start() {
// start or resume execution
}
// Called when the applet is stopped.
public void stop() {
// suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy() {
```

```
// perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g) {
// redisplay contents of window
}
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following window when viewed with an applet viewer:



Designing a web page:

Java applets are programs which reside in web pages. In order to run an applet, it is first necessary to have a web page that references the applet. A web page is also known as an HTML page or HTML document.

Web pages are stored using a file extension .html, we write the applet tag as

```
<applet
Code =hello.class
Width=400
Height=200>
</applet>
```

This HTML code tells the browser to load the hello.class which is in the same directory as the HTML file.

Adding applet to HTML file:

```
<html>
<body>
<applet
Code =hello.class
Width=400
Height=200>
</applet>
</body>
</html>
```

We save the file as hello.html and save the file in the same directory where the .class file is present.

Running the applet:

In the current directory we must have the following files
Hello.java

Hello.class

Hello.html

To run the applet we need one of the following tools:

- 1.java enabled web browser
- 2.java appletviewer

To run on a appletviewer we do as

Appletviewer hello.html

Passing Parameters to Applets:

the APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter()** method. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **Boolean** values, you will need to convert their string representations into their internal formats.

Here is an example that demonstrates passing parameters

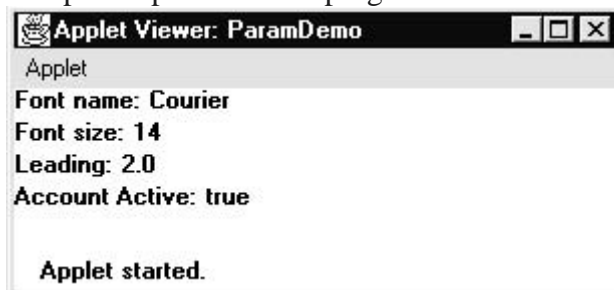
```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet{
String fontName;
int fontSize;
float leading;
boolean active;
// Initialize the string to be displayed.
public void start() {
String param;
fontName = getParameter("fontName");
if(fontName == null)
fontName = "Not Found";
param = getParameter("fontSize");
try {
if(param != null) // if not found
fontSize = Integer.parseInt(param);
else
fontSize = 0;
} catch(NumberFormatException e) {
fontSize = -1;
}
param = getParameter("leading");
try {
if(param != null) // if not found
leading = Float.valueOf(param).floatValue();
else
```

```

leading = 0;
} catch(NumberFormatException e) {
leading = -1;
}
param = getParameter("accountEnabled");
if(param != null)
active = Boolean.valueOf(param).booleanValue();
}
// Display parameters.
public void paint(Graphics g) {
g.drawString("Font name: " + fontName, 0, 10);
g.drawString("Font size: " + fontSize, 0, 26);
g.drawString("Leading: " + leading, 0, 42);
g.drawString("Account Active: " + active, 0, 58);
}
}

```

Sample output from this program is shown here:



As the program shows, you should test the return values from **getParameter()**. If a parameter isn't available, **getParameter()** will return **null**. Also, conversions to numeric types must be attempted in a **try** statement that catches **NumberFormatException**. Uncaught exceptions should never occur within an applet.

SWING

Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT. In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements. The number of classes and interfaces in the Swing packages is substantial

The Swing component classes that are used are shown here:

<u>Class</u>	<u>Description</u>
AbstractButton	Abstract superclass for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of Applet .
JButton	The Swing push button class.

JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (an combination of a drop-down list and text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

The Swing-related classes are contained in **javax.swing** and its subpackages, such as **javax.swing.tree**.

limitations of AWT:

AWT:

Pros

- Speed: use of native peers speeds component performance.
- Applet Portability: most Web browsers support AWT classes so AWT applets can run without the Java plugin.
- Look and Feel: AWT components more closely reflect the look and feel of the OS they run on.

Cons

- Portability: use of native peers creates platform specific limitations. Some components may not function at all on some platforms.
- Third Party Development: the majority of component makers, including Borland and Sun, base new component development on Swing components. There is a much smaller set of AWT components available, thus placing the burden on the programmer to create his or her own AWT-based components.
- Features: AWT components do not support features like icons and tool-tips.

Swing:

Pros

- Portability: Pure Java design provides for fewer platform specific limitations.
- Behavior: Pure Java design allows for a greater range of behavior for Swing components since they are not limited by the native peers that AWT uses.
- Features: Swing supports a wider range of features like icons and pop-up tool-tips for components.
- Vendor Support: Swing development is more active. Sun puts much more energy into making Swing robust.
- Look and Feel: The pluggable look and feel lets you design a single set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, Macintosh, etc.). It also makes it easier to make global changes to

your Java programs that provide greater accessibility (like picking a hi-contrast color scheme or changing all the fonts in all dialogs, etc.).

Cons

- **Applet Portability:** Most Web browsers do not include the Swing classes, so the Java plugin must be used.
- **Performance:** Swing components are generally slower and buggier than AWT, due to both the fact that they are pure Java and to video issues on various platforms. Since Swing components handle their own painting (rather than using native API's like DirectX on Windows) you may run into graphical glitches.
- **Look and Feel:** Even when Swing components are set to use the look and feel of the OS they are run on, they may not look like their native counterparts.

Exploring Swing:

JApplet:

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not found in **Applet**. For example, **JApplet** supports various “panes,” such as the content pane, the glass pane, and the root pane. one difference between **Applet** and **JApplet** is important to this discussion, because it is used by the sample applets in this chapter. When adding a component to an instance of **JApplet**, do not invoke the **add()** method of the applet. Instead, call **add()** for the *content pane* of the **JApplet** object. The content pane can be obtained via the method shown here

Container getContentPane()

The **add()** method of **Container** can be used to add a component to a content pane.

Its form is shown here:

void add(*comp*)

Here, *comp* is the component to be added to the content pane.

Icons and Labels:

In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. Two of its constructors are shown here:

ImageIcon(String *filename*)

ImageIcon(URL *url*)

The first form uses the image in the file named *filename*. The second form uses the image in the resource identified by *url*. The **ImageIcon** class implements the **Icon** interface that declares the methods

shown here:

Method

Description

int getIconHeight()

Returns the height of the icon in pixels.

int getIconWidth()

Returns the width of the icon in pixels.

void paintIcon(Component *comp*, Graphics *g*,int *x*, int *y*) Paints the icon at position *x*, *y* on the graphics context *g*. Additional information about the paint operation can be provided in *comp*.

Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and/or an icon. Some of its constructors are shown here:

JLabel(Icon *i*)

Label(String *s*)

JLabel(String *s*, Icon *i*, int *align*)

Here, *s* and *i* are the text and icon used for the label. The *align* argument is either **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes. The icon and text associated with the label can be read and written by the following methods:

Icon getIcon()

String getText()

void setIcon(Image i)

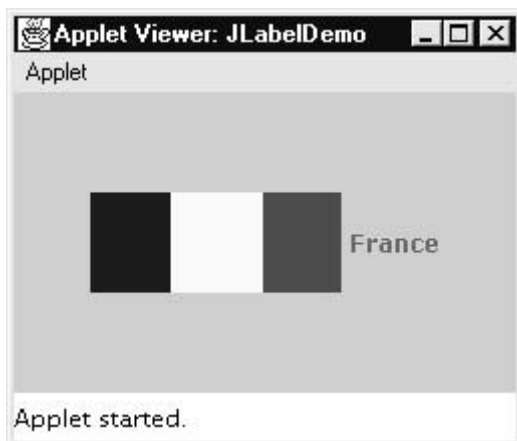
void setText(String s)

Here, *i* and *s* are the icon and text, respectively.

The following example illustrates how to create and display a label containing both an icon and a string. The applet begins by getting its content pane. Next, an **ImageIcon** object is created for the file **france.gif**. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet {
    public void init() {
        // Get content pane
        Container contentPane = getContentPane();
        // Create an icon
        ImageIcon ii = new ImageIcon("france.gif");
        // Create a label
        JLabel jl = new JLabel("France", ii, JLabel.CENTER);
        // Add label to the content pane
        contentPane.add(jl);
    }
}
```

Output:



Text Fields:

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField()

JTextField(int cols)

`JTextField(String s, int cols)`

`JTextField(String s)`

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

Buttons:

Swing buttons provide features that are not found in the **Button** class defined by the AWT. For example, you can associate an icon with a Swing button. Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**. **AbstractButton** contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons. For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over that component.

The following are the methods that control this behavior:

`void setDisabledIcon(Icon di)`

`void setPressedIcon(Icon pi)`

`void setSelectedIcon(Icon si)`

`void setRolloverIcon(Icon ri)`

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for these different conditions.

The text associated with a button can be read and written via the following methods:

`String getText()`

`void setText(String s)`

Here, *s* is the text to be associated with the button.

Concrete subclasses of **AbstractButton** generate action events when they are pressed. Listeners register and unregister for these events via the methods shown here:

`void addActionListener(ActionListener al)`

`void removeActionListener(ActionListener al)`

Here, *al* is the action listener.

AbstractButton is a superclass for push buttons, check boxes, and radio buttons.

The JButton Class:

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

`JButton(Icon i)`

`JButton(String s)`

`JButton(String s, Icon i)`

Here, *s* and *i* are the string and icon used for the button.

Check Boxes:

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are shown here:

`JCheckBox(Icon i)`

`JCheckBox(Icon i, boolean state)`

`JCheckBox(String s)`

`JCheckBox(String s, boolean state)`

`JCheckBox(String s, Icon i)`

`JCheckBox(String s, Icon i, boolean state)`

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not.

The state of the check box can be changed via the following method:

```
void setSelected(boolean state)
```

Here, *state* is **true** if the check box should be checked.

The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field. The content pane for the **JApplet** object is obtained, and a flow layout is assigned as its layout manager. Next, four check boxes are added to the content pane, and icons are assigned for the normal, rollover, and selected states. The applet is then registered to receive item events. Finally, a text field is added to the content pane.

When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged()**. Inside **itemStateChanged()**, the **getItem()** method gets the **JCheckBox** object that generated the event. The **getText()** method gets the text for that check box and uses it to set the text inside the text field.

Radio Buttons:

Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are shown here:

```
JRadioButton(Icon i)
```

```
JRadioButton(Icon i, boolean state)
```

```
JRadioButton(String s)
```

```
JRadioButton(String s, boolean state)
```

```
JRadioButton(String s, Icon i)
```

```
JRadioButton(String s, Icon i, boolean state)
```

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

```
void add(AbstractButton ab)
```

Here, *ab* is a reference to the button to be added to the group.

Combo Boxes:

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field. Two of **JComboBox**'s constructors are shown here:

```
JComboBox()
```

```
JComboBox(Vector v)
```

Here, *v* is a vector that initializes the combo box.

Items are added to the list of choices via the **addItem()** method, whose signature is shown here:

```
void addItem(Object obj)
```

Here, *obj* is the object to be added to the combo box.

Tabbed Panes:

A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

```
void addTab(String str, Component comp)
```

Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.

The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a **JTabbedPane** object.
2. Call **addTab()** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane. The first tab is titled “Cities” and contains four buttons. Each button displays the name of a city. The second tab is titled “Colors” and contains three check boxes. Each check box displays the name of a color. The third tab is titled “Flavors” and contains one combo box. This enables the user to select one of three flavors.

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet {
    public void init() {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        getContentPane().add(jtp);
    }
}

class CitiesPanel extends JPanel {
    public CitiesPanel() {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
    }
}

class ColorsPanel extends JPanel {

    public ColorsPanel() {

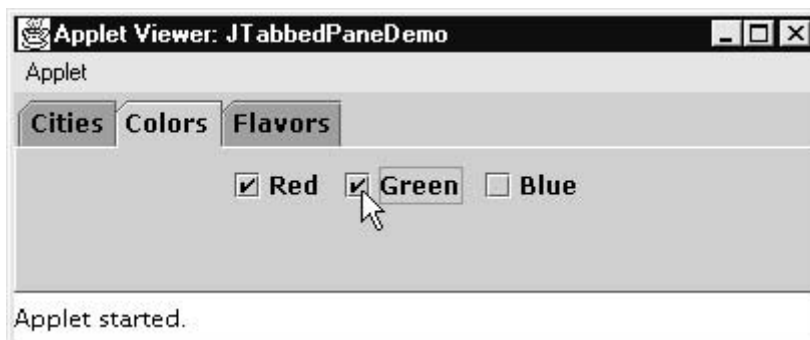
        JCheckBox cb1 = new JCheckBox("Red");
        add(cb1);
        JCheckBox cb2 = new JCheckBox("Green");
        add(cb2);
        JCheckBox cb3 = new JCheckBox("Blue");
        add(cb3);
    }
}
```

```

}
}
class FlavorsPanel extends JPanel {
public FlavorsPanel() {
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}

```

Output:



Scroll Panes:

A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are shown here:

```
JScrollPane(Component comp)
```

```
JScrollPane(int vsb, int hsb)
```

```
JScrollPane(Component comp, int vsb, int hsb)
```

Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are **int** constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the **ScrollPaneConstants** interface. Some examples of these constants are described as follows:

Constant	Description
HORIZONTAL_SCROLLBAR_ALWAYS bar	Always provide horizontal scroll bar
HORIZONTAL_SCROLLBAR_AS_NEEDED needed	Provide horizontal scroll bar, if needed
VERTICAL_SCROLLBAR_ALWAYS	Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED	Provide vertical scroll bar, if needed

Here are the steps that you should follow to use a scroll pane in an applet:

1. Create a **JComponent** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.

Trees:

A *tree* is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class, which extends **JComponent**. Some of its constructors are shown here:

```
JTree(Hashtable ht)
```

```
JTree(Object obj[ ])
```

```
JTree(TreeNode tn)
```

```
JTree(Vector v)
```

The first form creates a tree in which each element of the hash table *ht* is a child node. Each element of the array *obj* is a child node in the second form. The tree node *tn* is the root of the tree in the third form. Finally, the last form uses the elements of vector *v* as child nodes.

A **JTree** object generates events when a node is expanded or collapsed. The **addTreeExpansionListener()** and **removeTreeExpansionListener()** methods allow listeners to register and unregister for these notifications. The signatures of these methods are shown here:

```
void addTreeExpansionListener(TreeExpansionListener tel)
```

```
void removeTreeExpansionListener(TreeExpansionListener tel)
```

Here, *tel* is the listener object.

The **getPathForLocation()** method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

```
TreePath getPathForLocation(int x, int y)
```

Here, *x* and *y* are the coordinates at which the mouse is clicked. The return value is a **TreePath** object that encapsulates information about the tree node that was selected by the user.

The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString()** method is used. It returns a string equivalent of the tree path. The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node. The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add()** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, *child* is a mutable tree node that is to be added as a child to the current node.

Tree expansion events are described by the class **TreeExpansionEvent** in the **javax.swing.event** package. The **getPath()** method of this class returns a **TreePath** object that describes the path to the changed node. Its signature is shown here:

```
TreePath getPath()
```

The **TreeExpansionListener** interface provides the following two methods:

```
void treeCollapsed(TreeExpansionEvent tee)
```

```
void treeExpanded(TreeExpansionEvent tee)
```

Here, *tee* is the tree expansion event. The first method is called when a subtree is hidden, and the second method is called when a subtree becomes visible.

Here are the steps that you should follow to use a tree in an applet:

1. Create a **JTree** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

Tables:

A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent**. One of its constructors is shown here:

```
JTable(Object data[ ][ ], Object colHeads[ ])

```

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Here are the steps for using a table in an applet:

1. Create a **JTable** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

12. UNITWISE QUESTION BANK

(SUBJECTIVE & OBJECTIVE)

UNIT-I:**SUBJECTIVE:**

1. List at least ten major differences between C and Java
2. Distinguish between Objects and classes
3. Write the difference between Data abstraction and data encapsulation
4. Explain about Inheritance and polymorphism
5. Explain: (a) Dynamic binding
(b) Message passing.
6. Compare and contrast overloading and overriding methods.

OBJECTIVE:

1. What is main essential feature of object oriented programming Abstraction
2. Inheritance is the property by which one object acquires the properties of another object.
3. Having many forms is called polymorphism.
4. Static polymorphism is the polymorphism exhibited at compile time.
5. Dynamic polymorphism is the polymorphism exhibited at run time.
6. An object is an instance of a class.
7. instanceof means object.
8. Method overloading is an example of Polymorphism.
9. Two methods with same name ,same type signature is called as method overriding.
10. Errors that occur at runtime are called as Exceptions.

UNIT-I:**SUBJECTIVE:**

1. Describe the genesis of java. Also write brief overview of java
2. List and explain the control statements used in java. Also describe the syntax of the control statements with suitable illustration
3. What is an array? Why arrays are easier to use compared to a bunch of related variables?
4. What are conventional styles for class names, method names, constants and variables?
5. Can a java run on any machine? What is needed to run java on a computer?

6. Explain the concept of keywords. List some java keywords.
7. (a) what is a constructor? What are its special properties?
(b) How do we invoke a constructor?
(c) What are objects? How are they created from a class?
8. (a) What is the purpose of using a method? How do you declare a method? How do you invoke a method?
(b) What is method overloading? Can you define two methods that have same name but different parameter types? Can you define two methods in a class that have identical method names and parameter profile with different return value types or different modifier?
9. Describe the following terms:
(a) super and this
(b) final and abstract
(c) Passing parameter-call by value
(d) Overloading methods & Constructors.
10. (a) Describe the structure of a typical java program.
(b) Why do we need the import statement?
(c) What is statement? How do the java statements differ from those of C and C++?
11. (a) List the eight data types used in Java. Give examples.
(b) Write a while loop to find the smallest n such that n² is greater than 10,000.
12. Explain the following methods of StringBuffer class and write a java program illustrating these. Length(), capacity(), SetLength(), EnsureCapacity().

OBJECTIVE:

1. When java is compiled the code we will get Bytecode
2. Java's byte code is platform independent
3. JVM contains an interpreter, to speed up the interpretation what is there with interpreter JIT compiler
4. Java is a purely object oriented language
5. What is the memory size of short? 2 bytes
6. What is the memory size of short? 4 bytes
7. Characters are stored with their ASCII values.
8. Is string a class or data type? Both
9. Boolean act upon how many values? Two

10. Visibility of a variable is called as scope.
11. A specific element in array is accessed by its index.
12. Arrays of arrays means two dimensional arrays.
13. &&, ||, are called logical operators.
14. &,|,!, are called bitwise operators.
15. The operator used to refer a method in class is dot.
16. Which statement will terminate the application normally? System.exit(0).
17. Type conversion of two compatible java types is widening conversion.
18. A java program is compiled using javac command
19. Java program is run using java command.
20. Type casting of two incompatible types is narrowing conversion.
21. Which statement is useful to return a value to a calling method? Return
22. Which statement is used to continue the next repetition of loop? Continue.
23. Executing the statements one by one is called as sequential.
24. The operator used to refer a class in package is dot.
25. Which operator dynamically allocates memory for an object? new
26. Objects are created during runtime by JVM.
27. Variables declared as final are constants.
28. Static methods can access only static variables.
29. Static methods cannot refer to this and super.
30. A class within another class is known as nested class.
31. Every string you create is actually an object of type String.
32. Objects of type string are immutable.
33. Which operator is used to concatenate two strings? + Operator
34. What is the return type of string function equals ()? boolean
35. Which method of string class is used for eliminating all leading and trailing white Spaces? trim ()
36. Static classes are not there in java.
37. The scope of a default specifier is package scope

UNIT-I:

SUBJECTIVE:

1. (a) Discuss about Hybrid Inheritance with a suitable example.
(b) Discuss about Hierarchical Inheritance with a suitable example.
2. (a) Define Abstract class? Explain with a suitable example.
(b) Write a sample program to demonstrate the order of initialization of the base classes and derived classes. Now add member objects to both the base and derived classes, and show the order in which their initialization occurs during construction.
3. What are the types of inheritances in java? Explain each of them in detail.
4. Add a new method in the base class of Shapes. java that prints a message, but don't override it in the derived classes. Explain what happens. Now override it in one of the derived classes but not the others, and explain what happens. Finally, override it in all the derived classes, Explain in detail about each situation.
5. (a) Explain about final classes, final methods and final variables?
(b) Explain about the abstract class with example program?
6. Create a base class with an abstract print() method that is overridden in a derived class. The overridden version of the method prints the value of an int variable defined in the derived class. At the point of definition of this variable, give it a nonzero value. In the base-class constructor, call this method. In main(), create an object of the derived type, and then call its print() method. Explain the results.

OBJECTIVE:

1. Which oops concept is used for hierarchical classification? Inheritance
2. Which keyword is used to inherit a class? extends
3. Which constructor is always available to subclass? Default constructor
4. When a subclass object is created it contains the copy of super class object.
5. Private members of the super class are not available to sub class.
6. When a sub class refers to its immediate super class the keyword it uses is super.
7. Super keyword acts as that of this.
8. Multiple inheritance is not available in java.
9. Method overriding occurs when name and type signatures of two methods are identical.
10. Abstract methods must be declared with a keyword abstract.
11. A class with one or more abstract methods is called as abstract class.
12. We cannot create objects to abstract class.
13. A class cannot be both abstract and final.
14. Which is used for methods overriding Final

15. Which is used for methods to prevent Inheritance Final?
16. Methods declared as final cannot be overridden.
17. Declaring a class as final declare all of its methods implicitly final
18. All classes in java are subclass of object
19. Dynamic dispatch and method overloading implements runtime polymorphism.
20. Dynamic dispatch is made at runtime.

UNIT-II:

SUBJECTIVE:

1. (a) What is interface? Write a program to demonstrate how interfaces can be extended.
(b) What is package? How do you create a package? Explain about the access protection in packages?
2. Prove that all the methods in an interface are automatically public.
3. Create an interface with at least one method, in its own package. Create a class in a separate package. Add a protected inner class that implements the interface. In a third package, inherit from your class and, inside a method, return an object of the protected inner class, up casting to the interface during the return.
4. Write a sample program to illustrate packages.
5. What is interface? How do you define an interface? Explain with a suitable example.

OBJECTIVE:

1. Which keyword is to create package? Package.
2. Package contains name spaces in which classes are stored.
3. Java uses file system directory to store packages.
4. Import can be used to access package.
5. Java uses runtime directory for packages.
6. Import is used to import a package.
7. Dot operator is used to import a package.
8. An interface is specification of method prototype.
9. All methods of interfaces are public.
10. All methods of interfaces are public and abstract by default.
11. Can we create an object to interface no?
12. Can we extend interface to another interface yes.
13. Can class implement another interface yes.

14. An interface can be either abstract or default.
15. Variable in an interface is implicitly final and static.
16. The method that implements interface must be public.
17. Software that implements classes is driver.
18. class files are stored in the directory with the package name when a package is created.
19. If a package name is `jav.awt.image` then the directory structure is `java/awt/image`.
20. The general form of multilevel package is `Package p1[.p2[.p3]]`;

UNIT-III:

SUBJECTIVE:

1. (a) Give the Class hierarchy in Java related to exception handling. Briefly explain each class.
(b) What is the necessity of exception handling? Explain exception handling taking “divide-by-zero” as an example.
2. What is Error? What is Exception? Are they totally different or related? As a programmer
3. what is the difference in handling an error and an exception. With the help of a simple java program explain the concepts error and exception
4. (a) Explain throws statement in Java with the help of an example program.
(b) What is the difference between throw and throws statement.

OBJECTIVE:

1. An exception is runtime error.
2. Exception Handling is the concept of oops.
3. Java exception handling is managed by 5 no of keywords.
4. Try, catch and Throw is keyword used to manage exception handling.
5. throws is a keyword used to throw exception without handling.
6. Manually throw an exception keyword is throw.
7. Throw is the exception used to handling exception handling.
8. All exception is subclass of object and Throwable.
9. Exception is the immediate super class of runtime exception class.
10. Error type of exceptions are not caught under normal circumstances by your Program.
11. The statements that are protected by try are surrounded by curly braces.

12. Yes we can make try statement nested.
13. How many exceptions can be thrown with throw class with put handling them More than one.
14. Finally will execute whether or not option is thrown.
15. Finally and catch clause is optional.
16. Each try statement atleast one catch clause.
17. In java exception handling is defined in java.lang type.
18. 18. Most exceptions filled in runtime exception class are automatically available in java.
19. 19. We can create user defined exception by making subclasses to Runtime exception class.

UNIT-IV:

SUBJECTIVE:

1. (a) What is the significance of main thread in multithreading. Explain with an example how you can control main thread.
(b) What is the role of Sleep class in multithreading. Explain.
2. (a) why thread is called light weight task and process heavy weight task.
(b) What are the different things shared by different threads of a single process. What are the benefits of sharing?
(c) Is multithreading suitable for all types of applications. If yes explain any such application. If no, explain any application for which multithreading is not desired.
3. (a) With the help of an example, explain multithreading by extending thread class.
(b) Implementing Runnable interface and extending thread, which method you prefer for multithreading and why.
4. (a) Explain how threads with different priorities execute in environment which supports priorities and which doesn't support priorities.
(b) what are the functions available in java related to priority.

OBJECTIVE:

1. Thread is light weight class.
2. Context switching from one process to another process is costly in process.
3. Thread shows the address space.
4. 23. Thread Based is multitasking under control of java.
5. Multiple Try Blocks is not available in java.
6. Class object raise the exception by throwing user exception.

UNIT-V**SUBJECTIVE:**

1. (a) Why do you use frames?
(b) Explain the syntax and functionality of different methods related to Frames.
2. What are the methods supported by the following interfaces. Explain each of them
 - (a) ActionListener interface
 - (b) MouseMotionListener interface
 - (c) TextListener interface.
3. What are the methods supported by KeyListener interface and MouseListener Interface.
Explain each of them with examples.
4. using a Frame window, write a program to handle all mouse events?
5. (a) What is Delegation Event model? Explain it. What are its benefits?
(b) Define Event. Give examples of events. Define event handler. How it handles events.
6. Explain different event classes supported by Java.
7. Explain in detail about the following event classes:
 - (a) ComponentEvent
 - (b) ContainerEvent
 - (c) FocusEvent.
8. What is event source? Give examples of event sources. How events are generated. Are all events generated by user actions? Comment on it.

OBJECTIVE:

1. Handling events is based on the delegation event model.
2. A source generates an event and sends to one or more listeners.
3. An event is an object that describes state change in source.
4. The Awt event class is defined within java.awt package.
5. Event object is super class of all events.
6. Action Event is generated when a button is pressed or menu item is selected.
7. Item Event is generated when a check box or list item is clicked.
8. The methods used to return X and Y coordinates of mouse when event occurred is int getX (), int getY ().
9. An anonymous inner class is one that is not assigned a name.
10. AWT stands for Abstract window Tool kit.
11. The Graphics class defines a number of drawing class.
12. Lines are drawn by means of drawLine () method.

13. To set a change text in a label by using SetText () method.
14. To receive the current state of a checkbox we call getState () method.
15. The choice class is used to create a pop-up list of items which user may choose.
16. AWT includes multiline editor called TextArea.
17. Flow layout is the default layout manager.
18. Grid layout lays out components in a two-dimensional grid.
19. You can disable or enable a menu item by using the setEnabled () method.
20. Menu generates events when an item of type Menu item or Check box Menu Item is selected.

UNIT-V

SUBJECTIVE:

1. Explain the following:
 - (a) Creating an applet
 - (b) Passing parameters to applets
 - (c) Adding graphics and colors to applets.
2. (a) Write briefly applet display methods.
 - (b) Write a java program to form a calculator
3. How will you create check boxes and Choice boxes? Explain the steps in detail.
4. (a) Explain various components of User Interface.
 - (b) How will you arrange components on User Interface?
5. What are various JFC containers? List them according to their functionality. Explain each of them with examples.
6. (a) In what way JList differ from JComboBox?
 - (b) JList does not support scrolling. Why? How this can be remedied? Explain with an example.
7. Explain the steps involved in creating JCheckBox, JRadioButton, JButton, JLabel
8. Differentiate following with suitable examples:
 - (a) Frame, JFrame
 - (b) Applet, JApplet
 - (c) Menu, Jmenu.

OBJECTIVE:

1. All applets are subclass of Applet.

2. Applets are executed by either web browser or an applet viewer.
3. The standard applet viewer is provided by JDK.
4. Execution of an applet does not begin with at main().
5. Applet is window-based program.
6. The four methods defined by Applet are init(),start(),paint()
7. When an applet begins, the AWT calls the methods in init(),start(),paint() sequence.
8. Stop () and destroy () are called when an applet is terminated.
9. Init() method is called only once during the runtime of the applet.
10. The paint() method is called each time applet output must be redrawn.
11. The paint() method has only one parameter of type Graphics.
12. An applet writes to its window only when its update or paint() method is called by AWT
13. To retrieve a parameter from applet, use get parameter method.
14. Swing-related classes are contained in javax.swing.
15. Applets that use swing must be subclasses of JApplet.
16. The content pane can be obtained by getContentPane() method
17. In swing, icons are encapsulated by ImageIcon class, which paints an icon from image.
18. A tabbed pane is a component that appears as a group of folders in file cabinet.
19. A tree is a component that presents a hierarchical view of the data.
20. We call addTab () to add a tab to the pane.

Sample Objective question papers:

OOPS THROUGH JAVA Objective Exam

Name: _____ Hall Ticket No.

						A			
--	--	--	--	--	--	----------	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. The _____ is the mechanism that binds together code and the data it manipulates. []
A. Encapsulation B. Inheritance C. Polymorphism D. Abstraction
2. The variables declared in interface are by default. []
A. static B. final C. array D. vector
3. The constructor that is used to duplicate an existing object is called _____ constructor. []
A. parameterized B. default C. copy D. duplicate
5. Defining several methods in a single class with same name but having different number or types of parameters is called Method _____. []
A. Overloading B. Overriding C. Copying D. Defining
5. Which of the following method does not belong to String class. []
A. length () B. compareTo () C. equals () D. strlen ()
6. A method declared as protected is not visible in. []
A. Same Package Non-subclasses. B. Different package subclasses.
C. Same Package subclasses. D. Different package Non-subclasses.
7. The _____ class is a super class of all classes. []
A. Object B. Thread C. Applet D. Graphics.
8. If we wish to prevent inheritance then we must declare class as _____. []
A. public B. final C. static D. abstract
9. The package that contains all the classes for implementing graphics related functions. []
A. java.lang B. java.io C. java.awt D. java.util
10. The package that support basic classes required to write java program is []
A. java.lang B. java.util C. java.io D. java.net

Contd....2

II. Fill in the blanks:

11. A class for which we can't create the object directly is _____
12. In java the objects are passed by the use of _____
13. In a class we may define _____ number of constructors.
14. _____ is a process by which one object acquires the properties of another object
15. The size of float variable is _____
16. For compilation & execution of java program we require _____
17. The _____ method is used to invoke a constructor of the same class
18. Abstract class means _____
19. Method overriding is resolved runtime by _____ mechanism
20. Multiple inheritance in java is accomplished by _____ feature

-oOo-

OOPS THROUGH JAVA

Objective Exam

Name: _____ Hall Ticket No.

						A				
--	--	--	--	--	--	----------	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. Defining several methods in a single class with same name but having different number or types of parameters is called Method _____ []
A. Overloading B. Overriding C. Copying D. Defining
2. Which of the following method does not belong to String class. []
A. length () B. compareTo () C. equals () D. strlen ()
3. A method declared as protected is not visible in. []
A. Same Package Non-subclasses. B. Different package subclasses.
C. Same Package subclasses. D. Different package Non-subclasses.
4. The _____ class is a super class of all classes. []
A. Object B. Thread C. Applet D. Graphics.
5. If we wish to prevent inheritance then we must declare class as _____ []
A. public B. final C. static D. abstract
6. The package that contains all the classes for implementing graphics related functions. []
A. java.lang B. java.io C. java.awt D. java.util
7. The package that support basic classes required to write java program is []
A. java.lang B. java.util C. java.io D. java.net
8. The _____ is the mechanism that binds together code and the data it manipulates. []
A. Encapsulation B. Inheritance C. Polymorphism D. Abstraction
9. The variables declared in interface are by default. []
A. static B. final C. array D. vector
10. The constructor that is used to duplicates an existing object is called _____ constructor. []
A. parameterized B. default C. copy D. duplicate

Contd...2

Code No: 05210301

:2:

Set No. 2

II. Fill in the blanks:

11. _____ is a process by which one object acquires the properties of another object
12. The size of float variable is _____
13. For compilation & execution of java program we require ____
14. The _____ method is used to invoke a constructor of the same class
15. Abstract class means _____
16. Method overriding is resolved runtime by _____ mechanism
17. Multiple inheritance in java is accomplished by _____ feature
18. A class for which we can't create the object directly is _____
19. In java the objects are passed by the use of _____
20. In a class we may define _____ number of constructors.

-oOo-

OOPS THROUGH JAVA

Objective Exam

Name: _____ Hall Ticket No.

						A				
--	--	--	--	--	--	----------	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. A method declared as protected is not visible in. []
A. Same Package Non-subclasses. B. Different package subclasses.
C. Same Package subclasses. D. Different package Non-subclasses.
2. The _____ class is a super class of all classes. []
A. Object B. Thread C. Applet D. Graphics.
3. If we wish to prevent inheritance then we must declare class as _____ []
A. public B. final C. static D. abstract
4. The package that contains all the classes for implementing graphics related functions. []
A. java.lang B. java.io C. java.awt D. java.util
5. The package that support basic classes required to write java program is []
A. java.lang B. java.util C. java.io D. java.net
6. The _____ is the mechanism that binds together code and the data it manipulates. []
A. Encapsulation B. Inheritance C. Polymorphism D. Abstraction
7. The variables declared in interface are by default. []
A. static B. final C. array D. vector
8. The constructor that is used to duplicates an existing object is called _____ constructor. []
A. parameterized B. default C. copy D. duplicate
9. Defining several methods in a single class with same name but having different number or types of parameters is called Method _____ []
A. Overloading B. Overriding C. Copying D. Defining
10. Which of the following method does not belong to String class. []
A. length () B. compareTo () C. equals () D. strlen ()

Contd...2

II. Fill in the blanks:

11. For compilation & execution of java program we require _____
12. The _____ method is used to invoke a constructor of the same class
13. Abstract class means _____
14. Method overriding is resolved runtime by _____ mechanism
15. Multiple inheritance in java is accomplished by _____ feature
16. A class for which we can't create the object directly is _____
17. In java the objects are passed by the use of _____
18. In a class we may define _____ number of constructors.
19. _____ is a process by which one object acquires the properties of another object
20. The size of float variable is _____

-oOo-

OOPS THROUGH JAVA

Objective Exam

Name: _____ Hall Ticket No.

						A				
--	--	--	--	--	--	----------	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. If we wish to prevent inheritance then we must declare class as _____ []
A. public B. final C. static D. abstract
2. The package that contains all the classes for implementing graphics related functions. []
A. java.lang B. java.io C. java.awt D. java.util
3. The package that support basic classes required to write java program is []
A. java.lang B. java.util C. java.io D. java.net
4. The _____ is the mechanism that binds together code and the data it manipulates. []
A. Encapsulation B. Inheritance C. Polymorphism D. Abstraction
5. The variables declared in interface are by default. []
A. static B. final C. array D. vector
6. The constructor that is used to duplicates an existing object is called _____ constructor. []
A. parameterized B. default C. copy D. duplicate
7. Defining several methods in a single class with same name but having different number or types of parameters is called Method _____ []
A. Overloading B. Overriding C. Copying D. Defining
8. Which of the following method does not belong to String class. []
A. length () B. compareTo () C. equals () D. strlen ()
9. A method declared as protected is not visible in. []
A. Same Package Non-subclasses. B. Different package subclasses.
C. Same Package subclasses. D. Different package Non-subclasses.
10. The _____ class is a super class of all classes. []
A. Object B. Thread C. Applet D. Graphics.

Contd...2

II. Fill in the blanks:

11. Abstract class means _____
12. Method overriding is resolved runtime by ____mechanism
13. Multiple inheritance in java is accomplished by _____feature
14. A class for which we can't create the object directly is _____
15. In java the objects are passed by the use of _____
16. In a class we may define _____number of constructors.
17. _____is a process by which one object acquires the properties of another object
18. The size of float variable is _____
19. For compilation & execution of java program we require _____
20. The _____method is used to invoke a constructor of the same class

OOPS THROUGH JAVA
Keys

I. Choose the correct alternative:

1. a
2. b
3. c
4. a
5. d
6. d
7. a
8. b
9. c
10. a

II. Fill in the blanks:

11. Abstract class
12. call by reference
13. many
14. inheritance
15. 4 bytes
16. compiler & interpreter
17. new
18. A class that cannot be instantiated
19. dynamic method dispatch
20. interface

^^**^^

Code No: 05210301

Set No. 1

OOPS THROUGH JAVA Objective Exam

Name: _____ Hall Ticket No.

					A				
--	--	--	--	--	---	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. Which is a checked Runtime Exception? []
A. NullPointerException B. InterruptedException
C. ArithmeticException D. ArrayIndexOutOfBoundsException
2. Which of the following method is not defined by MouseListener Interface. []
A. mouseClicked B. mouseDragged C. mouseReleased D. mouseExited
3. Which of the following layout is used as default layout manager? []
A. BorderLayout B. CardLayout C. FlowLayout D. GridLayout
4. The following method is called when we leave a web page that contains an applet []
A. pause() B. stop() C. destroy() D. hide()
5. Which of the following method doesn't belong to Thread. []
A. isAlive B. join C. sleep D. wake
6. Which event is generated when a scrollbar is manipulated? []
A. Item Event B. Adjustment Event C. Check Event D. Text Event
7. Which block following will execute whether or not an exception is thrown? []
A. Try B. Catch C. Throw D. Finally
8. Which of the following is the class not used networking. []
A. DatagramPacket B. DatagramSocket C. InetAddress D. HTTPAddress
9. Which of the following is the valid priority we can use for thread? []
A. MIN_PRIORITY B. MINIMUM_PRIORITY
C. LOW_PRIORITY D. ZERO_PRIORITY
10. Which listener interface is needed in handling TextField? []
A. ActionListener B. ItemListener C. TextListener D. InputListener

Cont...2

II. Fill in the blanks:

11. _____ is used to connect Java's I/O system to other programs.
12. The _____ allows us to pass parameters to the Applet through HTML page.
13. TCP/IP is used to implement _____ connection.
14. The class EventObject is defined in _____ package.
15. The _____ function can be used to find IP address of the host machine.
16. The fundamental class of Java swing JApplet extends _____ class.
17. BorderLayout manager divides window in to _____ areas.
18. A try block may have _____ number of catch block(s).
19. To select or to change the font we have to use _____ method.
20. At the top of the AWT hierarchy is the _____ class.

-oOo-

Name: _____ Hall Ticket No.

					A				
--	--	--	--	--	---	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. The following method is called when we leave a web page that contains an applet []
A. pause() B. stop() C. destroy() D. hide()
2. Which of the following method doesn't belong to Thread. []
A. isAlive B. join C. sleep D. wake
3. Which event is generated when a scrollbar is manipulated? []
A. Item Event B. Adjustment Event C. Check Event D. Text Event
4. Which block following will execute whether or not an exception is thrown? []
A. Try B. Catch C. Throw D. Finally
5. Which of the following is the class not used networking. []
A. DatagramPacket B. DatagramSocket C. InetAddress D. HTTPAddress
6. Which of the following is the valid priority we can use for thread? []
A. MIN_PRIORITY B. MINIMUM_PRIORITY
C. LOW_PRIORITY D. ZERO_PRIORITY
7. Which listener interface is needed in handling TextField? []
A. ActionListener B. ItemListener C. TextListener D. InputListener
8. Which is a checked Runtime Exception? []
A. NullPointerException B. InterruptedException
C. ArithmeticException D. ArrayIndexOutOfBoundsException
9. Which of the following method is not defined by MouseListener Interface. []
A. mouseClicked B. mouseDragged C. mouseReleased D. mouseExited
10. Which of the following layout is used as default layout manager? []
A. BorderLayout B. CardLayout C. FlowLayout D. GridLayout

Cont...2

II. Fill in the blanks:

11. The class EventObject is defined in _____package.
12. The _____function can be used to find IP address of the host machine.
13. The fundamental class of Java swing JApplet extends _____class.
14. BorderLayout manager divides window in to _____areas.
15. A try block may have _____number of catch block(s).
16. To select or to change the font we have to use _____method.
17. At the top of the AWT hierarchy is the _____class.
18. _____is used to connect Java's I/O system to other programs
19. The _____allows us to pass parameters to the Applet through HTML page.
20. TCP/IP is used to implement _____connection

OOPS THROUGH JAVA Objective Exam

Name: _____ Hall Ticket No.

					A				
--	--	--	--	--	---	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. Which event is generated when a scrollbar is manipulated? []
A. Item Event B. Adjustment Event C. Check Event D. Text Event
2. Which block following will execute whether or not an exception is thrown? []
A. Try B. Catch C. Throw D. Finally
3. Which of the following is the class not used networking. []
A. DatagramPacket B. DatagramSocket C. InetAddress D. HTTPAddress
4. Which of the following is the valid priority we can use for thread? []
A. MIN_PRIORITY B. MINIMUM_PRIORITY
C. LOW_PRIORITY D. ZERO_PRIORITY
5. Which listener interface is needed in handling TextField? []
A. ActionListener B. ItemListener C. TextListener D. InputListener
6. Which is a checked Runtime Exception? []
A. NullPointerException B. InterruptedException
C. ArithmeticException D. ArrayIndexOutOfBoundsException
7. Which of the following method is not defined by MouseListener Interface. []
A. mouseClicked B. mouseDragged C. mouseReleased D. mouseExited
8. Which of the following layout is used as default layout manager? []
A. BorderLayout B. CardLayout C. FlowLayout D. GridLayout
9. The following method is called when we leave a web page that contains an applet []
A. pause() B. stop() C. destroy() D. hide()
10. Which of the following method doesn't belong to Thread. []
A. isAlive B. join C. sleep D. wake

Cont...2

II. Fill in the blanks:

11. The fundamental class of Java swing JApplet extends _____ class.
12. BorderLayout manager divides window in to _____ areas.
13. A try block may have _____ number of catch block(s).
14. To select or to change the font we have to use _____ method.
15. At the top of the AWT hierarchy is the _____ class.
16. _____ is used to connect Java's I/O system to other programs.
17. The _____ allows us to pass parameters to the Applet through HTML page.
18. TCP/IP is used to implement _____ connection.
19. The class EventObject is defined in _____ package.
20. The _____ function can be used to find IP address of the host machine.

Code No: 05210301

Set No. 4

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

IV B.Tech. I Sem., II Mid-Term Examinations, Oct / Nov. – 2009

OOPS THROUGH JAVA

Objective Exam

Name: _____ Hall Ticket No.

						A				
--	--	--	--	--	--	---	--	--	--	--

Answer All Questions. All Questions Carry Equal Marks. Time: 20 Min. Marks: 20.

I. Choose the correct alternative:

1. Which of the following is the class not used networking. []
A. DatagramPacket B. DatagramSocket C. InetAddress D. HTTPAddress
2. Which of the following is the valid priority we can use for thread? []
A. MIN_PRIORITY B. MINIMUM_PRIORITY
C. LOW_PRIORITY D. ZERO_PRIORITY
3. Which listener interface is needed in handling TextField? []
A. ActionListener B. ItemListener C. TextListener D. InputListener
4. Which is a checked Runtime Exception? []
A. NullPointerException B. InterruptedException
C. ArithmeticException D. ArrayIndexOutOfBoundsException
5. Which of the following method is not defined by MouseListener Interface. []
A. mouseClicked B. mouseDragged C. mouseReleased D. mouseExited
6. Which of the following layout is used as default layout manager? []
A. BorderLayout B. CardLayout C. FlowLayout D. GridLayout
7. The following method is called when we leave a web page that contains an applet []
A. pause() B. stop() C. destroy() D. hide()
8. Which of the following method doesn't belong to Thread. []
A. isAlive B. join C. sleep D. wake
9. Which event is generated when a scrollbar is manipulated? []
A. Item Event B. Adjustment Event C. Check Event D. Text Event
10. Which block following will execute whether or not an exception is thrown? []
A. Try B. Catch C. Throw D. Finally

Cont...2

II. Fill in the blanks:

11. A try block may have _____ number of catch block(s).
12. To select or to change the font we have to use _____ method.
13. At the top of the AWT hierarchy is the _____ class.
14. _____ is used to connect Java's I/O system to other programs.
15. The _____ allows us to pass parameters to the Applet through HTML page.
16. TCP/IP is used to implement _____ connection.
17. The class EventObject is defined in _____ package.
18. The _____ function can be used to find IP address of the host machine.
19. The fundamental class of Java swing JApplet extends _____ class.
20. BorderLayout manager divides window in to _____ areas.

OOPS THROUGH JAVA I.

Choose the correct alternative:

1. b
2. b
3. c
4. b
5. d
6. b
7. d
8. d
9. a
10. a

II Fill in the blanks:

11. Socket
12. PARAM
13. Bi-Directional
14. java.util
15. getHostAddress()
16. Applet
17. 5
18. any
19. setFont()
20. Component

ADD-ON CONTENT

JDBC PROGRAMMING

Working with leaders in the database field, JavaSoft developed a single API for database access--JDBC. As part of this process, they kept three main goals in mind:

- JDBC should be an SQL-level API.
- JDBC should capitalize on the experience of existing database APIs.
- JDBC should be simple.

An SQL-level API means that JDBC allows us to construct SQL statements and embed them inside Java API calls. In short, you are basically using SQL. But JDBC lets you smoothly translate between the world of the database and the world of the Java application. Your results from the database, for instance, are returned as Java variables, and access problems get thrown as exceptions. Later on in the book, we go a step further and talk about how we can completely hide the existence of the database from a Java application using a database class library.

Because of the confusion caused by the proliferation of proprietary database access APIs, the idea of a universal database access API to solve this problem is not a new one. In fact, JavaSoft drew upon the successful aspects of one such API, Open DataBase Connectivity (ODBC). ODBC was developed to create a single standard for database access in the Windows environment. Although the industry has accepted ODBC as the primary means of talking to databases in Windows, it does not translate well into the Java world. First of all, ODBC is a C API that requires intermediate APIs for other languages. But even for C developers, ODBC has suffered from an overly complex design that has made its transition outside of the controlled Windows environment a failure. ODBC's complexity arises from the fact that complex, uncommon tasks are wrapped up in the API with its simpler and more common functionality. In other words, in order for you to understand a little of ODBC, you have to understand a lot.

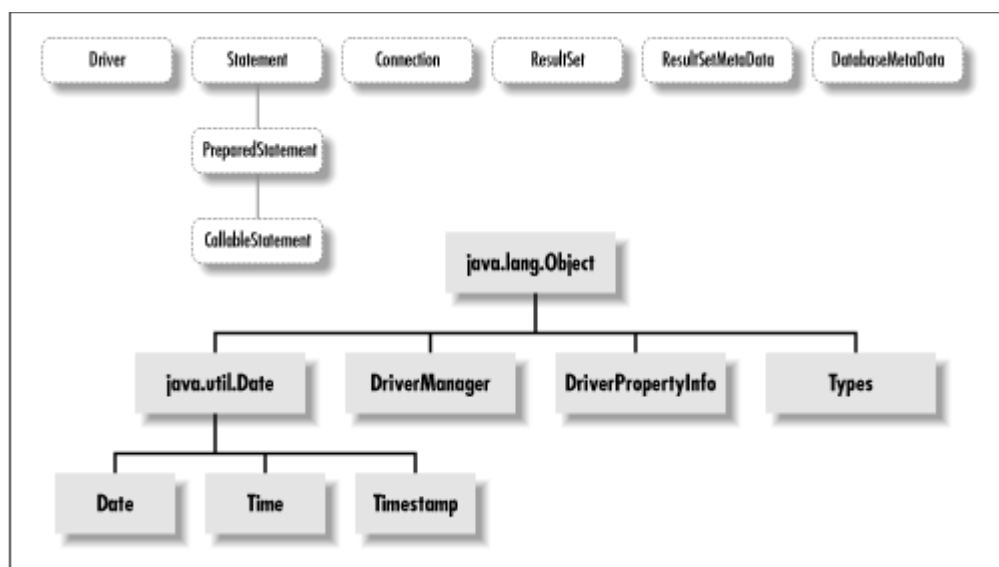
In addition to ODBC, JDBC is heavily influenced by existing database programming APIs such as X/OPEN SQL Call Level Interface. JavaSoft wanted to re-use the key abstractions from these APIs, which would ease acceptance by database vendors and capitalize on the existing knowledge capital of ODBC and SQL CLI developers. In addition, JavaSoft also realized that deriving an API from existing ones can provide quick development of solutions for database engines that support the old protocols. Specifically, JavaSoft worked in parallel with Intersolv to create an ODBC bridge that maps JDBC calls to ODBC calls, thus giving Java applications access to any database management system (DBMS) that supports ODBC.

JDBC attempts to remain as simple as possible while providing developers with maximum flexibility. A key criterion employed by JavaSoft is simply asking whether database access applications read well. The simple and common tasks use simple interfaces, while more uncommon or bizarre tasks are enabled through extra interfaces. For example, three interfaces handle a vast majority of database access. JDBC nevertheless provides several other interfaces for handling more complex and unusual tasks.

THE STRUCTURE OF JDBC

JDBC accomplishes its goals through a set of Java interfaces, each implemented differently by individual vendors. The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver. In building a database application, you do not have to think about the implementation of these underlying classes at all; the whole point of JDBC is to hide the specifics of each database and let you worry about just your application. [Figure 4-1](#) shows the JDBC classes and interfaces.

Figure 4-1. The classes and interfaces of java.sql, the JDBC API package



If you think about a database query for any database engine, it requires you to connect to the database, issue your SELECT statement, and process the result set. In [Example 4-1](#), we have the full code listing for a simple SELECT application from the Imaginary JDBC Driver for mSQL. [\[1\]](#) I wrote this driver for the Center for Imaginary Environments (<http://www.imaginary.com>), which is a non-commercial organization that promotes the development of virtual environment technologies like muds. This application is a single class that gets all of the rows from a table in an mSQL database located on my Sun box. First, it connects to the database by getting a database connection under my user id, borg, from the JDBC DriverManager class. It uses that database connection to create a Statement object that performs the SELECT query. A ResultSet object then provides the application with the key and val fields from the t_test table.

JDBC SERVLET PROGRAMMING

The Java Servlet API, introduced as the first standard extension to Java, provides a generic mechanism to extend the functionality of any kind of server. Servlets are most commonly used, however, to extend Web servers, performing tasks traditionally handled by CGI programs. Web servers that can support servlets include: Apache, Netscape's FastTrack and Enterprise Servers, Microsoft's IIS, O'Reilly's WebSite, and JavaSoft's Java Web Server.

The beauty of servlets is that they execute within the Web server's process space and they persist between invocations. This gives servlets tremendous performance benefits over CGI programs. Yet because they're written in Java, servlets are far less likely to crash a Web server than a C-based NSAPI or ISAPI extension. Servlets have full access to the various Java APIs and to third-party component classes, making them ideal for use in communicating with applets, databases, and RMI servers. Plus, servlets are portable between operating systems and between servers -- with servlets you can "write once, serve everywhere."

Java Servlet Programming covers everything you need to know to write effective servlets and includes numerous examples that you can use as the basis for your own servlets. The book explains the servlet life cycle, showing how you can use servlets to maintain state information effortlessly. It also describes how to serve dynamic Web content, including both HTML pages and multimedia data. Finally, it explores more advanced topics like integrated session tracking, efficient database connectivity using JDBC, applet-servlet communication, inter-servlet communication, and internationalization.

PACKAGE JAVA.SQL

Provides the API for accessing and processing data stored in a data source (usually a relational database) using the JavaTM programming language.

See:

[Description](#)

Interface Summary	
Array	The mapping in the Java programming language for the SQL type ARRAY.
Blob	The representation (mapping) in the Java TM programming language of an SQL BLOB value.
CallableStatement	The interface used to execute SQL stored procedures.
Clob	The mapping in the Java TM programming language for the SQL CLOB type.
Connection	A connection (session) with a specific database.
DatabaseMetaData	Comprehensive information about the database as a whole.

<u>Driver</u>	The interface that every driver class must implement.
<u>ParameterMetaData</u>	An object that can be used to get information about the types and properties of the parameters in a PreparedStatement object.
<u>PreparedStatement</u>	An object that represents a precompiled SQL statement.
<u>Ref</u>	The mapping in the Java programming language of an SQL REF value, which is a reference to an SQL structured type value in the database.
<u>ResultSet</u>	A table of data representing a database result set, which is usually generated by executing a statement that queries the database.
<u>ResultSetMetaData</u>	An object that can be used to get information about the types and properties of the columns in a ResultSet object.
<u>Savepoint</u>	The representation of a savepoint, which is a point within the current transaction that can be referenced from the Connection.rollback method.
<u>SQLData</u>	The interface used for the custom mapping of an SQL user-defined type (UDT) to a class in the Java programming language.
<u>SQLInput</u>	An input stream that contains a stream of values representing an instance of an SQL structured type or an SQL distinct type.
<u>SQLOutput</u>	The output stream for writing the attributes of a user-defined type back to the database.
<u>Statement</u>	The object used for executing a static SQL statement and returning the results it produces.
<u>Struct</u>	The standard mapping in the Java programming language for an SQL structured type.

Class Summary

<u>Date</u>	A thin wrapper around a millisecond value that allows JDBC to identify this as an SQL DATE value.
<u>DriverManager</u>	The basic service for managing a set of JDBC drivers. NOTE: The DataSource interface, new in the JDBC 2.0 API, provides another way to connect to a data source.

<u>DriverPropertyInfo</u>	Driver properties for making a connection.
<u>SQLPermission</u>	The permission for which the SecurityManager will check when code that is running in an applet calls the DriverManager.setLogWriter method or the DriverManager.setLogStream (deprecated) method.
<u>Time</u>	A thin wrapper around the java.util.Date class that allows the JDBC API to identify this as an SQL TIME value.
<u>Timestamp</u>	A thin wrapper around java.util.Date that allows the JDBC API to identify this as an SQL TIMESTAMP value.
<u>Types</u>	The class that defines the constants that are used to identify generic SQL types, called JDBC types.

Exception Summary

<u>BatchUpdateException</u>	An exception thrown when an error occurs during a batch update operation.
<u>DataTruncation</u>	An exception that reports a DataTruncation warning (on reads) or throws a DataTruncation exception (on writes) when JDBC unexpectedly truncates a data value.
<u>SQLException</u>	An exception that provides information on a database access error or other errors.
<u>SQLWarning</u>	An exception that provides information on database access warnings.

PACKAGE JAVA.SQL DESCRIPTION

Provides the API for accessing and processing data stored in a data source (usually a relational database) using the Java™ programming language. This API includes a framework whereby different drivers can be installed dynamically to access different data sources. Although the JDBC™ API is mainly geared to passing SQL statements to a database, it provides for reading and writing data from any data source with a tabular format. The reader/writer facility, available through the javax.sql.RowSet group of interfaces, can be customized to use and update data from a spread sheet, flat file, or any other tabular data source.

WHAT THE JDBC™ 3.0 API INCLUDES

The JDBC™ 3.0 API includes both the java.sql package, referred to as the JDBC core API, and the javax.sql package, referred to as the JDBC Optional Package API. This complete JDBC API is included in the Java™ 2 SDK, Standard Edition (J2SE™), version 1.4. The javax.sql package extends the functionality of the JDBC API from a client-side API to a server-side API, and it is an essential part of the Java™ 2 SDK, Enterprise Edition (J2EE™) technology. (Note that the J2EE platform also includes the complete JDBC API; features new in the JDBC 3.0 API are included in the J2EE version 1.3).

VERSIONS

The JDBC 3.0 API incorporates all of the previous JDBC API versions:

- The JDBC 2.1 core API
- The JDBC 2.0 Optional Package API
(Note that the JDBC 2.1 core API and the JDBC 2.0 Optional Package API together are referred to as the JDBC 2.0 API.)
- The JDBC 1.2 API
- The JDBC 1.0 API

Classes, interfaces, methods, fields, constructors, and exceptions have the following "since" tags that indicate when they were introduced into the Java platform. When these "since" tags are used in Javadoc™ comments for the JDBC API, they indicate the following:

- Since 1.4 -- new in the JDBC 3.0 API and part of the J2SE platform, version 1.4
- Since 1.2 -- new in the JDBC 2.0 API and part of the J2SE platform, version 1.2
- Since 1.1 or no "since" tag -- in the original JDBC 1.0 API and part of the JDK™, version 1.1

NOTE: Many of the new features are optional; consequently, there is some variation in drivers and the features they support. Always check your driver's documentation to see whether it supports a feature before you try to use it.

NOTE: The class `SQLPermission` was added in the Java™ 2 SDK, Standard Edition, version 1.3 release. This class is used to prevent unauthorized access to the logging stream associated with the `DriverManager`, which may contain information such as table names, column data, and so on.

WHAT THE JAVA.SQL PACKAGE CONTAINS

The java.sql package contains API for the following:

- Making a connection with a database via the `DriverManager` facility
 - `DriverManager` class -- makes a connection with a driver
 - `SQLPermission` class -- provides permission when code running within a Security Manager, such as an applet, attempts to set up a logging stream through the `DriverManager`
 - `Driver` interface -- provides the API for registering and connecting drivers based on JDBC technology ("JDBC drivers"); generally used only by the `DriverManager` class

- DriverManager class -- provides properties for a JDBC driver; not used by the general user
- Sending SQL statements to a database
 - Statement -- used to send basic SQL statements
 - PreparedStatement -- used to send prepared statements or basic SQL statements (derived from Statement)
 - CallableStatement -- used to call database stored procedures (derived from PreparedStatement)
 - Connection interface -- provides methods for creating statements and managing connections and their properties
 - Savepoint -- provides savepoints in a transaction
- Retrieving and updating the results of a query
 - ResultSet interface
- Standard mappings for SQL types to classes and interfaces in the Java programming language
 - Array interface -- mapping for SQL ARRAY
 - Blob interface -- mapping for SQL BLOB
 - Clob interface -- mapping for SQL CLOB
 - Date class -- mapping for SQL DATE
 - Ref interface -- mapping for SQL REF
 - Struct interface -- mapping for SQL STRUCT
 - Time class -- mapping for SQL TIME
 - Timestamp class -- mapping for SQL TIMESTAMP
 - Types class -- provides constants for SQL types
- Custom mapping an SQL user-defined type (UDT) to a class in the Java programming language
 - SQLData interface -- specifies the mapping of a UDT to an instance of this class
 - SQLInput interface -- provides methods for reading UDT attributes from a stream
 - SQLOutput interface -- provides methods for writing UDT attributes back to a stream
- Metadata
 - DatabaseMetaData interface -- provides information about the database
 - ResultSetMetaData interface -- provides information about the columns of a ResultSet object
 - ParameterMetaData interface -- provides information about the parameters to PreparedStatement commands
- Exceptions
 - SQLException -- thrown by most methods when there is a problem accessing data and by some methods for other reasons
 - SQLWarning -- thrown to indicate a warning
 - DataTruncation -- thrown to indicate that data may have been truncated
 - BatchUpdateException -- thrown to indicate that not all commands in a batch update executed successfully

JAVA.SQL AND JAVAX.SQL FEATURES INTRODUCED IN THE JDBC 3.0 API

- Pooled statements -- reuse of statements associated with a pooled connection
- Savepoints -- allow a transaction to be rolled back to a designated savepoint

- Properties defined for `ConnectionPoolDataSource` -- specify how connections are to be pooled
- Metadata for parameters of a `PreparedStatement` object
- Ability to retrieve values from automatically generated columns
- Ability to have multiple `ResultSet` objects returned from `CallableStatement` objects open at the same time
- Ability to identify parameters to `CallableStatement` objects by name as well as by index
- `ResultSet` holdability -- ability to specify whether cursors should be held open or closed at the end of a transaction
- Ability to retrieve and update the SQL structured type instance that a `Ref` object references
- Ability to programmatically update `BLOB`, `CLOB`, `ARRAY`, and `REF` values.
- Addition of the `java.sql.Types.DATALINK` data type -- allows JDBC drivers access to objects stored outside a data source
- Addition of metadata for retrieving SQL type hierarchies

JAVA.SQL FEATURES INTRODUCED IN THE JDBC 2.1 CORE API

- Scrollable result sets--using new methods in the `ResultSet` interface that allow the cursor to be moved to a particular row or to a position relative to its current position
- Batch updates
- Programmatic updates--using `ResultSet` updater methods
- New data types--interfaces mapping the SQL3 data types
- Custom mapping of user-defined types (UDTs)
- Miscellaneous features, including performance hints, the use of character streams, full precision for `java.math.BigDecimal` values, additional security, and support for time zones in date, time, and timestamp values.

JAVAX.SQL FEATURES INTRODUCED IN THE JDBC 2.0 OPTIONAL PACKAGE API

- The `DataSource` interface as a means of making a connection. The Java Naming and Directory Interface™ (JNDI) is used for registering a `DataSource` object with a naming service and also for retrieving it.
- Pooled connections -- allowing connections to be used and reused
- Distributed transactions -- allowing a transaction to span diverse DBMS servers
- `RowSet` technology -- providing a convenient means of handling and passing data

CUSTOM MAPPING OF UDTs

A user-defined type (UDT) defined in SQL can be mapped to a class in the Java programming language. An SQL structured type or an SQL `DISTINCT` type are the UDTs that may be custom mapped. The following three steps set up a custom mapping:

1. Defining the SQL structured type or `DISTINCT` type in SQL
2. Defining the class in the Java programming language to which the SQL UDT will be mapped. This class must implement the `SQLData` interface.
3. Making an entry in a `Connection` object's type map that contains two things:
 - the fully-qualified SQL name of the UDT

- the Class object for the class that implements the `SQLData` interface

When these are in place for a UDT, calling the methods `ResultSet.getObject` or `CallableStatement.getObject` on that UDT will automatically retrieve the custom mapping for it. Also, the `PreparedStatement.setObject` method will automatically map the object back to its SQL type to store it in the data source.

PACKAGE SPECIFICATION

- [Specification of the JDBC 3.0 API](#)

PACKAGE JAVAX.SERVLET

The `javax.servlet` package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.

See:

[Description](#)

Interface Summary	
Filter	A filter is an object that performs filtering tasks on either the request to a resource (a servlet or static content), or on the response from a resource, or both.
FilterChain	A <code>FilterChain</code> is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource.
FilterConfig	A filter configuration object used by a servlet container to pass information to a filter during initialization.
RequestDispatcher	Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server.
Servlet	Defines methods that all servlets must implement.
ServletConfig	A servlet configuration object used by a servlet container to pass information to a servlet during initialization.
ServletContext	Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file.

<u>ServletContextAttributeListener</u>	Implementations of this interface receive notifications of changes to the attribute list on the servlet context of a web application.
<u>ServletContextListener</u>	Implementations of this interface receive notifications about changes to the servlet context of the web application they are part of.
<u>ServletRequest</u>	Defines an object to provide client request information to a servlet.
<u>ServletRequestAttributeListener</u>	A ServletRequestAttributeListener can be implemented by the developer interested in being notified of request attribute changes.
<u>ServletRequestListener</u>	A ServletRequestListener can be implemented by the developer interested in being notified of requests coming in and out of scope in a web component.
<u>ServletResponse</u>	Defines an object to assist a servlet in sending a response to the client.
<u>SingleThreadModel</u>	Deprecated. <i>As of Java Servlet API 2.4, with no direct replacement.</i>

Class Summary

<u>GenericServlet</u>	Defines a generic, protocol-independent servlet.
<u>ServletContextAttributeEvent</u>	This is the event class for notifications about changes to the attributes of the servlet context of a web application.
<u>ServletContextEvent</u>	This is the event class for notifications about changes to the servlet context of a web application.
<u>ServletInputStream</u>	Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time.
<u>ServletOutputStream</u>	Provides an output stream for sending binary data to the client.
<u>ServletRequestAttributeEvent</u>	This is the event class for notifications of changes to the

	attributes of the servlet request in an application.
ServletRequestEvent	Events of this kind indicate lifecycle events for a ServletRequest.
ServletRequestWrapper	Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet.
ServletResponseWrapper	Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet.

Exception Summary	
ServletException	Defines a general exception a servlet can throw when it encounters difficulty.
UnavailableException	Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

PACKAGE JAVAX.SERVLET DESCRIPTION

The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.