UNIT-1
OOPS CONCEPTS AND JAVA PROGRAMMING

Everywhere you look in the real world you see objects—people, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. Telephones, houses, traffic lights, microwave ovens and water coolers are just a few more objects. Computer programs, such as the Java programs you'll read in this book and the ones you'll write, are composed of lots of interacting software objects.

We sometimes divide objects into two categories: animate and inanimate. Animate objects are ―alive‖ in some sense—they move around and do things. Inanimate objects, on the other hand, do not move on their own .Objects of both types, however, have some things in common. They all have attributes (e.g., size, shape, color and weight), and they all exhibit behaviors (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleep crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water). We will study the kinds of attributes and behaviors that software objects have. Humans learn about existing objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Object-oriented design provides a natural and intuitive way to view the software design process—namely, modeling objects by their attributes and behaviors just as we describe real-world objects. OOD also models communication between objects. Just as people send messages to one another (e.g., a sergeant commands a soldier to stand at attention), objects also communicate via messages. A bank account object may receive a message to decrease its balance by a certain amount because the customer has withdrawn that amount of money.

Object-Oriented:

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's ―everything is an object‖ paradigm and the pragmatist's ―stay out of my way‖ model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

OOD encapsulates (i.e., wraps) attributes and operations (behaviors) into objects, an object's attributes and operations are intimately tied together. Objects have the property of information hiding. This means that objects may know how to communicate with one another across well-defined interfaces, but normally they are not allowed to know how other objects are implemented ,implementation details are hidden within the objects themselves. We can drive a car effectively, for instance, without knowing the details of how engines, transmissions, brakes and exhaust systems work internally—as long as we know how to use the accelerator pedal, the brake pedal, the wheel and so on. Information hiding, as we will see, is crucial to good software engineering.

Languages like Java are object oriented. Programming in such a language is called object-oriented programming (OOP), and it allows computer programmers to implement an object-oriented design as a working system. Languages like C, on the other hand, are procedural, so programming tends to be action oriented. In C, the unit of programming is the function. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. In Java, the unit of programming is the class from which objects are eventually instantiated (created). Java classes contain methods (which implement operations and are similar to functions in C) as well as fields (which implement attributes).

Java programmers concentrate on creating classes. Each class contains fields, and the set of methods that manipulate the fields and provide services to clients (i.e., other classes that use the class). The programmer uses existing classes as the building blocks for constructing new classes. Classes are to objects as blueprints are to houses. Just as we can build many houses from one blueprint, we can instantiate (create) many objects from one class.

Classes can have relationships with other classes. For example, in an object-oriented design of a bank, the ―bank teller‖ class needs to relate to the ―customer‖ class, the ―cash drawer‖ class, the ―safe‖ class, and so on. These relationships are called associations.

Packaging software as classes makes it possible for future software systems to reuse the classes. Groups of related classes are often packaged as reusable components. Just as realtors often say that the three most important factors affecting the price of real estate are ―location, location and location,‖ people in the software community often say that the three most important factors affecting the future of software development are ―reuse, reuse and reuse.‖ Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps programmers build more reliable and effective systems, because existing classes and components often have gone through extensive testing, debugging and performance tuning.

Indeed, with object technology, you can build much of the software you will need by combining classes, just as automobile manufacturers combine interchangeable parts. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software development efforts.

NEED FOR OOP PARADIGM:

Object-Oriented Programming:

Object-oriented programming is at the core of Java. In fact, all Java programs are object-oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

Two Paradigms of Programming:

As you know, 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 (that is, code). The process-oriented model can be thought of as code acting

on data. Procedural languages such as C employ this model to considerable success. Problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived.

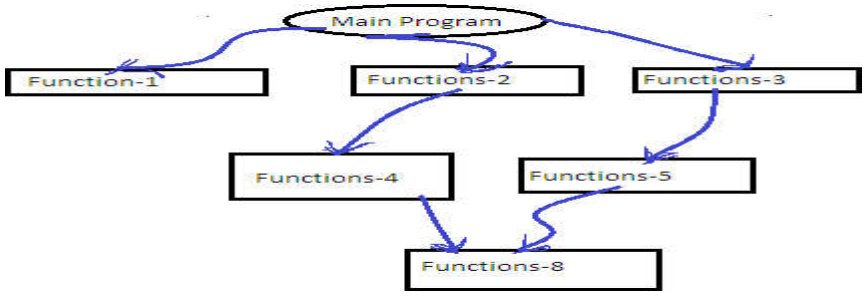
Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Procedure oriented Programming:

In this approach, the problem is always considered as a sequence of tasks to be done. A number of functions are written to accomplish these tasks. Here primary focus on –Functions‖ and little attention on data.

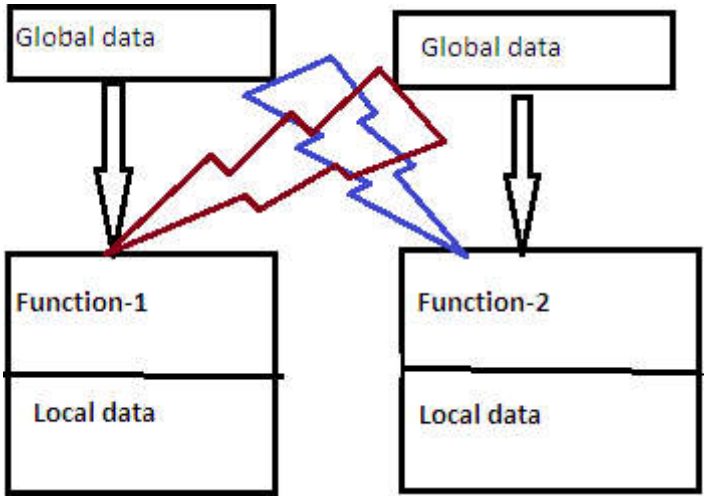
There are many high level languages like COBOL, FORTRAN, PASCAL, C used for conventional programming commonly known as POP.

POP basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.



Normally a flowchart is used to organize these actions and represent the flow of control logically sequential flow from one to another. In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an in advent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all the functions that access the data. This provides an opportunity for bugs to creep in.

Drawback: It does not model real world problems very well, because functions are action oriented and do not really corresponding to the elements of the problem.



Characteristics of POP:

- Emphasis is on doing actions.
- Large programs are divided into smaller programs known as functions.
- Most of the functions shared global data.
- Data move openly around the program from function to function.
- Functions transform data from one form to another.
- Employs top-down approach in program design.

OOP:

OOP allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

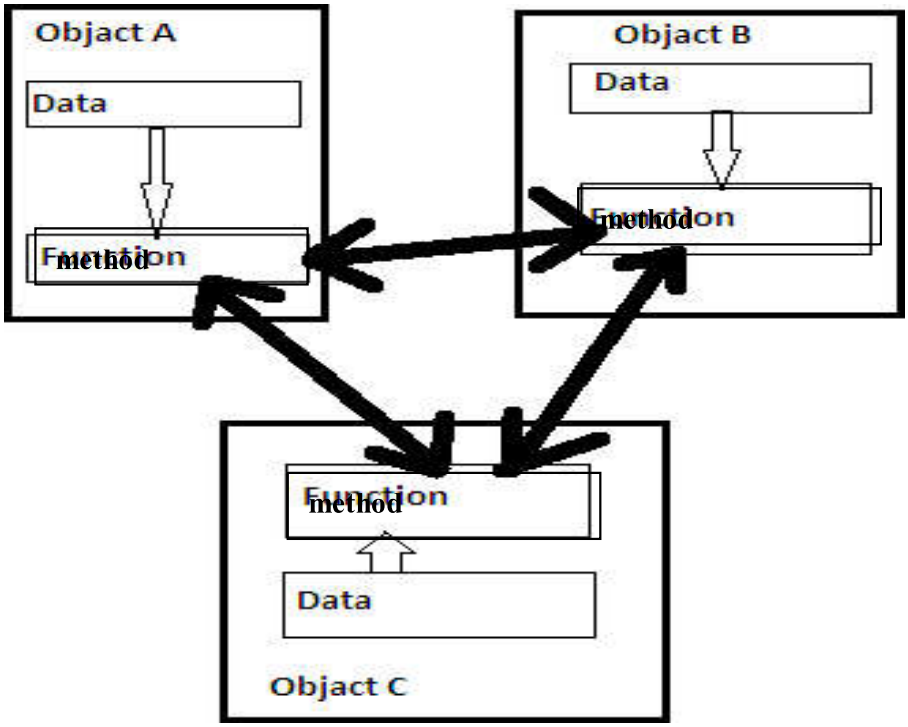
DEF: OOP is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of such modules on demand.

That is ,an object a considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

OOP Chars:

Emphasis on data .
Programs are divided into what are known as methods.
Data structures are designed such that they characterize the objects.
Methods that operate on the data of an object are tied together .
Data is hidden.
Objects can communicate with each other through methods.
Reusability.
Follows bottom-up approach in program design.

Organization of OOP:



Evolution of Computing and Programming: Computer use is increasing in almost every field of endeavor. Computing costs have been decreasing dramatically due to rapid developments in both hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago can now be inscribed on silicon chips smaller than a fingernail, costing perhaps a few dollars each. Fortunately, silicon is one of the most abundant materials on earth it is an ingredient in common sand. Silicon chip technology has made computing so economical that about a billion general-purpose computers are in use worldwide, helping people in business, industry and government, and in their personal lives. The number could easily double in the next few years. Over the years, many programmers learned the programming methodology called structured programming.

You will learn structured programming and an exciting newer methodology, object-oriented programming. Why do we teach both? Object orientation is the key programming methodology used by programmers today. You will create and work with many software objects in this text. But you will discover that their internal structure is often built using structured-programming techniques. Also, the logic of manipulating objects is occasionally expressed with structured programming.

Language of Choice for Networked Applications: Java has become the language of choice for implementing Internet-based applications and software for devices that communicate over a network. Stereos and other devices in homes are now being networked together by Java technology. At the May 2006 JavaOne conference, Sun announced that there were one billion java-enabled mobile phones and hand held devices! Java has evolved rapidly into the large-scale applications arena. It's the preferred language for meeting many organizations' enterprise-wide programming needs. Java has evolved so rapidly that this seventh edition of Java How to Program was published just 10 years after the first edition was published. Java has grown so large that it has two other editions. The Java Enterprise Edition (Java EE) is geared toward developing large-scale, distributed networking applications and web-based applications. The Java Micro Edition (Java ME) is geared toward developing applications for small, memory constrained devices, such as cell phones, pagers and PDAs.

Data Abstraction

An essential element of object-oriented programming is *abstraction.* Humans manage complexity through abstraction. For example, people do not think of a car as a set ofte ns of 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 to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work. Instead they are free to utilize the object as a whole.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other
complex system) through the use of hierarchical abstractions.

Encapsulation

An object encapsulates the methods and data that are contained inside it .the rest of the system interacts with an object only through a well defined set of services that it provides.

Inheritance

- I have more information about Flora – not necessarily because she is a florist but because she is a shopkeeper.
- One way to think about how I have organized my knowledge of Flora is in terms of a hierarchy of categories:

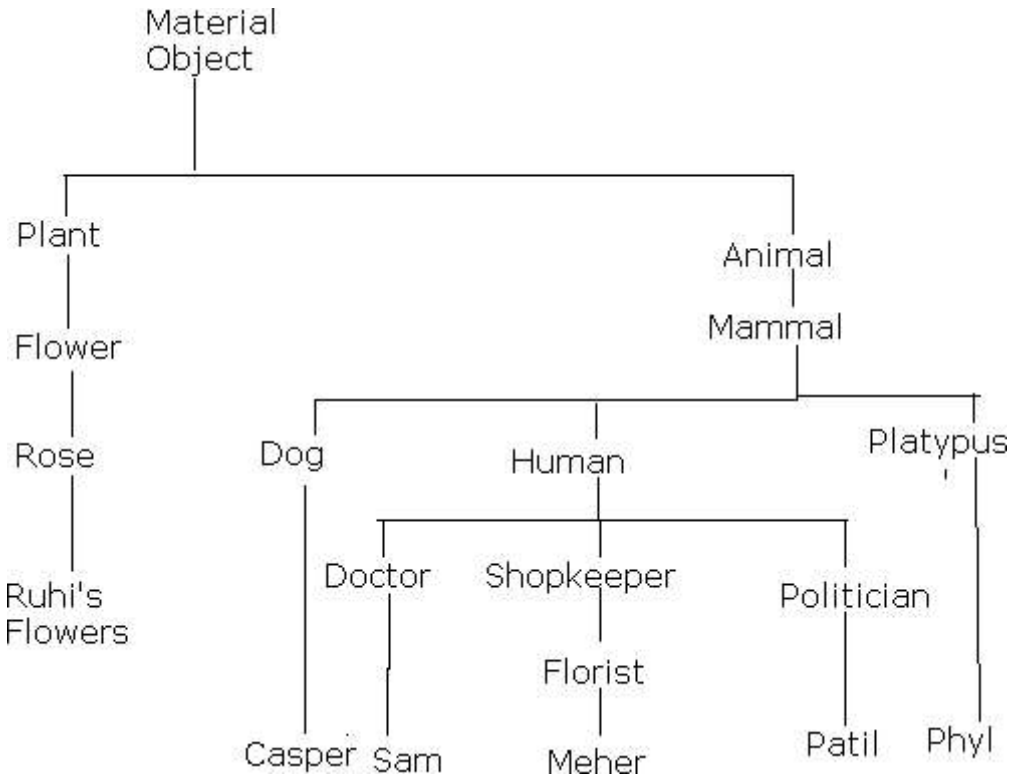


Fig : A Class Hierarchy for Different kinds of Material objects

CLASSES AND OBJECTS

classes and objects: Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapters primarily exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax.

Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. The general form of a **class** definition is shown here:

class *classname* {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(*parameter-list*) {
// body of method
}
type methodname2(p*arameter-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. Thus, it is the methods that determine how a class' data can be used.

Declaring Objects

As just explained, 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**.

Ex: Box mybox = new Box();

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object

A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new *classname*( );

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**.

Procedural	OO programming
Code is placed into totally distinct functions or procedures	Everything treated as an Object
Data placed in separate structures and is manipulated by these functions or procedures	Every object consist of attributes(data) and behaviors (methods)
Code maintenance and reuse is difficult	Code maintenance and reuse is easy
Data is uncontrolled and unpredictable (i.e. multiple functions may have access to the global data)	The data of an object can be accessed only by the methods associated with the object
You have no control over who has access to the data	Good control over data access
Testing and debugging are much more difficult	Testing and debugging are much easy
Not easy to upgrade	Easy to upgrade
Not easy to partition the work in a project	Easy to partition the work in a project

HISTORY OF JAVA

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first Working version. This language was initially called ―Oak‖ but was renamed ―Java‖ in 1995. Between the initial implementation of Oak in the fall of 1992 and the public Announcement of Java in the spring of 1995, many more people contributed to the designand evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lind Holm were key contributors to the maturing of the original prototype.

The trouble With C and C++ (and most other languages) is that they are designed to be compiled For a specific target. Although it is possible to compile a C++ program for just about Any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The Problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution,Gosling and others began work on a portable, platform-independent language thatcould be used to produce code that would run on a variety of CPUs under differing Environments. This effort ultimately led to the creation of Java.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers.

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++, most programmers have little trouble learning Java..

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. Borrowing liberally from many seminal object-software

environments of the last few decades, Java manages to strike a balance between the purist's –everything is an object‖ paradigm and the pragmatist's –stay out of my way‖ model.

Robust

The multi platformed 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 was given a high priority in the design of Java.

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 ask in traditional programming environments. For example, in C/C++, the pro grammer 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.

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. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess .synchronization that enables you to construct smoothly running interactive systems.

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 up grades, 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, any time, forever.‖ To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. 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. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs.

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 intra address-space messaging. This allowed objects on two different computers to execute procedures remotely. Java revived these interfaces in a package called *Remote MethodInvocation* (*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 bytecode may be dynamically updated on a running system.

DATA TYPES

Java defines eight simple (or elemental) 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

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other Computer languages, including C/C++, support both signed and unsigned integers.

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

byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128to 127. Variables of type **byte** are especially useful when you're working with a streamof data from a network or file. They are also useful when you're working with rawbinary data that may not be directly compatible with Java's other built-in types.

Syntax: byte b, c;

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called *big-endian* format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce.
Here are some examples of **short** variable declarations:
short s;
short t;
int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Any time you have an integer expression involving **byte**s, **short**s, **int**s, and literal numbers, the entire expression Is *promoted* to **int** before the calculation is done.
long

long is a signed 64-bit type and is useful for those occasions where an **int** type is notlarge enough to hold the desired value. The range of a **long** is quite large. This makesit useful when big, whole numbers are needed. For example, here is a program thatcomputes the number of miles that light will travel in a specified number of days.
Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type.
Their width and ranges are shown here:

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

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are

useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents.
Here are some example **float** variable declarations:
float hightemp, lowtemp;
double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations.

Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
public static void main(String args[]) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
System.out.println("Area of circle is " + a);
}
}
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integertype that is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to representcharacters.. There are no negative **char**s. The standard set of characters known asASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1,ranges from 0 to 255.
Booleans

Java has a simple type, called **boolean**, for logical values. It can have only one of twopossible values, **true** or **false**. This is the type returned by all relational operators, suc has **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**.
Here is a program that demonstrates the **boolean** type:

There are three interesting things to notice about this program. First, as you can see,when a **boolean** value is output by **println( )**, ‑true‖ or ‑false‖ is displayed. Second,the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. Thereis no need to write an **if** statement like this:
if(b == true) ...
Third, the outcome of a relational operator, such as <, is a **boolean** value. This is why the expression **10 > 9** displays the value ‑true.‖ Further, the extra set of parentheses around **10 > 9** is necessary because the + operator has a higher precedence than the >.
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. These elementsare examined next.

Declaring a Variable

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. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable.

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, initializing
                                        // d and f.
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

So far, all of the variables used have been declared at the start of the **main( )** method. However, Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curlybrace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a 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. However, these traditional scopes do not fit well with Java's strict, object oriented model. The scope defined by a method begins with its opening curly brace.

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

```
// Demonstrate block scope.
```

OPERATORS

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment

%=	Modulus assignment
– –	Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them on **boolean** types, but you can use them on **char** types, since the **char** type in Java is, essentially, a subset of **int**.

The Bitwise Operators

Java defines several *bitwise operators* which can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Relational Operators

The *relational operators* determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

The Assignment Operator

You have been using the assignment operator since Chapter 2. Now it is time to take a formal look at it. The *assignment operator* is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

var = *expression*;

Here, the type of *var* must be compatible with the type of *expression.*

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables **x**, **y**, and **z** to 100 using a single statement. This works because the **=** is an operator that yields the value of the right-hand expression. Thus, the value of **z = 100** is 100, which is then assigned to **y**, which in turn is assigned to **x**. Using a ―chain of assignment‖ is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types ofif-then-else statements. This operator is the **?**, and it works in Java much like it doesin C, C++, and C#. It can seem somewhat confusing at first, but the **?** can be used very effectively once mastered. The **?** has this general form:

expression1 **?** *expression2* **:** *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?** operation is that of the expression evaluated. Both *expression2* and *expression3* are required to return the same type, which can't be **void**.

Type Conversion and Casting

Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to 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.

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 met:

- 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.

It has this general form:

(*target-type*) *value*

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
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 you know, integers do not have fractional components.Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.
The following program demonstrates some type conversions that require casts:

// Demonstrate casts.

```
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);
}
}
```

This program generates the following 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
```

Enum

Enum in java is a data type that contains fixed set of constants.

It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc. The java enum constants are static and final implicitly. It is available from JDK 1.5.

Java Enums can be thought of as classes that have fixed set of constants.

```
class EnumExample1{
    public enum Season { WINTER, SPRING, SUMMER, FALL }
            public static void main(String[] args) {
            for (Season s : Season.values())
                    System.out.println(s);

                    }}
```

CONTROL STATEMENTS

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

if (*condition*) *statement1*;
else *statement2*;

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. The **else** clause is optional.

```
int a, b;
// ...
if(a < b) a = 0;
else b = 0;
```

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the ***if-else-if*** *ladder*. It looks like this:

```
if(condition)
statement;
else if(condition)
statement;
else if(condition)
statement;
...
else
statement;
```

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it

often provides a better alternative than a large series of **if-else-if** statements.
Here is the general form of a **switch** statement:

```
switch (expression) {
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
...
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what we commonly call *loops*. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

While

The **while** loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
While (condition) {
// body of loop
}
```

The *condition* can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When *condition* becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with.

Systex:

```
do {
// body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
// Demonstrate the do-while loop.
class DoWhile {
public static void main(String args[]) {
int n = 10;
do {
System.out.println("tick " + n);
n--;
} while(n > 0);
}
}
```

For

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful and versatile construct. Here is the general form of the **for** statement:

```
for(initialization; condition; iteration) {
// body
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loopcontrol variable,* which acts as a counter that controls the loop.. Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

```
// Demonstrate the for loop.
class ForTick {
public static void main(String args[]) {
int n;
for(n=10; n>0; n--)
System.out.println("tick " + n);
}
}
```

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a ‑civilized‖ form of goto. The last two uses are explained here.

Return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 7, a brief look at **return** is presented here.

As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller.

SIMPLE JAVA PROGRAM

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Arrays

An *array* is a group of like-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 elementin an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A *one-dimensional array* is, essentially, 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. The base type determines the data type of each element that comprises the array.

```
// Demonstrate a one-dimensional array.
class Array {

public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;

System.out.println("April has " + month_days[3] + " days.");
}
```

```
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you mightexpect, look and act like regular multidimensional arrays. However, as you will see there are a couple of subtle differences. 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**. Internally this matrix is implemented as an *array* of *arrays* of **int**.

```
// 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();
}
}
}
```

This program generates the following output:

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

As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two dimensional array in which the sizes of the second dimension are unequal.

Constructor

Constructor in java is a *special type of method* that is used to initialize the object. Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

There are basically two rules defined for the constructor.

- Constructor name must be same as its class name
- Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

Default constructor (no-arg constructor)
Parameterized constructor

Default constructor

```
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```

Parameterized constructor

```
class Student4{
   int id;
   String name;

   Student4(int i,String n){
   id = i;
   name = n;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control.*

How a member can be accessed is determined by the *access specifier* that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy. Java's access specifiers are **public**, **private**, and **protected**. Java also defines a
default access level. **protected** applies only when inheritance is involved. The other
access specifiers are described next.

Let's begin by defining **public** and **private**. When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code. When a member of a class is specified as **private**, then that member can only be accessed byother members of its class. Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

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. To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.
Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables.

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be

wondering how such 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. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

Overloading methods and constructors

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java implements polymorphism.

```
// Demonstrate method overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
```

```
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

```
No parameters a:
10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test( )** is overloaded four times.

Overloading Constructor

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
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;
}
}
```

Argument/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.* 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. As you will see, Java uses both approaches, depending upon what is passed.

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);
}
}
```

The output from this program is shown here:

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

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.*

```
// A simple example of recursion(factorial).
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));
}
}
```

The output from this program is shown here:
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

String class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string ‒This is a String, too‖ is a **String** constant. Fortunately, Java handles **String** constants in the same way that other computer languages handle ‒normal‖ strings, soyou don't have to worry about this.

The second thing to understand about strings is that objects of type **String** are immutable; once a **String** object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons:

■ If you need to change a string, you can always create a new one that contains the modifications.

■ Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java.
(**StringBuffer** is described in Part II of this book.)
Strings can be constructed a variety of ways. The easiest is to use a statement like this:
String myString = "this is a test";
Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**:
System.out.println(myString);
Java defines one operator for **String** objects: +. It is used to concatenate two strings.
For example, this statement

```
String myString = "I" + " like " + "Java.";
```

results in **myString** containing –I like Java.‖

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:

```
boolean equals(String object)
int length( )
char charAt(int index)
```

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " +
strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
System.out.println("strOb1 == strOb2");
else
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
```

```
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
public static void main(String args[]) {
String str[] = { "one", "two", "three" };
for(int i=0; i<str.length; i++)
System.out.println("str[" + i + "]: " +
str[i]);
}
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

UNIT-II

INHERITANCE, INTERFACES AND PACKAGES

- In OOP another important feature is Reusability which is very use full in two major issues of development :
  - Saving Time
  - Saving Memory
- In C++/Java the mechanism implements this reusability is Inheritance.

General form of a class declaration is:

```
Class subclass-name extends super class-name
{
    // body of super class
 }
```

```
// A simple example of inheritance.
// Create a superclass.
class A {
int i, j;
void showij() {
System.out.println("i and j: " + i + " " + j);
}
}
// Create a subclass by extending class A.
class B extends A {
int k;
void showk() {
System.out.println("k: " + k);
}
void sum() { System.out.println("i+j+k: "
+ (i+j+k));
}
}
class SimpleInheritance {
public static void main(String args[]) { A
superOb = new A();
B subOb = new B();
// The superclass may be used by itself.
superOb.i = 10;
superOb.j = 20; System.out.println("Contents of
superOb: "); superOb.showij();
System.out.println();
/* The subclass has access to all public members of its
superclass. */
subOb.i = 7;
subOb.j = 8;
```

```
subOb.k = 9;
System.out.println("Contents of subOb: ");
subOb.showij();
subOb.showk();
System.out.println();
System.out.println("Sum of i, j and k in subOb:");
subOb.sum();
}
}
```

Output:

```
Contents of superOb: i
and j: 10 20 Contents of
subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Forms of Inheretence:

- Single Inheretence
- Hierarichal Inherintence
- Multiple Inherintence
- Multilevel Inherintence
- Hybrid Inherintence

Single Inherintence:

Derivation a subclass from only one super class is called Single Inherintence.

Hierarchical Inherintence:

Derivation of several classes from a single super class is called Hierarchical Inherintence:

Multilevel Inheritance:

Derivation of a classes from another derived classes called Multilevel Inheritance.

Multiple Inheritance:

Derivation of one class from two or more super classes is called Multiple Inheritance
But java does not support Multiple Inheritance directly. It can be implemented by using interface concept.

Hybrid Inheritance:

Derivation of a class involving more than one from on Inheritance is called Hydrid Inheritance

Defining a Subclass:

A subclass is defined as

Systax: class subclass-name extends superclass-name

{

Variable declaration;
Method declaration;

}

The keyword extends signifies that the properties of the super class name are extended to the subclass name. The subclass will now contain its own variables and methods as well as those of the super class. But it is not vice-versa.

Member access rules

- Even though a subclass includes all of the members of its super class, it cannot access those members who are declared as **Private** in super class.
- We can assign a reference of super class to the object of sub class. In that situation we can access only super class members but not sub class members. This concept is called as –Super class Reference, Sub class Object‖.

```
/* In a class hierarchy, private members remain private to their class.
This program contains an error and will not
compile.
*/
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
```

```
System.out.println("Total is " + subOb.total);
}
}
```

Super class variables can refer sub-class object

- To a reference variable of a super class can be assigned a reference to any subclass derived from that super class.
- When a reference to a subclass object is assigned to a super class reference variable, we will have to access only to those parts of the object defined by the super class
- It is bcz the super class has no knowledge about what a sub class adds to it.

Program

```
class RefDemo

{
public static void main(String args[])

{
BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
Box plainbox = new Box();
double vol;
vol = weightbox.volume();
System.out.println("Volume of weightbox is " + vol);
System.out.println("Weight of weightbox is " +
weightbox.weight);
System.out.println();
// assign BoxWeight reference to Box reference
plainbox = weightbox;
vol = plainbox.volume(); // OK, volume() defined in Box
System.out.println("Volume of plainbox is " + vol);
/* The following statement is invalid because plainbox
does not define a weight member. */
// System.out.println("Weight of plainbox is " + plainbox.weight);
}
}
```

Using super keyword

- When ever a sub class needs to refer to its immediate super class, it can do so by use of the key word **super**.
- Super has two general forms:
  - Calling super class constructor
  - Used to access a member of the super class that has been hidden by a member of a sub class

Using super to call super class constructor

- A sub class can call a constructor defined by its super class by use of the following form of super:
  - super (parameter-list);
  - Parameter list specifies parameters needed by the constructor in the super class.

Note: Super ( ) must always by the first statement executed inside a sub-class constuctor.

```
// A complete implementation of BoxWeight.
class Box {
private double width;
private double height;
private double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.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;
}
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
double weight; // weight of box
// construct clone of an object
BoxWeight(BoxWeight ob) { // pass object to constructor
super(ob);
weight = ob.weight;
```

```
}
// constructor when all parameters are specified
BoxWeight(double w, double h, double d, double m) {
super(w, h, d); // call superclass constructor
weight = m;
}
// default constructor
oxWeight() {
super();
weight = -1;
}
// constructor used when cube is created
BoxWeight(double len, double m) {
super(len);
weight = m;
}
}
class DemoSuper {
public static void main(String args[]) {
BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
BoxWeight mybox3 = new BoxWeight(); // default
BoxWeight mycube = new BoxWeight(3, 2);
BoxWeight myclone = new BoxWeight(mybox1);
double vol;
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
System.out.println("Weight of mybox1 is " + mybox1.weight);
System.out.println();
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
System.out.println("Weight of mybox2 is " + mybox2.weight);
System.out.println();
vol = mybox3.volume();
System.out.println("Volume of mybox3 is " + vol);
System.out.println("Weight of mybox3 is " + mybox3.weight);
System.out.println();
vol = myclone.volume();
System.out.println("Volume of myclone is " + vol);
System.out.println("Weight of myclone is " + myclone.weight);
System.out.println();
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
System.out.println("Weight of mycube is " + mycube.weight);
System.out.println();
}
}
```

Output:
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0

Calling members of super class using super

- The second form of super acts somewhat like **this** keyword, except that it always refers to the super class of the sub class in which it is used.
- The syntax is:
  - **Super.member ;**
  - Member can either be method or an instance variable

Program

```
// Using super to overcome name hiding.
class A {
int i;
}
// Create a subclass by extending class A.
class B extends A {
int i; // this i hides the i in A
B(int a, int b) {
super.i = a; // i in A
i = b; // i in B
}
void show() {
System.out.println("i in superclass: " + super.i);
System.out.println("i in subclass: " + i);
}
}
class UseSuper {
public static void main(String args[]) {
B subOb = new B(1, 2);
subOb.show();
}
}
```

Output:
i in superclass: 1
i in subclass: 2

When the constructor called:

Always the super class constructor will be executed first and sub class constructor will be executed last.

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
C c = new C();
}
}
```

Output:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Using Final keyword:

- We can use final key word in **three** ways:
  - Used to create equivalent of a named constant
    - Final datatype identifier = ……..;
  - Used to prevent inheritance
    - Final class ……..
  - Used to avoid overloading
    - Final return type …….

Using final to Prevent Overriding:

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}

class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Using final to Prevent Inheritance:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Polymorphism Method overriding:

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

- The version of the method defined in the super class is hidden.
- In this situation, first it checks the method is existed in super class are not. If it is existed then it executes the version of sub class otherwise it gives no such method found exception.

Note: Methods with different signatures overloading but not overriding.

```
// 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
}
}
```

Output:

```
k: 3
```

Dynamic method dispatch

- It is a mechanism by which a call to an overridden method is resolved at run time rather then compile time.
- It is important because this is how java implements runtime polymorphism.
- Before going to that we must know about super class reference sub class object.

```
// Dynamic Method Dispatch
class A {
void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}

class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
          allme(); // calls C's version of callme
}
}
```

Output:
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method

Abstract class:

- An abstract method is a method that is declared with only its signatures with out implementations.
- An abstract class is class that has at least one abstract method.

The syntax is:

```
Abstract class class-name
{
Variables
Abstract methods;
Concrete methods;
.
.
.}
```

- We can't declare any abstract constructor.
- Abstract class should not include any abstract static method.
- Abstract class can't be directly instantiated with the new operator.
- Any sub class of abstract class must be either implements all the abstract methods in the super class or declared it self as abstract.
- Abstract modifier referred as –*subclass responsibilities*‖ . because of no implementation of methods. Thus, a sub class must overridden them.

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
                    a
          llme()
          ;
          b.call
          metoo
          ();
}
}

// Using abstract methods and classes.
abstract class Figure {
double dim1;
double dim2;
Figure(double a, double b) {
dim1 = a;
```

```
dim2 = b;
}
// area is now an abstract method
abstract double area();
}
class Rectangle extends Figure {
Rectangle(double a, double b) {
super(a, b);
}
// override area for rectangle
double area() {
System.out.println("Inside Area for Rectangle.");
return dim1 * dim2;
}
}
class Triangle extends Figure {
Triangle(double a, double b) {
super(a, b);
}
// override area for right triangle
double area() {
System.out.println("Inside Area for Triangle.");
return dim1 * dim2 / 2;
}
}

class AbstractAreas

{
public static void main(String args[])
{
// Figure f = new Figure(10, 10); // illegal now
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref; // this is OK, no object is created
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
}
}
```

Packages and Interfaces : Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, differences between classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces.

.**Defining Package:**

- Generally, any java source file contains any (or all) of the following 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)

- Packages and Interfaces are two of the basic components of a java program.
- Packages are collection of related classes.
- Packages are containers for classes that are used to keep the class name compartmentalized.
- Packages are stored in an hierarchical manner and are explicitly imported into new class defination.
- Java packages are classified into two types:
- Java API package or pre-defined packages or built – in – packages .
- User – defined packages
- Java 2 API contains 60 java.* packages.
- For Example:
  - java.lang
  - Java.io
  - Java.awt
  - Java.util
  - Java.net
  - Javax.swing

- To create a package
- Just give package <<packagename>> as a first statement in java program.
- Any classes declared within that file will belong to the specified package.
- If we omit package statement, the classes are stored in the default package.
- Syntex:
- Package packagename

Syntax: **Package packagename.subpackage**

Access protection:

- Classes and packages both means of encapsulating and containing the name space and scope of variables and methods.
- Packages acts as a containers for classes and other sub – ordinate packages.

- Classes act as containers for data and code.
- Java address four categories of visibility for class members:
  - Sub – classes in the same package.
  - Non – sub class in the same package.
  - Sub – classes in the different package.
  - Classes that are neither in the same package nor subclasses.
- The 3 access specifiers private, public and protected provide a variety of ways to produce the many levels of access required by these categories.

Access specifier **Access Location**	**Private**	**No modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package sub class	No	Yes	Yes	Yes
Same package non – sub class	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non sub class	No	No	No	Yes

From the above table,

- Any thing declared public can be accessed from any where
- Any thing accessed private cannot be accessed from outside of its class
- In the default, it is visible to sub-class as well as to other classes in the same package
- Any thing declared as protected, this is allow an element to be seen outside your current package, but also allow to sub class in other packages access.

UNDERSTANDING CLASSPATH:

As just explained, packages are mirrored by directories. This raises an important question: How does the 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. Ultimately, the choice is yours.

Importing Packages:

There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, 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 a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

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.*;
```

Difference between classes and interfaces:

A class is a template for an abject.

(or)

A class is a way of binding variables and methods in a single unit. With the class it is possible to create object for that object. With the one class we can extend an another class.

A interface is collection of undefined method. Means all the methods are not contain any body. We have to provide the body for that interface. with the interface it is not possible to create object. For the declared interface we have to implement that interface.

Defining Interfaces:

- Interface is a collection of method declarations and constants that one or more classes of objects will use.

- We can implement multiple inheritance using interface.
- Because interface consists only signatures followed by semi colon and parameter list they are implicitly abstract.
- Variables can be declared and initialized inside interface they are implicitly final and static.
- An interface method can't be final or static or private or native or protected.
- An interface can be extended from another interface.
- Declaration of interface:

```
Access interface name
{
    Return type member-name1(parametelist);
    Return type member-name2(parametelist);
    .
    .
    .
   Type finalvariablename=initialization;
}
```

  - There will be no default implementation for methods specified in an interface.
  - Each class that include interface must implements all methods.
  - All the methods and variables are implicitly public if interface itself is declared as public.

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 includes the **implements** clause looks like this:

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.

Applying Interfaces:

To understand the power of interfaces, let's look at a more practical example. In earlier chapters you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be ‒growable.‖ The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to

the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to 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.

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
}
```

Variables in Interfaces:

When you 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.

```
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());
}
}
```

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. Following is an 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();
}
}
```

As an experiment you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

UNIT-III

EXCEPTION HANDLING AND MULTITHREADING

Introduction

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instruction.

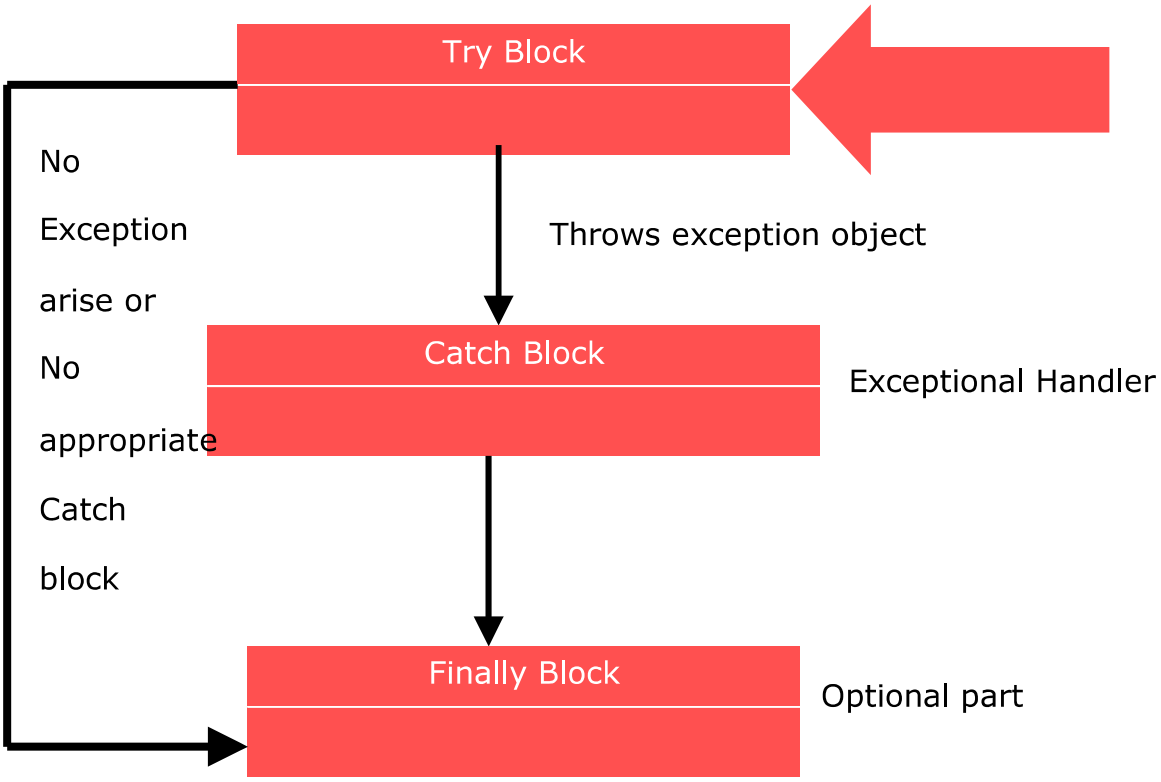
Or

- An abnormal condition that disrupts Normal program flow.
- There are many cases where abnormal conditions happen during program execution, such as
  - Trying to access an out - of – bounds array elements.
  - The file you try to open may not exist.
  - The file we want to load may be missing or in the wrong format.
  - The other end of your network connection may be non – existence.
- If these cases are not prevented or at least handled properly, either the program will be aborted abruptly, or the incorrect results or status will be produced.
- When an error occurs with in the java method, the method creates an exception object and hands it off to the runtime system.
- The exception object contains information about the exception including its type and the state of the program when the error occurred. The runtime system is then responsible for finding some code to handle the error.
- In java creating an exception object and handling it to the runtime system is called throwing an exception.
- Exception is an object that is describes an exceptional (i.e. error) condition that has occurred in a piece of code at run time.
- When a 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.
- System generated exceptions are automatically thrown by the Java runtime system

General form of 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
}
```



- By using exception to managing errors, Java programs have have the following advantage over traditional error management techniques:
  - Separating Error handling code from regular code.
  - Propagating error up the call stack.
  - Grouping error types and error differentiation.

For Example:

```
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. Here is the output generated when

this example is executed.
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**

Try and Catch Blocks

- If we don't want to prevent the program to terminate by the exception we have to trap the trap the exception using the try block. So we can place the statements that may causes an exception in the try block.

```
Try
{
}
```

- If an exception occurs with in the try block, the appropriate exception handler that is associated with the try block handles the exception immediately following the try block, include a catch clause specifies the exception type we wish to catch. A try block must have at least one catch block or finally that allows it immediately.

Catch block

- The catch block is used to process the exception raised. A try block can be one or more catch blocks can handle a try block.
- Catch handles are placed immediately after the try block.

```
Catch(exceptiontype e)
{
    //Error handle routine is placed here for handling exception
}
```

Program 1

```
Class trycatch
{
Public static void main(String args[])
{
Int[] no={1,2,3};
Try
{
System.out.println(no[3]);
}
Catch(ArrayIndexOutOfBoundsException e)
{
System.out.println(–Out of bonds‖);
}
System.out.println(–Quit‖);
}
}
```

Output

```
Out of the Range
Quit
```

Program 2

```
Class ArithExce
{
Public static void main(String args[])
{
Int a=10;
Int b=0;
Try
{
  a=a/b; System.out.println(–Won't
Print‖);
}
```

```
Catch(ArithmeticException  e)
{
System.out.println(–Division by Zero error‖);
System.out.println(–Change the b value‖);

}
System.out.println(–Quit‖);
}
}
```

Output

```
Division By zero error
Please change the B value
Quit
```

Note:

- A try ad its catch statement form a unit.
- We cannot use try block alone.
- The compiler does not allow any statement between try block and its associated catch block

Displaying description of an Exception

- 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.

```
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 Blocks

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]**.
Here is the 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.

Throw Keyword

- So far, we have only been catching exceptions that are thrown by the Java Run – Time systems. How ever, it is possible for our program to throw an exception explicitly, using the throw statement.
  - 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
  - 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

```
// 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. Here is the resulting 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 Keyword

- 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

Program

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

- Here is the output generated by running this example program:
- inside throwOne
- caught java.lang.IllegalAccessException

S.No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Finally block

- 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.

```
// 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**.*

- Here is the output generated by the preceding program:

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

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executedwhether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1. class FinalExample{
2. public static void main(String[] args){
3. final int x=100;
4. x=200;//Compile Time
Error 5. }}
```

Java finally example

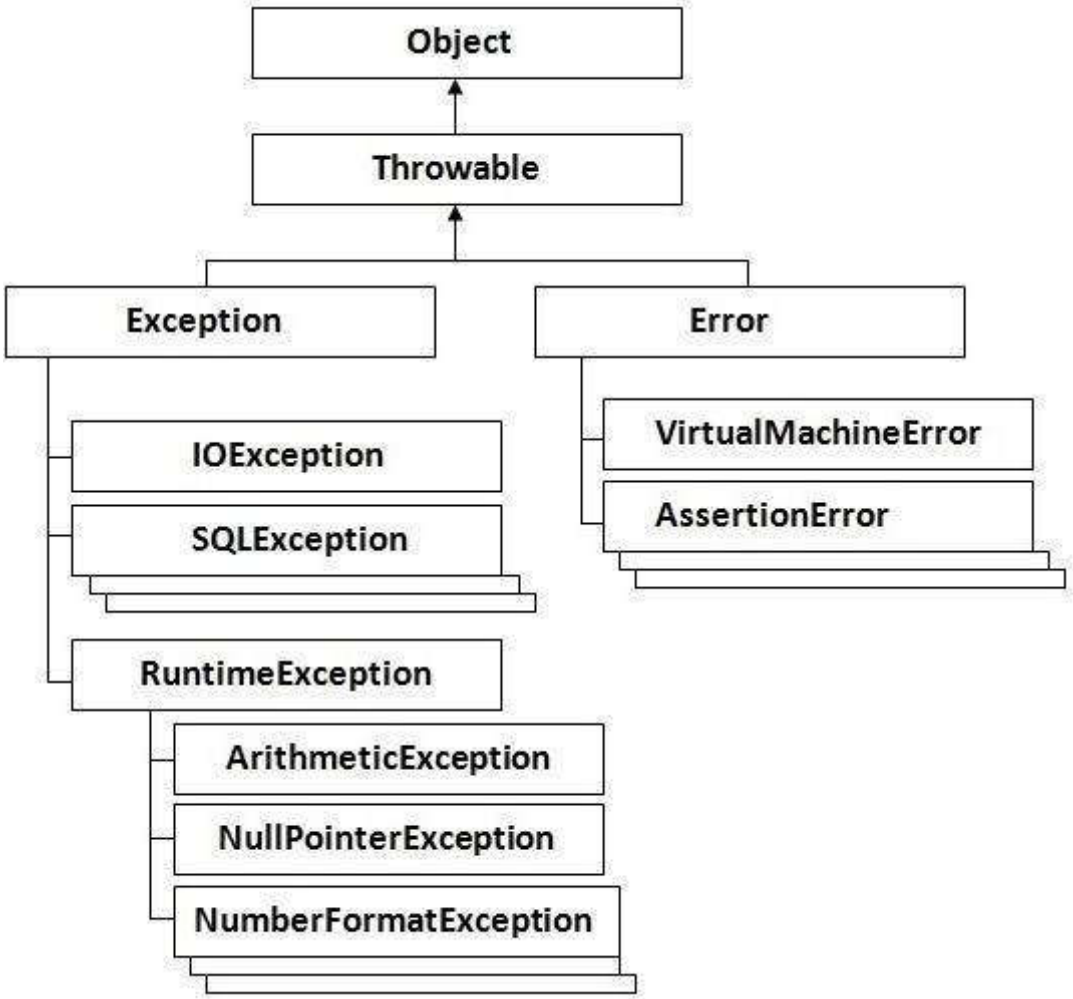
```
1. class FinallyExample{
2. public static void main(String[] args){
3. try{
4. int x=300;
5. }catch(Exception e){System.out.println(e);}
```

6. finally{System.out.println("finally block is executed");} 7. }}

Java finalize example

```
1. class FinalizeExample{
2. public void finalize(){System.out.println("finalize called");}
3. public static void main(String[] args){
4. FinalizeExample f1=new FinalizeExample();
5. FinalizeExample f2=new FinalizeExample();
6. f1=null;
7. f2=null;
8. System.gc()
; 9. }}
```

Hierarchy of Java Exception classes



Java 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 10-1. Table 10-2 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

List of Unchecked exceptions

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

Exception	Meaning
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

List of Checked exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does r implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

User defined exceptions

- We can create our own exception by extending exception class.
- The throw and throws keywords are used while implementing user defined exceptions

```
Class ownExcepion extends Exception
{
ownException(String msg)
{
Super(msg);
}
}

Class test
{
Public static void main(String args[])
Int mark=101;
Try
```

```
{
if(mark>100)
{
Throw new ownException(—Marks>100‖);
}
}
Catch(ownException e)
{
System.out.println (—Exception caughtr‖);
System.out.println.(—e.getMessage());
}
Finally
{
System.out.println(–End of prg‖);
}
}
}
```

Output:

Exception caught
Marks is > 100
End of program

Multi Threaded Programming

Introduction:

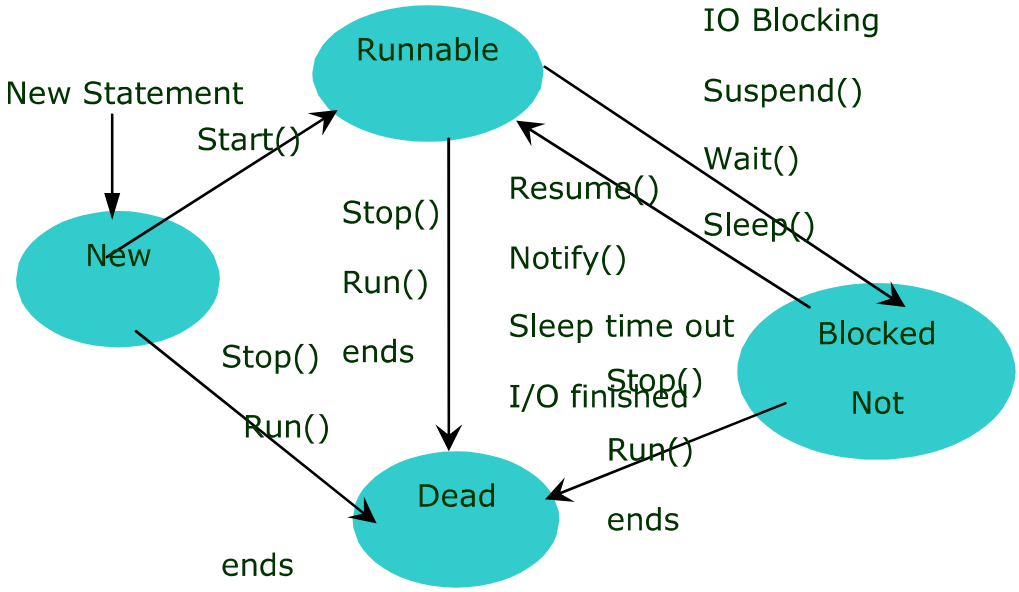
- Java provides a built – in support for multithreaded programming.
- A multithreaded program contains two o more parts that can run concurrently.
- Each part of such a program called thread.
- Each thread defines a separate path of execution.
- Thus multi thread is a specialized form of multi tasking.
- Multi tasking is supported by OS
- There are two distinct types of multi tasking
- Process based multi tasking
- Process is a program that is executing.
- In process based multi tasking, a program is the smallest unit of code that can be dispatched by the scheduler
- Process based multi tasking is a feature that allows computer to run two or more programs concurrently
- For example :
- This tasking enables us to run the Java compiler and text editor at the same time
- Thread based multi tasking
- Thread is a smallest unit of dispatchable code

- The single program can perform two or more tasks simultaneously.
- For example:
- A text editor can format text at the same time that is printing as long as these two actions are performed by two separate threads.
- Multitasking threads require less overhead than multitasking processes.

Thread Model

- 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.

- Thread States
  - 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
- Every thread after creation and before destruction can have any one of four states:
  - Newly created
  - Runnable state
  - Blocked
  - Dead



THREAD LIFE CYCLE

New State

- A thread enters the newly created by using a new operator.
- It is new state or born state immediately after creation. i.e. when a constructor is called the Thread is created but is not yet to run() method will not begin until it start() method is called.
- After the start() method is called, the thread will go to the next state, Runnable state.
- Note : in the above cycle stop(), resume() and suspand are deprecated methods. Java 2 strongly discourage their usage in the program

Runnable state

- Once we invoke the start() method, the thread is runnable.
- It is divided into two states:
  - The running state
    - When the thread is running state, it assigned by CPU cycles and is actually running.
  - The Queued state.
    - When the thread is in Queued state, it is waiting in the Queue and competing for its turn to spend CPU cycles
    - It is controlled by Virtual Machine Controller.
    - When we use yield() method it makes sure other threads of the same priority have chance to run.
    - This method cause voluntary move itself to the queued state from the running state.

Blocked state

- The blocked state is entered when one of the following events occurs:
  - The thread itself or another thread calls the suspend() method (it is deprecated)
  - The thread calls an object's wait() method
  - The thread itself calls the sleep() method.
  - The thread is waiting for some I/O operations to complete.
  - The thread will join() another thread.

Dead state

- A thread is dead for any one of the following reasons:
  - It dies a natural death because the un method exists normally.
  - It dies abruptly because an uncaught exception terminates the run method.
  - In particular stop() is used to kill the thread. This is depricated.
  - To find whether thread is alive i.e. currently running or blocked
    - Use isAlive() method
      - If it returns true the thread is alive

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 preempted 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.

Synchronization

- Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you 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, you 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

- After you divide your 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.

Thread class and Runnable interface

The Thread Class and the Runnable InterfaceJava's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you 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
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Main method

- 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

- 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.

```
// 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");
}
}
}
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

How to create a thread

- In the most general sense, you 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 thread class

- 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 you 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( )

Extending thread class

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.");
}
}
```

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.

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.

Daemon Threads

A ―daemon‖ thread is one that is supposed to provide a general service in the background as long as the program is running, but is not part of the essence of the program. Thus when all of the non-daemon threads complete, the program is terminated. you an find out if a thread is a daemon by calling isDaemon(), and you can turn the ―daemonhood‖ of a thread on and off with setDaemon().if a thread is a daemon, then any threads it creates will automatically be daemons.

INTER-THREAD COMMUNICATION IN JAVA

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final voidwait(long timeout)throws InterruptedException	waits for the specified amount of time.

2) notify() method

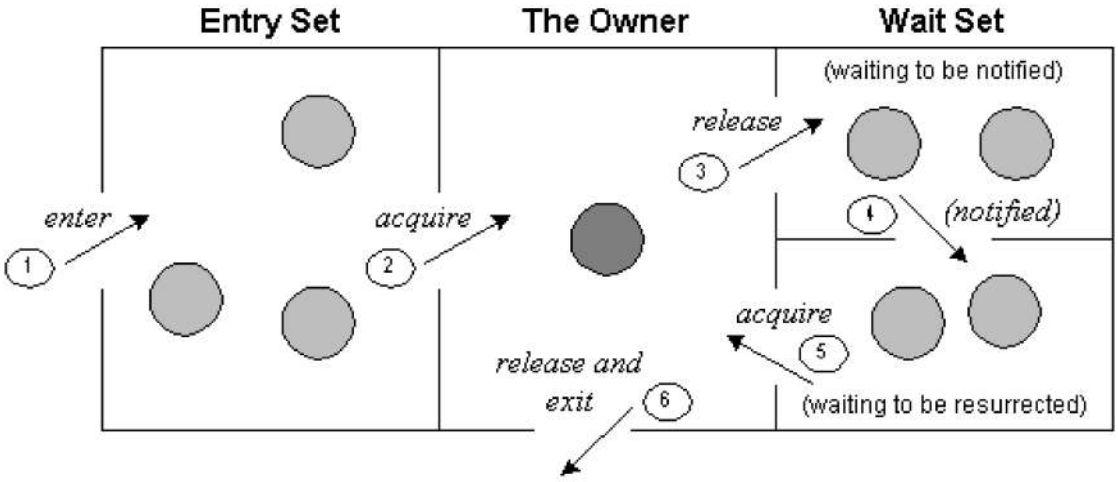
Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax: **public final void notify()**

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor.
Syntax: **public final void notifyAll()**

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

Let's see the important differences between wait and sleep methods.

wait()	sleep()
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
1. class Customer{
2. int amount=10000;
3.
4. synchronized void withdraw(int amount){
5. System.out.println("going to withdraw...");
6.
7. if(this.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. try{wait();}catch(Exception e){}
10. }
11. this.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
14.
15. synchronized void deposit(int amount){
16. System.out.println("going to deposit...");
```

```
17. this.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
22.
23. class Test{
24. public static void main(String args[]){
25. final Customer c=new Customer();
26. new Thread(){
27. public void run(){c.withdraw(15000);}
28. }.start();
29. new Thread(){
30. public void run(){c.deposit(10000);}
31. }.start();
32.
33. }}
```

```
Output: going to withdraw...
        Less balance; waiting for deposit...
        going to deposit...
        deposit completed...
        withdraw completed
```

INTERRUPTING A THREAD:

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 Methods Provided By The Thread Class For Interrupting A Thread

public void interrupt()

public static boolean interrupted()

public boolean isInterrupted()

Example of interrupting a thread that stops working

In this example, after interrupting the thread, we are propagating it, so it will stop working. If we don't want to stop the thread, we can handle it where sleep() or wait() method is invoked. Let's first see the example where we are propagating the exception.

```
1. class TestInterruptingThread1 extends Thread{
2. public void run(){
3. try{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }catch(InterruptedException e){
7. throw new RuntimeException("Thread interrupted..."+e);
8. }
9.
10.
}
11.
12. public static void main(String args[]){
13. TestInterruptingThread1 t1=new TestInterruptingThread1();
14. t1.start();
15. try{
16. t1.interrupt();
17. }catch(Exception e){System.out.println("Exception handled
"+e);} 18.
19. }
20. }
```

```
Output:Exception in thread-0
        java.lang.RuntimeException:  Thread  interrupted...
        java.lang.InterruptedException: sleep interrupted at A.run(A.java:7)
```

Example of interrupting a thread that doesn't stop working

In this example, after interrupting the thread, we handle the exception, so it will break out the sleeping but will not stop working.

```
1. class TestInterruptingThread2 extends Thread{
2. public void run(){
3. try{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }catch(InterruptedException e){
7. System.out.println("Exception handled "+e);
8. }
9. System.out.println("thread is running...");
10.
}
11.
12. public static void main(String args[]){
13. TestInterruptingThread2 t1=new TestInterruptingThread2();
14. t1.start();
15.
```

```
16. t1.interrupt();
17.
18. }
19. }
```

```
Output:Exception  handled java.lang.InterruptedException: sleep interrupted thread
            is running...
```

Example of interrupting thread that behaves normally

If thread is not in sleeping or waiting state, calling the interrupt() method sets the interrupted flag to true that can be used to stop the thread by the java programmer later.

```
1. class TestInterruptingThread3 extends Thread{
2.
3.   public void run(){
4. for(int
i=1;i<=5;i++)
5. System.out.println(i);
6. }
7.
8.   public static void main(String args[]){
9.   TestInterruptingThread3 t1=new TestInterruptingThread3();
10. t1.start();
11.
12. t1.interrupt();
13.
14. }
15. }
```

```
Output:1
            2
            3
            4
            5
```

What About Isinterrupted And Interrupted Method?

The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true.

```
1. public class TestInterruptingThread4 extends Thread{
2.
3.   public void run(){
4. for(int i=1;i<=2;i++){
5.   if(Thread.interrupted()){
6.   System.out.println("code for interrupted thread");
7.  }
8.   else{
9.   System.out.println("code for normal thread");
10.
}
11.
12. }//end of for loop
13.
}
14.
15. public static void main(String
args[]){ 16.
17.  TestInterruptingThread4 t1=new TestInterruptingThread4();
18.  TestInterruptingThread4 t2=new TestInterruptingThread4();
19.
20.  t1.start();
21.  t1.interrupt();
22.
23. t2.start();
24.
25. }
26. }
```

Output:Code for interrupted thread code for
normal thread code for normal
thread code for normal thread

UNIT- IV

FILES AND CONNECTING TO DATABASE

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;

public class CopyFile {
  public static void main(String args[]) throws IOException
  {
    FileInputStream in = null;
    FileOutputStream out = null;

    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
```

```
         out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , **FileReader** and **FileWriter.**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {
  public static void main(String args[]) throws IOException
  {
    FileReader in = null;
    FileWriter out = null;

    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
```

```
        if (in != null) {
          in.close();
        }
        if (out != null) {
          out.close();
        }
      }
    }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware if C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader** to read standard input stream until the user types a "q":

```
import java.io.*;

public class ReadConsole {
  public static void main(String args[]) throws IOException
  {
    InputStreamReader cin = null;
```

```
    try {
      cin = new InputStreamReader(System.in);
      System.out.println("Enter characters, 'q' to quit.");
      char c;
      do {
        c = (char) cin.read();
        System.out.print(c);
      } while(c != 'q');
    }finally {
      if (cin != null) {
        cin.close();
      }
    }
  }
}
```

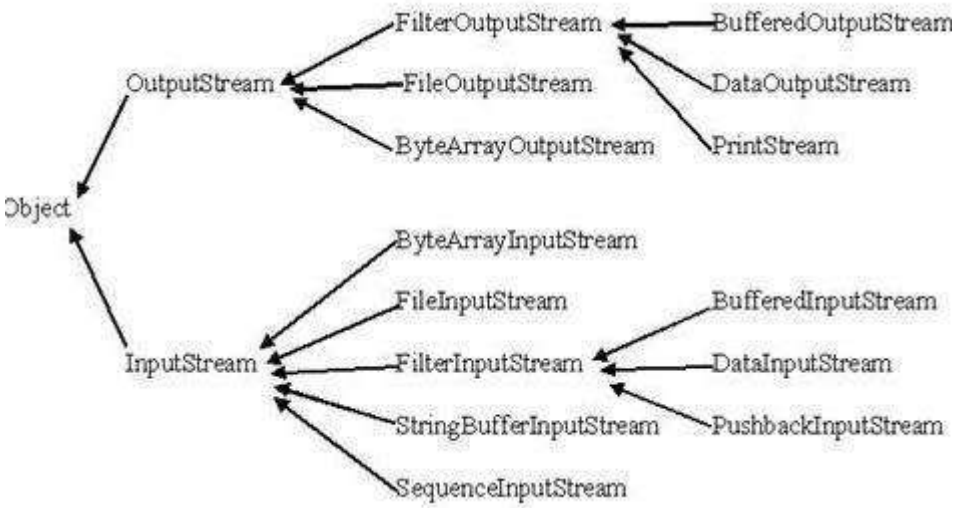
Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

InputStream f = new FileInputStream("C:/java/hello");

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close() throwsIOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**Protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.

3	**Public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**Public int read(byte[] r)throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links:

- ByteArrayInputStream
- DataInputStream

FileOutputStream:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**Public void write(int w)throws IOException{}**

This methods writes the specified byte to the output stream.

4 **Public void write(byte[] w)**
Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

  public static void main(String args[]){

  try{
    byte bWrite [] = {11,21,3,40,5};
    OutputStream os = new FileOutputStream("test.txt");
    for(int x=0; x < bWrite.length ; x++){
      os.write( bWrite[x] ); // writes the bytes
    }
    os.close();

    InputStream is = new FileInputStream("test.txt");
    int size = is.available();

    for(int i=0; i< size; i++){
      System.out.print((char)is.read() + " ");
    }
    is.close();
  }catch(IOException e){
    System.out.print("Exception");
  }
  }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories:

There are two useful **File** utility methods, which can be used to create directories:

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;

public class CreateDir {
  public static void main(String args[]) {
    String dirname = "/tmp/user/java/bin";
    File d = new File(dirname);
    // Create directory now.
    d.mkdirs();
  }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories:

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;

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

    File file = null;
    String[] paths;

    try{
      // create new file object
      file = new File("/tmp");

      // array of files and directory
      paths = file.list();

      // for each name in the path array
      for(String path:paths)
      {
        // prints filename and directory name
        System.out.println(path);
      }
    }catch(Exception e){
      // if any error occurs
      e.printStackTrace();
    }
  }
}
```

This would produce following result based on the directories and files available in your **/tmp** directory:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

JAVA DATABASE CONNECTIVITY

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks commonly associated with database usage:

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing that SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as:

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs)

All of these different executables are able to use a JDBC driver to access a database and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

JDBC Architecture:

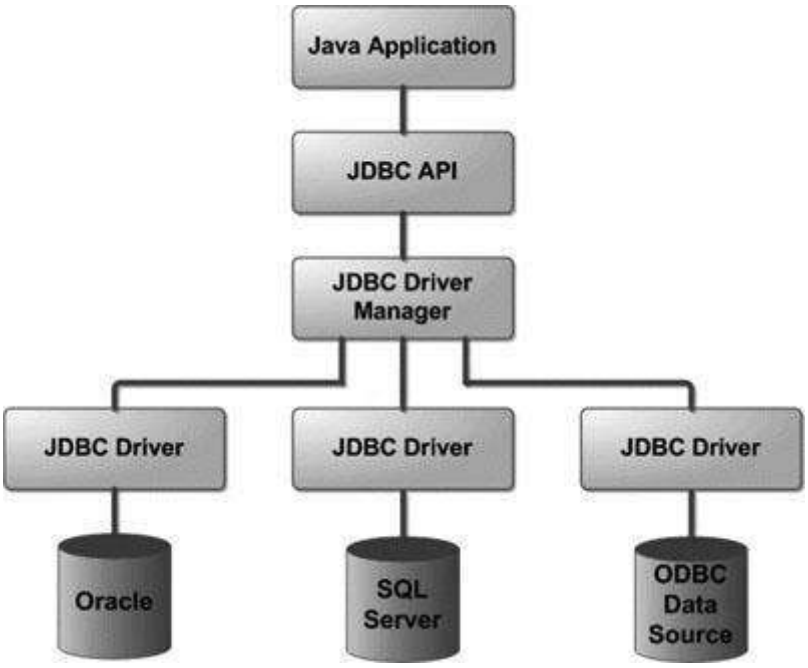
The JDBC API supports both two-tier and three-tier processing models for database access but in general JDBC Architecture consists of two layers:

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application:



Common JDBC Components:

The JDBC API provides the following interfaces and classes:

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication subprotocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects
- **Connection :** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement :** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

Creating JDBC Application:

There are following six steps involved in building a JDBC application:

- **Import the packages .** Requires that you include the packages containing the JDBC classes needed for database programming. Most often, using *import java.sql.** will suffice.
- **Register the JDBC driver .** Requires that you initialize a driver so you can open a communications channel with the database.
- **Open a connection .** Requires using the *DriverManager.getConnection()* method to create a Connection object, which represents a physical connection with the database.
- **Execute a query .** Requires using an object of type Statement for building and submitting an SQL statement to the database.
- **Extract data from result set .** Requires that you use the appropriate *ResultSet.getXXX()* method to retrieve the data from the result set.
- **Clean up the environment .** Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

Creating JDBC Application:

- There are six steps involved in building a JDBC application which I'm going to brief in this tutorial:
- **Import the packages:**
- This requires that you include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql.* will suffice as follows:
- //STEP 1. Import required packages
- import java.sql.*;
- **Register the JDBC driver:**
- This requires that you initialize a driver so you can open a communications channel with the database. Following is the code snippet to achieve this:
- //STEP 2: Register JDBC driver
- Class.forName("com.mysql.jdbc.Driver");
- **Open a connection:**
- This requires using the DriverManager.getConnection() method to create a Connection object, which represents a physical connection with the database as follows:
- //STEP 3: Open a connection
- // Database credentials
- static final String USER = "username";
- static final String PASS = "password";
- System.out.println("Connecting to database...");
- conn = DriverManager.getConnection(DB_URL,USER,PASS);
- **Execute a query:**
- This requires using an object of type Statement or PreparedStatement for building and submitting an SQL statement to the database as follows:
- //STEP 4: Execute a query
- System.out.println("Creating statement...");
- stmt = conn.createStatement();
- String sql;

sql = "SELECT id, first, last, age FROM Employees";
ResultSet rs = stmt.executeQuery(sql);
If there is an SQL UPDATE,INSERT or DELETE statement required, then following code snippet would be required:
//STEP 4: Execute a query
System.out.println("Creating statement...");
stmt = conn.createStatement();
String sql;
sql = "DELETE FROM Employees";
ResultSet rs = stmt.executeUpdate(sql);
Extract data from result set:
This step is required in case you are fetching data from the database. You can use the appropriate ResultSet.getXXX() method to retrieve the data from the result set as follows:
//STEP 5: Extract data from result set
while(rs.next()){
    //Retrieve by column name
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");
•
    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
• }
Clean up the environment:
You should explicitly close all database resources versus relying on the JVM's garbage collection as follows:
//STEP 6: Clean-up environment
rs.close();
stmt.close();
conn.close();

First JDBC Program:

Based on the above steps, we can have following consolidated sample code which we can use as a template while writing our JDBC code:
This sample code has been written based on the environment and database setup done in Environment chapter.
//STEP 1. Import required packages
import java.sql.*;
•
public class FirstExample {

```
// JDBC driver name and database URL
static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
static final String DB_URL = "jdbc:mysql://localhost/EMP";

// Database credentials
static final String USER = "username";
static final String PASS = "password";

public static void main(String[] args) {
Connection conn = null;
Statement stmt = null;
try{
  //STEP 2: Register JDBC driver
  Class.forName("com.mysql.jdbc.Driver");

  //STEP 3: Open a connection
  System.out.println("Connecting to database...");
  conn = DriverManager.getConnection(DB_URL,USER,PASS);

  //STEP 4: Execute a query
  System.out.println("Creating statement...");
  stmt = conn.createStatement();
  String sql;
  sql = "SELECT id, first, last, age FROM Employees";
  ResultSet rs = stmt.executeQuery(sql);

  //STEP 5: Extract data from result set
  while(rs.next()){
    //Retrieve by column name
    int id = rs.getInt("id");
    int age = rs.getInt("age");
    String first = rs.getString("first");
    String last = rs.getString("last");

    //Display values
    System.out.print("ID: " + id);
    System.out.print(", Age: " + age);
    System.out.print(", First: " + first);
    System.out.println(", Last: " + last);
  }
  //STEP 6: Clean-up environment
  rs.close();
  stmt.close();
  conn.close();
}catch(SQLException se){
  //Handle errors for JDBC
```

```
      se.printStackTrace();
    }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
  }finally{
      //finally block used to close resources
      try{
        if(stmt!=null)
          stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
        if(conn!=null)
          conn.close();
      }catch(SQLException se){
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");
  }//end main
  }//end FirstExample
```

- Now let us compile above example as follows:

```
C:\>javac FirstExample.java
C:\>
```

- When you run **FirstExample**, it produces following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

- **SQLException Methods:**
- A SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.
- The passed SQLException object has the following methods available for retrieving additional information about the exception:

Method	**Description**
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this

	method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.
getNextException( )	Gets the next Exception object in the exception chain.
printStackTrace( )	Prints the current exception, or throwable, and its backtrace to a standard error stream.
printStackTrace(PrintStream s)	Prints this throwable and its backtrace to the print stream you specify.
printStackTrace(PrintWriter w)	Prints this throwable and its backtrace to the print writer you specify.

- By utilizing the information available from the Exception object, you can catch an exception and continue your program appropriately. Here is the general form of a try block:
- try {
- // Your risky code goes between these curly braces!!!
- }
- catch(Exception ex) {
- // Your exception handling code goes between these
- // curly braces, similar to the exception clause
- // in a PL/SQL block.
- }
- finally {
- // Your must-always-be-executed code goes between these
- // curly braces. Like closing database connection.
- }
- **JDBC - Data Types:**
- The following table summarizes the default JDBC data type that the Java data type is converted to when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

SQL	**JDBC/Java**	**setXXX**	**updateXXX**
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	Boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	Byte	setByte	updateByte
SMALLINT	Short	setShort	updateShort
INTEGER	Int	setInt	updateInt
BIGINT	Long	setLong	updateLong
REAL	Float	setFloat	updateFloat
FLOAT	Float	setFloat	updateFloat

DOUBLE	Double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

- JDBC 3.0 has enhanced support for BLOB, CLOB, ARRAY, and REF data types. The ResultSet object now has updateBLOB(), updateCLOB(), updateArray(), and updateRef() methods that enable you to directly manipulate the respective data on the server.
- The setXXX() and updateXXX() methods enable you to convert specific Java types to specific JDBC data types. The methods, setObject() and updateObject(), enable you to map almost any Java type to a JDBC data type.
- ResultSet object provides corresponding getXXX() method for each data type to retrieve column value. Each method can be used with column name or by its ordinal position.

SQL	**JDBC/Java**	**setXXX**	**getXXX**
VARCHAR	java.lang.String	setString	getString
CHAR	java.lang.String	setString	getString
LONGVARCHAR	java.lang.String	setString	getString
BIT	Boolean	setBoolean	getBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	getBigDecimal
TINYINT	Byte	setByte	getByte
SMALLINT	Short	setShort	getShort
INTEGER	Int	setInt	getInt
BIGINT	Long	setLong	getLong
REAL	Float	setFloat	getFloat
FLOAT	Float	setFloat	getFloat
DOUBLE	Double	setDouble	getDouble
VARBINARY	byte[ ]	setBytes	getBytes
BINARY	byte[ ]	setBytes	getBytes
DATE	java.sql.Date	setDate	getDate
TIME	java.sql.Time	setTime	getTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	getTimestamp
CLOB	java.sql.Clob	setClob	getClob

BLOB	java.sql.Blob	setBlob	getBlob
ARRAY	java.sql.Array	setARRAY	getARRAY
REF	java.sql.Ref	SetRef	getRef
STRUCT	java.sql.Struct	SetStruct	getStruct

Sample Code:

This sample example can serve as a **template** when you need to create your own JDBC application in the future.

This sample code has been written based on the environment and database setup done in previous chapter.

Copy and past following example in FirstExample.java, compile and run as follows:

```
//STEP 1. Import required packages
import java.sql.*;

public class FirstExample {
  // JDBC driver name and database URL
  static final String JDBC_DRIVER = "com.mysql.jdbc.Driver";
  static final String DB_URL = "jdbc:mysql://localhost/EMP";

  // Database credentials
  static final String USER = "username";
  static final String PASS = "password";

  public static void main(String[] args) {
  Connection conn = null;
  Statement stmt = null;
  try{
    //STEP 2: Register JDBC driver
    Class.forName("com.mysql.jdbc.Driver");

    //STEP 3: Open a connection
    System.out.println("Connecting to database...");
    conn = DriverManager.getConnection(DB_URL,USER,PASS);

    //STEP 4: Execute a query
    System.out.println("Creating statement...");
    stmt = conn.createStatement();
    String sql;
    sql = "SELECT id, first, last, age FROM Employees";
```

```
      ResultSet rs = stmt.executeQuery(sql);

      //STEP 5: Extract data from result set
      while(rs.next()){
        //Retrieve by column name
        int id = rs.getInt("id");
        int age = rs.getInt("age");
        String first = rs.getString("first");
        String last = rs.getString("last");

        //Display values
        System.out.print("ID: " + id);
        System.out.print(", Age: " + age);
        System.out.print(", First: " + first);
        System.out.println(", Last: " + last);
      }
      //STEP 6: Clean-up environment
      rs.close();
      stmt.close();
      conn.close();
    }catch(SQLException se){
      //Handle errors for JDBC
      se.printStackTrace();
    }catch(Exception e){
      //Handle errors for Class.forName
      e.printStackTrace();
    }finally{
      //finally block used to close resources
      try{
        if(stmt!=null)
          stmt.close();
      }catch(SQLException se2){
      }// nothing we can do
      try{
        if(conn!=null)
          conn.close();
      }catch(SQLException se){
        se.printStackTrace();
      }//end finally try
    }//end try
    System.out.println("Goodbye!");
}//end main
}//end FirstExample
```

Now let us compile above example as follows:

```
C:\>javac FirstExample.java
C:\>
```

When you run **FirstExample**, it produces following result:

```
C:\>java FirstExample
Connecting to database...
Creating statement...
ID: 100, Age: 18, First: Zara, Last: Ali
ID: 101, Age: 25, First: Mahnaz, Last: Fatma
ID: 102, Age: 30, First: Zaid, Last: Khan
ID: 103, Age: 28, First: Sumit, Last: Mittal
C:\>
```

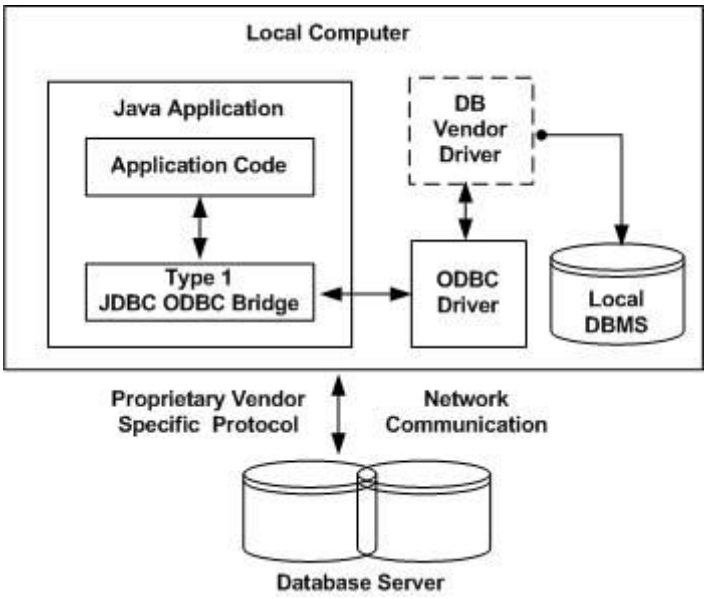
JDBC Drivers Types:

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below:

Type 1: JDBC-ODBC Bridge Driver:

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC requires configuring on your system a Data Source Name (DSN) that represents the target database.

When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.

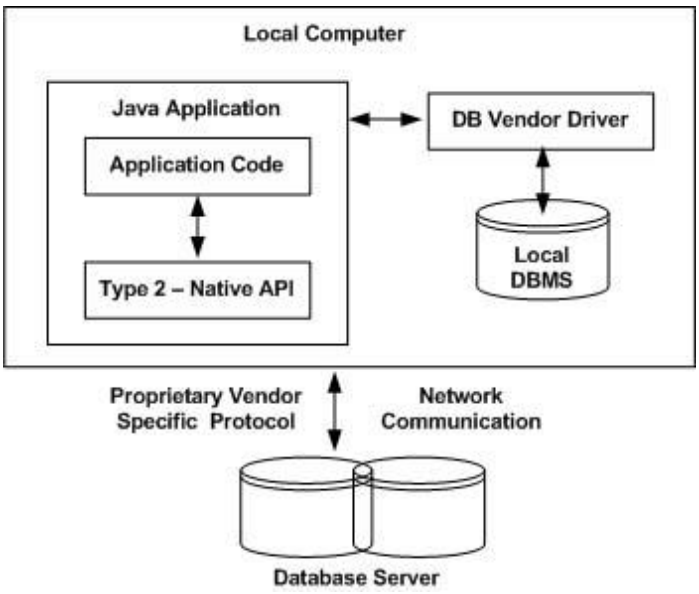


The JDBC-ODBC bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API:

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls which are unique to the database. These drivers typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge, the vendor-specific driver must be installed on each client machine.

If we change the Database we have to change the native API as it is specific to a database and they are mostly obsolete now but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

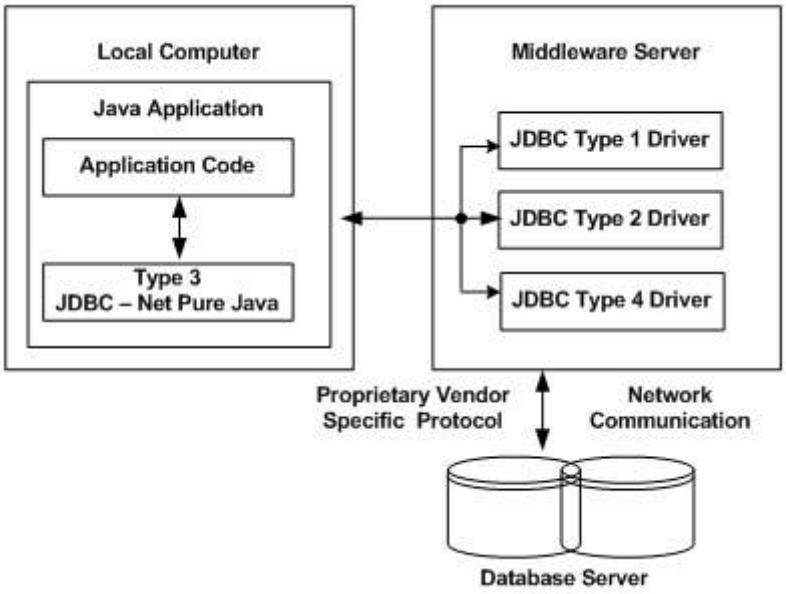


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java:

In a Type 3 driver, a three-tier approach is used to accessing databases. The JDBC clients use standard network sockets to communicate with an middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



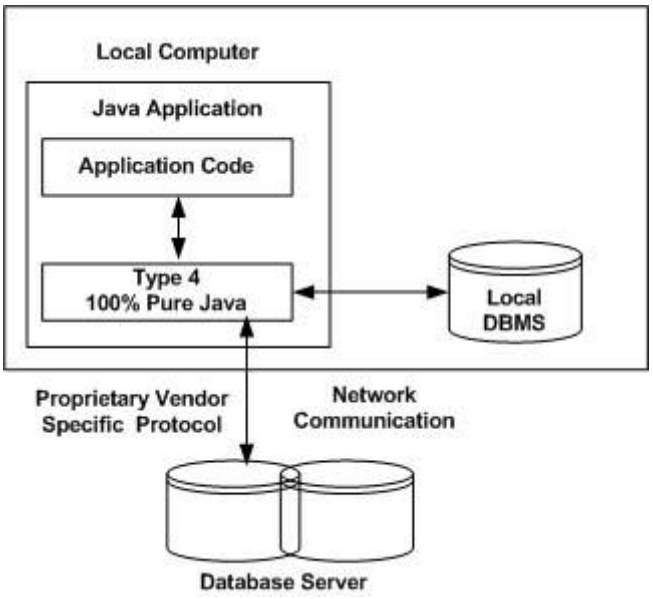
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% pure Java:

In a Type 4 driver, a pure Java-based driver that communicates directly with vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations where a type 3 or type 4 driver is not available yet for your database.

The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

UNIT-V

GUI PROGRAMMING AND APPLETS

INTRODUCTION OF SWING

The Swing-related classes are contained in **javax.swing** and its subpackages, such as **javax.swing.tree**. Many other Swing-related classes and interfaces exist that are not examined in this chapter.
The remainder of this chapter examines various Swing components and illustrates them through sample applets.

JApplet

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that useSwing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is notfound in **Applet**. For example, **JApplet** supports various ―panes,‖ such as the content pane, the glass pane, and the root pane. For the examples in this chapter, we will not be using most of **JApplet**'s enhanced features. However, 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*)

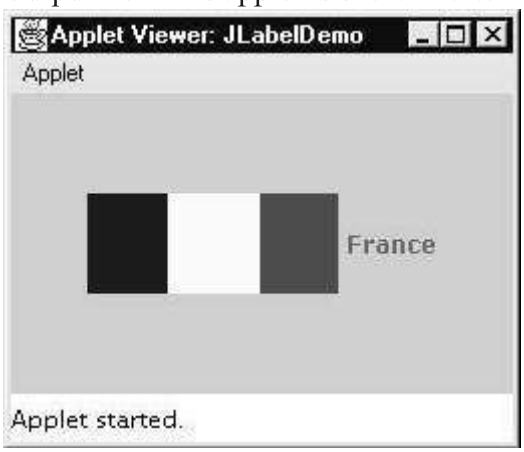
Here, *s* and *i* are the text and icon used for the label. The *align* argument is either **LEFT**,**RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

The icon and text associated with the label can be read and written by the
following methods:
Icon getIcon( )
String getText( )
void setIcon(Icon *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 bothan 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 from this applet is shown here:



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.
The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

Output from this applet is shown here:

Buttons

Swing buttons provide features that are not found in the **Button** class defined by theAWT. 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.
Each is examined next.

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 checkboxes 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 itslayout 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.

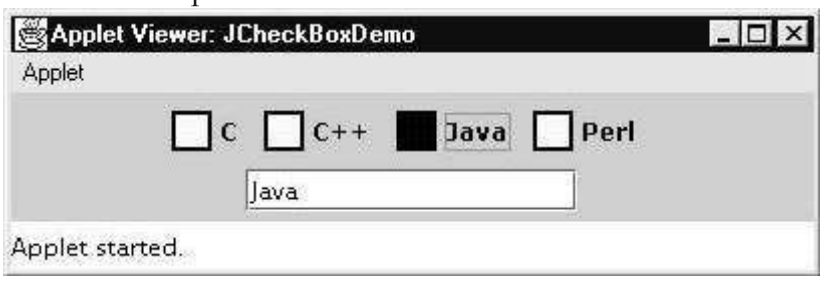
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo extends JApplet
implements ItemListener {
JTextField jtf;
public void init() {




// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("C++", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
```

```
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

Here is the output:



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.

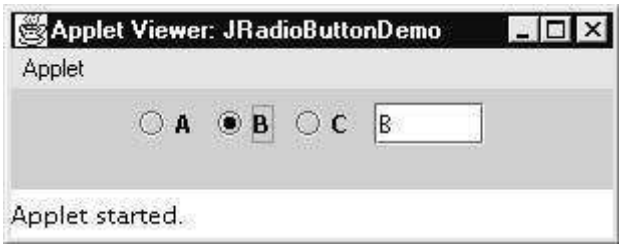
Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is

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.

Radio button presses generate action events that are handled by **actionPerformed( )**.
The **getActionCommand( )** method gets the text that is associated with a radio button and uses it to set the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener {
JTextField tf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
```

}
}
Output from this applet is shown here:



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.

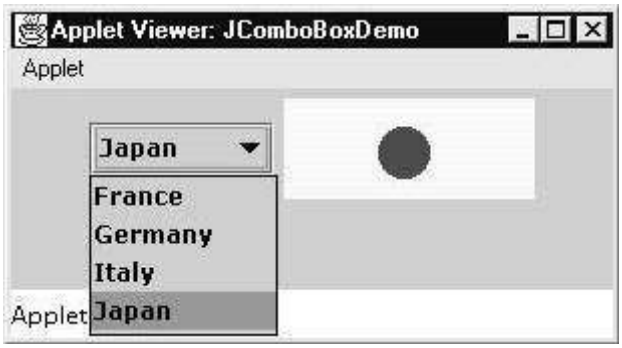
Items are added to the list of choices via the **addItem( )** method, whose signature is shown here:
void addItem(Object *obj*)
Here, *obj* is the object to be added to the combo box.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for "France", "Germany", "Italy", and "Japan".
When a country is selected, the label is updated to display the flag for that country.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener {
JLabel jl;
```

```
ImageIcon france, germany, italy, japan;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it
// to the panel
JComboBox jc = new JComboBox();
jc.addItem("France");
jc.addItem("Germany");
jc.addItem("Italy");
jc.addItem("Japan");
jc.addItemListener(this);
contentPane.add(jc);
// Create label
jl = new JLabel(new ImageIcon("france.gif"));
contentPane.add(jl);
}
public void itemStateChanged(ItemEvent ie) {
String s = (String)ie.getItem();
jl.setIcon(new ImageIcon(s + ".gif"));
}
}
```

Output from this applet is shown here:



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.

Scroll Panes

A *scroll pane* is a component that presents a rectangular area in which a componentmay 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*)

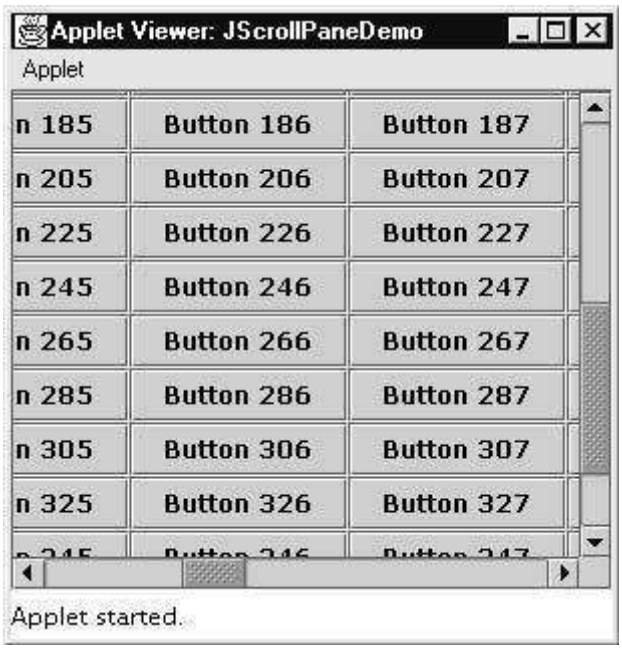
Here, *comp* is the component to be added to the scroll pane. *vsb* and *hsb* are **int**constants that define when vertical and horizontal scroll bars for this scroll pane areshown. These constants are defined by the **ScrollPaneConstants** interface. Some
examples of these constants are described as follows:
Constant Description
HORIZONTAL_SCROLLBAR_ALWAYS Always provide horizontal scroll bar
HORIZONTAL_SCROLLBAR_AS_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.

The following example illustrates a scroll pane. First, the content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. Next, a **JPanel** object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. You can use the scroll bars to scroll the buttons into view.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet {
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
// Add 400 buttons to a panel
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++) {
for(int j = 0; j < 20; j++) {
jp.add(new JButton("Button " + b));
++b;
}
}
// Add panel to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Output from this applet is shown here:



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.

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.

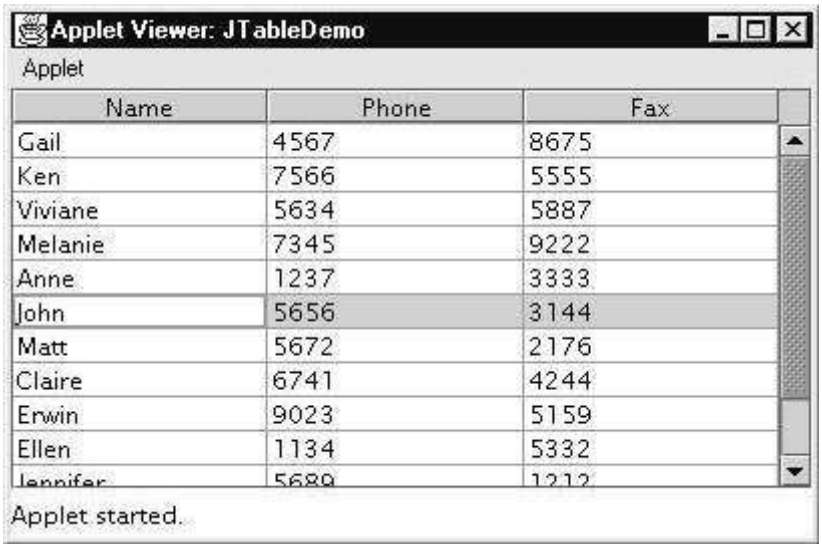
The following example illustrates how to create and use a table. The content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager.A one-dimensional array of strings is created for the column headings. This table has three columns. A two-dimensional array of strings is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
</applet>
*/
public class JTableDemo extends JApplet {

public void init() {
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Phone", "Fax" };
// Initialize data
final Object[][] data = {
{ "Gail", "4567", "8675" },
{ "Ken", "7566", "5555" },
{ "Viviane", "5634", "5887" },
{ "Melanie", "7345", "9222" },
{ "Anne", "1237", "3333" },
{ "John", "5656", "3144" },
```

```
{ "Matt", "5672", "2176" },
{ "Claire", "6741", "4244" },
{ "Erwin", "9023", "5159" },
{ "Ellen", "1134", "5332" },
{ "Jennifer", "5689", "1212" },
{ "Ed", "9030", "1313" },
{ "Helen", "6751", "1415" }
};
// Create the table
JTable table = new JTable(data, colHeads);
// Add table to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Here is the output:



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

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. Many other user operations could also be cited as examples. 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 add*Type*Listener(*Type*Listener *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 add*Type*Listener(*Type*Listener *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 remove*Type*Listener(*Type*Listener *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 Classes

The classes that represent events are at the core of Java's event handling mechanism.Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.
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( )

As expected, **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( )

Additional details about **AWTEvent** are provided at the end of Chapter 22. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

■ **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. Table 20-1 enumerates the most important of these event classes and provides a brief description of when they are generated. The most commonly used constructors and methods in each class are described in the following sections.

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.

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 conceptis quite simple: a *source* generates an event and sends it to one or more *listeners*. In thisscheme, 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 applicationlogic 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.

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

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.

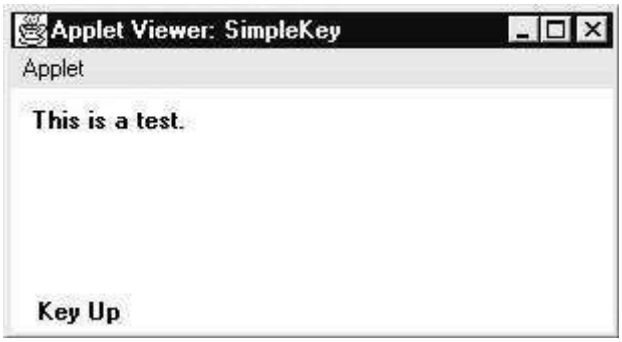
Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated bythe 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.

```
// 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:



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. Table 20-4 lists the commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

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.

Adapter Class	**Listener Interface**
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	
MouseMotionListener	
WindowAdapter	WindowListener

```
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");
}
}
```

Inner 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.");
}
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates howan anonymous inner class can facilitate the writing of event handlers. 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. As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

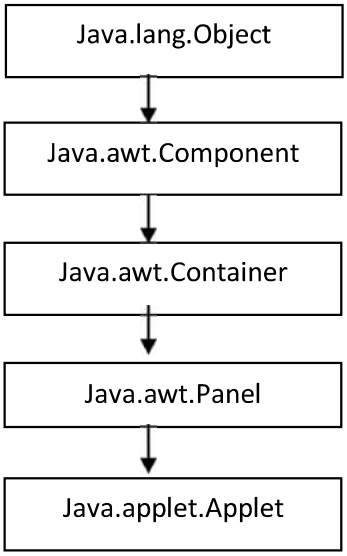
Applets

- Java programs are classified into two ways
  - Applications program
    - Application programs are those programs which are normally created, compiled and executed as similar to the other languages.
    - Each application program contains one main( ) method
    - Programs are created in a local computer.
    - Programs are compiled with javac compiler and executed in java interpreter.

  - Applet program
    - An applet is a special program that we can embedded in a web page such that the applet gains control over a certain part of the displayed page.
    - It is differ from application program
    - Applets are created from classes
    - An applet do not have main as an entry. Instead have several methods to control specific aspects of applet execution.

Class Hierarchy of Applet

- Every applet that we are creating must be a sub class of Applet.
- This Applet is extended from Panel.
- This Panel is extended from Container.
- The Container class extends Component class.
- Component class is extending from Object class which is parent of all Java API classes



About Java.awt.Component

- java.lang.Object
  - java.awt.Component
- public abstract class Component extends Object implements ImageObserver, MenuContainer, Serializable
- A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

About Java.awt.Container

- java.lang.Object
  - java.awt.Component
    - **java.awt.Container**

- public class **Container** extends Component

- A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components.
- Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).
- Example : Add()

About Java.awt.Panel

- java.lang.Object
    - java.awt.Component
        - java.awt.Container
            - **java.awt.Panel**
- public class **Panel** extends Container implements Accessible
- Panel is the simplest container class. A panel provides space in which an application can attach any other component, including other panels.
- The default layout manager for a panel is the FlowLayout layout manager.

Java.applet.Applet

- java.lang.Object
    - java.awt.Component
        - java.awt.Container
            - java.awt.Panel
                - **java.applet.Applet**
- public class **Applet** extends Panel
    - An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application.
    - The Applet class must be the superclass of any applet that is to be embedded in a Web page or viewed by the Java Applet Viewer. The Applet class provides a standard interface between applets and their environment.

Example - 1

```
import java.awt.*;
import java.applet.*;
public class myapplet extends Applet
{
public void paint(Graphics g)
{
        rawString("Welcome to applet",30,30);
}
}
```

Html file

```
<html>
<head>
<body>
```

```
<applet code = "myapplet.class" width=200 height=100>
abcd
</applet>

</body>
</head>
```

Execution procedure of applet program

- Save the above program with myapplet.java
- Compile the program using
  - Javac myapplet.java
- In order to run the program there are two methods
  - Using web browser
    - executing the applet through html within a java compatible browser such as HotJava, Netscape Navigator or Internet explorer. to execute this method, we need write the HTML file with Applet tag.
    - Save the program with test.html.
    - Open web browser, and type path of file at address bar
  - From console
    - At the console window give the following command
      - Appletviewer test.html

Method – 2

```
import java.awt.*;
import java.applet.*;
public class myapplet extends Applet
{
public void paint(Graphics g)
{
        rawString("Welcome to applet",30,30);
}
}

/*
<applet code = "myapplet.class" width = 200 height = 100>
</applet>
*/
```

- Save the above program with myapplet.java
- Compile it using javac myapplet.java
- Run using appletviewer myapplet.java

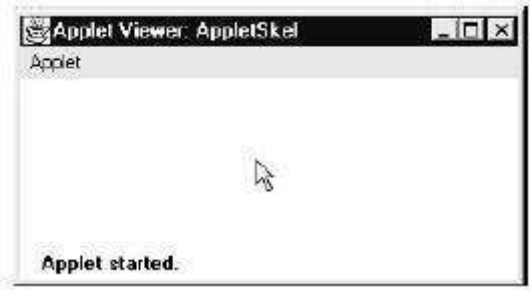
Architecture of an Applet

- Since applet is an window based program. Its architecture is different from the console based programs.
- In window based program we must understand few concepts:
    - First, applets are event driven.
    - how the event-driven architecture impacts the design of an applet.
        - An applet resembles a set of interrupt service routines.
    - Here is how the process works.
        - An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the AWT. This is a crucial point. For the most part, your applet should not enter a ―mode‖ of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the AWT run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution.
    - Second, the user initiates interaction with an applet—not the other way around.
        - As you know, in a no windowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond.
        - For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

Applet Skelton

- 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. efault 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:

Applet Initialization and Termination

When an applet begins, the AWT calls the following methods, in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**.

This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Overriding update( )

In some situations, your applet may need to override another method defined by theAWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** first fills an applet with the default background color and then calls **paint( )**. If you fill the background using a different color in **paint( )**, the user will experience a flash of the default background each time **update( )** is called—that is, whenever the window is repainted.

One way to avoid this problem is to override the **update( )** method so that it performs all necessary display activities. Then have **paint( )** simply call **update( )**. Thus, for some applications, the applet skeleton will override **paint( )** and **update( )**, as shown here:

```
public void update(Graphics g) {
// redisplay your window, here.
}
public void paint(Graphics g) {
update(g);
}
```

- To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:
  - **void drawString(String *message*, int *x*, int *y*)**
    - Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The **drawString( )** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin.
- To set the background color of an applet's window, use **setBackground( )**.
- To set the foreground color use **setForeground( )**.
- These methods are defined by **Component**, and they have the following general forms:
  - **void setBackground(Color *newColor*)**
  - **void setForeground(Color *newColor*)**
    - Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

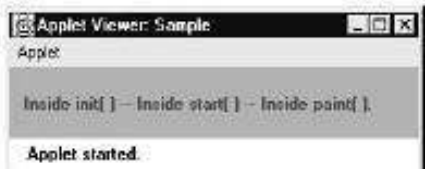
- Color.black
- Color.magenta
- Color.blue
- Color.orange
- Color.cyan
- Color.pink
- Color.darkGra
- Color.red
- Color.gray
- Color.white
- Color.green
- Color.yellow
- Color.lightGray

- For example, this sets the background color to green and the text color to red:
  - **setBackground(Color.green);**
  - **setForeground(Color.red);**
  - We has to set the foreground and background colors is in the **init( )** method
  - we can change these colors during execution of program also

- The default foreground color is black.
- The default background color is light gray.
- we can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:
    - **Color getBackground( )**
    - **Color getForeground( )**
-

Program

```
/* A simple applet that sets the foreground and
background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet{
String msg;
// set the foreground and background colors.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
msg = "Inside init( ) --";
}
// Initialize the string to be displayed.
public void start() {
msg += " Inside start( ) --";
}
// Display msg in applet window.
public void paint(Graphics g) {
msg += " Inside paint( ).";
g.drawString(msg, 10, 30);
}
}
```



- **Using Repaint method**

- As a general rule, an applet writes to its window only when its **update( )** or **paint( )** method is called by the AWT.
- How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls?
- It cannot create a loop inside **paint( )** that repeatedly scrolls the banner

- The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**
- The **repaint( )** method has four forms.
- The simplest version of **repaint( )** is shown here:
    - **void repaint( )**
- This version causes the entire window to be repainted.
- The following version specifies a region that will be repainted:
    - **void repaint(int *left*, int *top*, int *width*, int *height*)**
- Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels
- Calling **repaint( )** is essentially a request that your **applet** be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:
    - **void repaint(long *maxDelay*)**
    - **void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)**

- Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called

Program

```
/* A simple banner applet.
This applet creates a thread that scrolls
the message contained in msg right to left
across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/
public class SimpleBanner extends Applet implements Runnable {
```

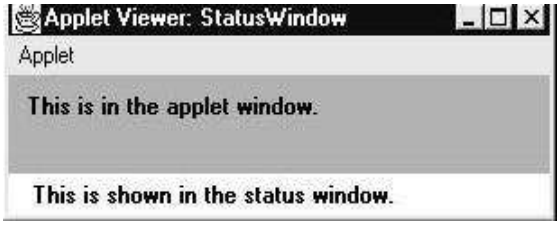
```
String msg = " A Simple Moving Banner.";
Thread t = null;
int state;
boolean stopFlag;
// Set colors and initialize thread.
public void init() {
setBackground(Color.cyan);
setForeground(Color.red);
}
// Start thread
public void start() {
t = new Thread(this);
stopFlag = false;
t.start();
}
// Entry point for the thread that runs the banner.
public void run() {
char ch;
// Display banner
for( ; ; ) {
try {
repaint();
Thread.sleep(250);
ch = msg.charAt(0);
msg = msg.substring(1, msg.length());
msg += ch;
if(stopFlag)
break;
} catch(InterruptedException e) {}
}
}
// Pause the banner.
public void stop() {
stopFlag = true;
t = null;
}
// Display the banner.
public void paint(Graphics g) {
g.drawString(msg, 50, 30);
}
}
```

Using Status window

- In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running.

- To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet{
public void init() {
setBackground(Color.cyan);
}
// Display msg in applet window.
public void paint(Graphics g) {
                rawString("This is in the applet
        window.", 10, 20); showStatus("This is
        shown in the status window.");
}
}
```



Syntax of Applet tag in HTML

The syntax for the standard APPLET tag is shown here. Bracketed items
are optional.

```
< APPLET
[CODEBASE = codebaseURL]
CODE = appletFile
[ALT = alternateText]
[NAME = appletInstanceName]
WIDTH = pixels HEIGHT = pixels
[ALIGN = alignment]
[VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

CODEBASE

CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

CODE

CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

ALT

The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

NAME

NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

WIDTH AND HEIGHT

WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area. ALIGN ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE

These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE

The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

HANDLING OLDER BROWSERS

Some very old web browsers can't execute applets and don't recognize the APPLET tag. Although these browsers are now nearly extinct (having been replaced by Java-compatible ones), you may need to deal with them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your <applet></applet> tags.
If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the <applet></applet> tags and disregard the alternate markup.

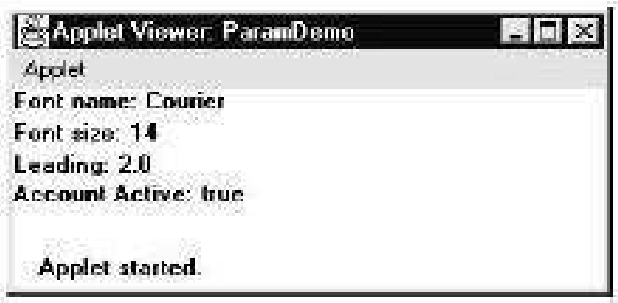
Here's the HTML to start an applet called **SampleApplet** in Java and to display a message in older browsers:

```
<applet code="SampleApplet" width=200 height=40>
If you were driving a Java powered browser,
you'd see &quote;A Sample Applet&quote; here.<p>
</applet>
```

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.
- // 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);
}
}
```