UNIT-1
INTRODUCTION TO JAVA

1.JAVA FEATURES

SIMPLE

Java was designed to be easy for the professional programmer to learn and use effectively. If user already understand the basic concepts of object-oriented programming, learning Java will be even easier. if user 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. Java has another attribute that makes it easy to learn: it makes an effort not to have surprising features. In Java, there are a small number of clearly defined ways to accomplish a given task.

The syntax for Java is, a cleaned-up version of the syntax for C++. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on.

Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.

OBJECT ORIENTED

Object oriented is one which allows the developer to design or create the programs using classes and objects.

Object: An object is one which exists physically.Eg[mango,tiger,..]

Class : A class is a group name which contains the description of an object.

The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects.

RHOBUST[Strong]

Java is intended for writing programs that must be reliable in a variety of ways.Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error-prone. . . . The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.

This feature is also very useful. The Java compiler detects many problems that, in other languages, would show up only at runtime. As for the second point, anyone who has spent hours chasing memory corruption caused by a pointer bug will be very happy with this feature of Java**.**

SECURE

Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.
Java was designed to make certain kinds of attacks impossible, among them:

- Overrunning the runtime stack—a common attack of worms and viruses
- Corrupting memory outside its own process space
- Reading or writing files without permission

ARCHITECTURE NEUTRAL

The compiler generates an architecture-neutral object file format—the compiled code is executable on many processors, given the presence of the Java runtime system. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.

PORTABLE

Unlike C and C++, there are no "implementation-dependent" aspects of the specification.The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.
For example, an int in Java is always a 32-bit integer. In C/C++, int can mean a 16-bit integer, a 32-bit integer, or any other size that the compiler vendor likes. The only restriction is that the int type must have at least as many bytes as a short int and cannot have more bytes than a long int. Having a fixed size for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. Strings are saved in a standard Unicode format**.**

Interpreted

The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.

High Performance

While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on.

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.
Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Dynamic

In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients. In Java, finding outruntime type information is straightforward.

2.JAVA PROGRAMMING ENVIRONMENT

To compile Java programs, you will need to download and install the Java Development Kit (JDK). This is available from Sun's website. Other hardware and operating system vendors also supply Java Development Kits for their platforms, although they may change the name of the kit. Sun produces JDKs for Windows, Linux, and Solaris.
There are numerous environments you can use to develop your Java programs. You can choose to write your programs in a text editor and then compile them using the command line, or you can use an integrated development environment (IDE). IDEs like NetBeans and Eclipse provide many useful functions such as syntax error checking, code completion, automatic compilation and debugging, which you may find useful, especially if this is your first foray into programming.

The Java Compiler

The JDK consists of a set of tools necessary to construct Java programs. The most notable tool in the JDK is the Java compiler, also known as javac.javac compiles Java source files into executable Java class files. Source files are text files with a .java file name extension. You can create such files with a text editor, like Notepad, or an IDE. `javac` then compiles these files into loadable and executable class files, using the .class extension.

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on cell phones and other small devices
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Sun's term for a bug fix release
NetBeans	—	Sun's integrated development environment

The bytecode

It should be clear from the above paragraph that the Java compiler compiles source code text files with the extension .java into executable code usually confined into a class file with the .class extension. Such code is called Bytecode.

The JIT compiler

Being compiled halfway through, it is the job of the Java Virtual Machine to compile the rest of the program to native code at the time of its execution making Java code follow the "Write Once, Run Anywhere" (WORA) policy. The compiler used to compile bytecode into machine-code at runtime is called the *Just-In-Time* or *JIT compiler*. Once a piece of code is compiled by the JVM to execution code, the code is used and re-used again and again, to speed up execution.

The Java Runtime Environment

The Java Runtime Environment, or JRE, is responsible for the execution of Java programs. The Sun JDK also includes the JRE. The JRE however can also be installed and used without installing the JDK which is useful if you wish to execute Java programs but not build them. The JRE helps load Java programs into the memory and executes them.

3.FUNDAMENTAL PROGRAMMINS STRUCTURES IN JAVA

Data Types : A data type is one which can store the type of the data in a variable.

The Simple 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 wholevalued 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.

Integer Types

The integer types are for numbers without fractional parts. Negative values are allowed.
Java provides the four integer types shown in Table

Type	Storage Requirement	Range (Inclusive)
int	4 bytes	–2,147,483,648 to 2,147,483, 647 (just over 2 billion)
short	2 bytes	–32,768 to 32,767
long	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
byte	1 byte	–128 to 127

Long integer numbers have a suffix L (for example, 4000000000L). Hexadecimal numbers have a prefix 0x (for example, 0xCAFE). Octal numbers have a prefix 0. For example, 010 is 8.

Byte

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types.
Byte variables are declared by use of the byte keyword. For example, the following declares two byte variables called b and c: **byte b, c;**
Short : short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably theleast-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 bytes, shorts, ints, and literal numbers, the entire expression is promoted to int before the calculation is done.

The int type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math.

long

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when big, whole numbers are needed.

Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table

Type	Storage Requirement	Range
float	4 bytes	approximately ±3.40282347E+38F (6–7 significant decimal digits)
double	8 bytes	approximately ±1.79769313486231570E+308 (15 significant decimal digits)

The name double refers to the fact that these numbers have twice the precision of the float type.

Numbers of type float have a suffix F (for example, 3.402F). Floating-point numbers without an F suffix (such as 3.402) are always considered to be of type double.

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.Eg : **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. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, double is the best choice.

The char Type

The char type is used to describe individual characters. Most commonly, these will be character constants. For example, 'A' is a character constant with value 65. It is different from "A", a string containing a single character. Unicode code units can be expressed as hexadecimal values that run from \u0000 to \uFFFF. For example, \u2122 is the trademark symbol (™) and \u03C0 is the Greek letter pi (π).

Escape Sequences for Special Characters

Escape Sequence	Name	Unicode Value
\b	Backspace	\u0008
\t	Tab	\u0009
\n	Linefeed	\u000a
\r	Carriage return	\u000d
\"	Double quote	\u0022
\'	Single quote	\u0027
\\	Backslash	\u005c

The boolean Type

The boolean type has two values, false and true. It is used for evaluating logical conditions.user cannot convert between integers and boolean values.

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.

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. The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value. To declare more than one variable of the specified type, use a comma-separated list.

EG:

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

Java allows variables to be declared within any block. a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a scope. Thus, each time you start a new block, you are creating a new scope. Ascope determines what objects are visible to other parts of your program. It also determinesthe lifetime of those objects.

OPERATORS

Operator is a symbol which can act b/w two operands or variables.

Arithmetic Operators

The usual arithmetic operators + – * / are used in Java for addition, subtraction, multiplication, and division. The / operator denotes integer division if both arguments are integers, and floating-point division otherwise. Integer remainder (sometimes called
modulus) is denoted by %. For example, 15 / 2 is 7, 15 % 2 is 1, and 15.0 / 2 is 7.5.
Note that integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

Increment and Decrement Operators

The increment operator is used to increment the value of a variable default by one.The decrement operator is used to decrement the value of the variable default by one.
Because these operators change the value of a variable, they cannot be applied to numbers themselves. For example, 4++ is not a legal statement.
There are actually two forms of these operators; the "postfix" form of the operator that is placed after the operand. There is also a prefix form, ++n. Both change the value of the variable by 1. The difference between the two only appears when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates
to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

Relational and boolean Operators

Java has the full complement of relational operators. To test for equality you use a double
equal sign, ==. For example, the value of
3 == 7 is false.
Use a != for inequality. For example, the value of
3 != 7 is true.
Finally, user have the usual < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators.
Java, following C++, uses && for the logical "and" operator and || for the logical "or" operator. As user can easily remember from the != operator, the ! is the logical negation operator. The && and || operators are evaluated in "short circuit" fashion.
The second argument is not evaluated if the first argument already determines the value. If user combine two expressions with the && operator, expression1 && expression2
and the truth value of the first expression has been determined to be false, then it is impossible for the result to be true. Thus, the value for the second expression is not calculated. This behavior can be exploited to avoid errors. For example, in the expression
x != 0 && 1 / x > x + y // no division by 0
the second part is never evaluated if x equals zero. Thus, 1 / x is not computed if x is zero, and no divide-by-zero error can occur.
Similarly, the value of expression1 || expression2 is automatically true if the first expression is true, without evaluation of the second expression.
Finally, Java supports the **ternary ?: operator** that is occasionally useful. The expression
condition ? expression1 : expression2
evaluates to the first expression if the condition is true, to the second expression otherwise.
For example,
x < y ? x : y gives the smaller of x and y.

Bitwise Operators

When working with any of the integer types, user have operators that can work directly with the bits that make up the integers. This means that user can use masking techniques to get at individual bits in a number. The bitwise operators are

& (“and”) | (“or”) ^ (“xor”) ~ (“not”)
These operators work on bit patterns. For example, if n is an integer variable, then
int fourthBitFromRight = (n & 8) / 8;
gives you a 1 if the fourth bit from the right in the binary representation of n is 1, and 0 if not. Using & with the appropriate power of 2 lets you mask out all but a single bit.
When applied to boolean values, the & and | operators yield a boolean value. Theseoperators are similar to the && and || operators, except that the & and | operators are notevaluated in “short circuit” fashion. That is, both arguments are first evaluated before the result is computed.
There are also >> and << operators, which shift a bit pattern to the right or left. These
operators are often convenient when you need to build up bit patterns to do bit **masking:**
int fourthBitFromRight = (n & (1 << 3)) >> 3;
There is no <<< operator.

Operator Precedence Operator Precedence

Operators	Associativity
[] . () (method call)	Left to right
! ~ ++ -- + (unary) - (unary) () (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Right to left

Strings

Java strings are sequences of Unicode characters. For example, the string "Java\u2122" consists of the five Unicode characters J, a, v, a, and ™. Java does not have a built-in string type. Instead, the standard Java library contains a predefined class called, naturally enough, String. Each quoted string is an instance of the String class:

String e = ""; // an empty string

String greeting = "Hello";

Substrings

You extract a substring from a larger string with the substring method of the String class.

For example,

String greeting = "Hello";

String s = greeting.substring(0, 3);

creates a string consisting of the characters "Hel".

The second parameter of substring is the first position that you do not want to copy.

In our case, we want to copy positions 0, 1, and 2 position.As substring counts it, this means from position 0 inclusive to position 3 exclusive.

There is one advantage to the way substring works: Computing the length of the substring is easy. The string s.substring(a, b) always has length b - a. For example, the substring "Hel" has length 3 – 0 = 3.

Concatenation

Java, like most programming languages, allows you to use the + sign to join (concatenate) two strings.

String expletive = "created";

String PG13 = "modified";

String message = created + PG13;

The preceding code sets the variable message to the string "created+deleted".The + sign joins two strings in the order.

When you concatenate a string with a value that is not a string, the latter is converted to a string.

For example,

int age = 13;

String rating = "PG" + age;

sets rating to the string "PG13".

This feature is commonly used in output statements. **For example,**

System.out.println("The answer is " + answer);

is perfectly acceptable and will print the answer.

Strings are immutable i.e., immutable is one which does not modify the contents of the object.

Testing Strings for Equality

To test whether two strings are equal, use the equals method. The expression

s.equals(t)

returns true if the strings s and t are equal, false otherwise. Note that s and t can be string variables or string constants. For example, the expression

"Hello".equals(greeting)

is perfectly legal. To test whether two strings are identical except for the upper/lowercase letter distinction, use the equalsIgnoreCase method.

"Hello".equalsIgnoreCase("hello")

Do not use the == operator to test whether two strings are equal! It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical
strings in different places.

```
String greeting = "Hello"; //initialize greeting to a string
if (greeting == "Hello") . . .
// probably true
if (greeting.substring(0, 3) == "Hel") . . .
// probably false
```

If the virtual machine would always arrange for equal strings to be shared, then you could use the == operator for testing equality. But only string constants are shared, not strings that are the result of operations like + or substring. Therefore, never use == to compare strings lest you end up with a program with the worst kind of bug—an intermittent one that seems to occur randomly.

The length method yields the number of code units required for a given string in the

UTF-16 encoding. For example:

```
String greeting = "Hello";
int n = greeting.length(); // is 5.
```

To get the true length, that is, the number of code points, call

int cpCount = greeting.codePointCount(0, greeting.length());

The call s.charAt(n) returns the code unit at position n, where n is between 0 and s.length() – 1.

For example:

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

To get at the ith code point, use the statements

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```


API **java.lang.String 1.0**

- `char charAt(int index)`
  returns the code unit at the specified location. You probably don't want to call this method unless you are interested in low-level code units.
- `int codePointAt(int index)` **5.0**
  returns the code point that starts or ends at the specified location.
- `int offsetByCodePoints(int startIndex, int cpCount)` **5.0**
  returns the index of the code point that is `cpCount` code points away from the code point at `startIndex`.
- `int compareTo(String other)`
  returns a negative value if the string comes before `other` in dictionary order, a positive value if the string comes after `other` in dictionary order, or 0 if the strings are equal.
- `boolean endsWith(String suffix)`
  returns `true` if the string ends with `suffix`.
- `boolean equals(Object other)`
  returns `true` if the string equals `other`.
- `boolean equalsIgnoreCase(String other)`
  returns `true` if the string equals `other`, except for upper/lowercase distinction.
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`
  returns the start of the first substring equal to the string `str` or the code point `cp`, starting at index 0 or at `fromIndex`, or –1 if `str` does not occur in this string.
- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
- `int lastindexOf(int cp)`
- `int lastindexOf(int cp, int fromIndex)`
  returns the start of the last substring equal to the string `str` or the code point `cp`, starting at the end of the string or at `fromIndex`.
- `int length()`
  returns the length of the string.
- `int codePointCount(int startIndex, int endIndex)` **5.0**
  returns the number of code points between `startIndex` and `endIndex - 1`. Unpaired surrogates are counted as code points.
- `String replace(CharSequence oldString, CharSequence newString)`
  returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`. You can supply `String` or `StringBuilder` objects for the `CharSequence` parameters.
- `boolean startsWith(String prefix)`
  returns `true` if the string begins with `prefix`.
- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`
  returns a new string consisting of all code units from `beginIndex` until the end of the string or until `endIndex - 1`.

INPUT AND OUTPUT :

Reading Input

To read console input, first construct a Scanner that is attached to **System.in:**

Scanner in = new Scanner(System.in);

Now you use the various methods of the Scanner class to read input. **For example, the**

nextLine method reads a line of input.

System.out.print("What is your name? ");
String name = in.nextLine();

Here, we use the nextLine method because the input might contain spaces. To read a single word (delimited by whitespace), call

String firstName = in.next();

To read an integer, use the nextInt method.

System.out.print("How old are you? ");
int age = in.nextInt();

Similarly, the nextDouble method reads the next floating-point number.

Finally, note the line

import java.util.*;

at the beginning of the program. The Scanner class is defined in the java.util package. Whenever you use a class that is not defined in the basic java.lang package, user need to use an import directive.

Eg:

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates console input.
5.  * @version 1.10 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class InputTest
9. {
10.    public static void main(String[] args)
11.    {
12.       Scanner in = new Scanner(System.in);
13.
14.       // get first input
15.       System.out.print("What is your name? ");
16.       String name = in.nextLine();
17.
18.       // get second input
19.       System.out.print("How old are you? ");
20.       int age = in.nextInt();
21.
22.       // display output on console
23.       System.out.println("Hello, " + name + ". Next year, you'll be " + (age + 1));
24.    }
25. }
```


The Scanner class is not suitable for reading a password from a console since the
input is plainly visible to anyone. Java SE 6 introduces a Console class specifically for this
purpose. To read a password, use the following code:

```
Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");
```

Formatting Output

User can print a number x to the console with the statement System.out.print(x). That command
will print x with the maximum number of non-zero digits for that type. For example,

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

prints

3333.3333333333335

That is a problem if you want to display, for example, dollars and cents.

In early versions of Java, formatting numbers was a bit of a hassle. Fortunately, Java SE 5.0 brought back the venerable printf method from the C library. For example, the call

```
System.out.printf("%8.2f", x);
```

prints x with a field width of 8 characters and a precision of 2 characters. That is, the printout contains a leading space and the seven characters

3333.33

User can supply multiple parameters to printf. For example:

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

Each of the format specifiers that start with a % character is replaced with the corresponding argument. The conversion character that ends a format specifier indicates the type of the value to be formatted: f is a floating-point number, s a string, and d a decimal integer.

File Input and Output

To read from a file, construct a Scanner object from a File object, like this:

```
Scanner in = new Scanner(new File("myfile.txt"));
```

If the file name contains backslashes, remember to escape each of them with an additional backslash: "c:\\mydirectory\\myfile.txt".

To write to a file, construct a PrintWriter object. In the constructor, simply supply the file name:

```
PrintWriter out = new PrintWriter("myfile.txt");
```


Conversions for printf

Conversion Character	Type	Example
d	Decimal integer	159
x	Hexadecimal integer	9f
o	Octal integer	237
f	Fixed-point floating-point	15.9
e	Exponential floating-point	1.59e+01
g	General floating-point (the shorter of e and f)	—
a	Hexadecimal floating-point	0x1.fccdp3
s	String	Hello
c	Character	H
b	boolean	true
h	Hash code	42628b2
tx	Date and time	See Table 3–7
%	The percent symbol	%
n	The platform-dependent line separator	—

CONTROL FLOW

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: **selection, iteration,** and jump. Selection statements allow user's program to choose different paths of execution based upon the outcome of an expression or the state of a variable. Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops). Jump statements allow the program to execute in a nonlinear fashion**.**

Conditional Statements

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. The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

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

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.
Most often, the expression used to control the if will involve the relational operators.

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```
if(i == 10) {
if(j < 20) a = b;
if(k > 100) c = d; // this if is
else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final else is not associated with if(j<20), because it is not in the same block (even though it is the nearest if without an else). Rather, the final else is associated with if(i==10). The inner else refers to if(k>100), because it is the closest if within the same block.

The if-else-if Ladder

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

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


...
else
statement;
The if statements are executed from the top down. As soon as one of the conditionscontrolling the if is true, the statement associated with that if is executed, and the restof the ladder is bypassed. If none of the conditions is true, then the final else statementwill be executed. The final else acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all otherconditions are false, then no action will take place.
Here is a program that uses an if-else-if ladder to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class IfElse {
public static void main(String args[]) {
int month = 4; // April
String season;
if(month == 12 || month == 1 || month == 2)
season = "Winter";
else if(month == 3 || month == 4 || month == 5)
season = "Spring";
else if(month == 6 || month == 7 || month == 8)
season = "Summer";
else if(month == 9 || month == 10 || month == 11)
season = "Autumn";
else
season = "Bogus Month";
System.out.println("April is in the " + season + ".");
}
}
```

Here is the output produced by the program:
April is in the Spring.

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 specifiedin the case statements must be of a type compatible with the expression. Each casevalue must be a unique literal (that is, it must be a constant, not a variable). Duplicatecase values are not allowed.The switch statement works like this: The value of the expression is compared witheach of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken. The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement. This has the effect of "jumping out" of the switch.

```
// A simple example of the switch.
class SampleSwitch {
public static void main(String args[]) {
for(int i=0; i<6; i++)
switch(i) {
case 0:
System.out.println("i is zero.");
break;
case 1:
System.out.println("i is one.");
break;
case 2:
System.out.println("i is two.");
break;
```

```
case 3:
System.out.println("i is three.");
break;
default:
System.out.println("i is greater than 3.");
}
}
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

ITERATION STATEMENTS

Java's iteration statements are for, while, and do-while. These statements create loops, a loop repeatedly executes the same set of instructions until a termination condition is true.

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.

Here is a while loop that counts down from 10, printing exactly ten lines of "tick":

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


}
When you run this program, it will "tick" ten times:
tick 10 tick 9 tick 8 tick 7 tick 6 tick 5 tick 4 tick 3 tick 2 tick 1

do-while

The do-while loop always executes
its body at least once, because its conditional expression is at the bottom of the loop. Its general form is
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. As with all of Java's loops, condition must be a Boolean expression.

```
// 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

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 loop control variable, which acts as a counter that controls the loop. It is important to understand
that the initialization expression is only executed once. 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. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
// 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.

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

This program generates the following output:

```
i: 0 i: 1 i: 2 i: 3 i: 4 i: 5 i: 6 i: 7 i: 8 i: 9
Loop complete.
```

The break statement can be used with any of Java's loops, including intentionally infinite loops.

Using continue

Sometimes it is useful to force an early iteration of a loop. In while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. // Demonstrate continue.

```
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

0 1
2 3
4 5
6 7
8 9

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 element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, first user 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. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named month_days with the type "array of int":

int month_days[];

Although this declaration establishes the fact that month_days is an array variable, no array actually exists. In fact, the value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, you must allocate one using new and assign it to month_days. new is a special operator that allocates memory.

The general form of new as it applies to one-dimensional
arrays appears as follows:

array-var = new type[size];

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array-var is the array variable that is linked to the array. That is, to use new to allocate an array, you must specify the type and number of elements to allocate.

The elements in the array allocated by new will automatically be initialized to zero.

This example allocates a 12-element array of integers and links them to month_days.

month_days = new int[12];

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

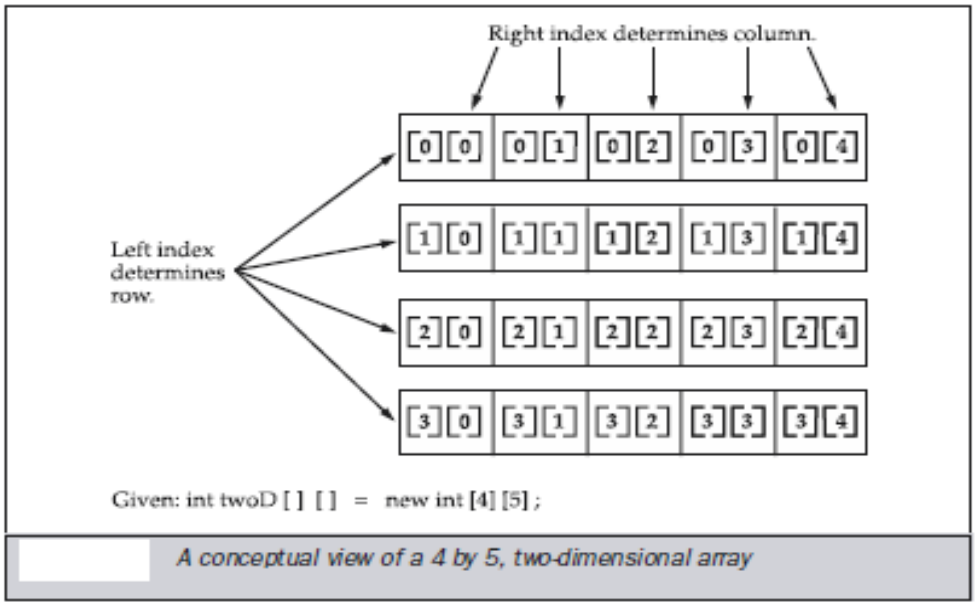

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
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. 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.



A conceptual view of a 4 by 5, two-dimensional array

```
// 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:

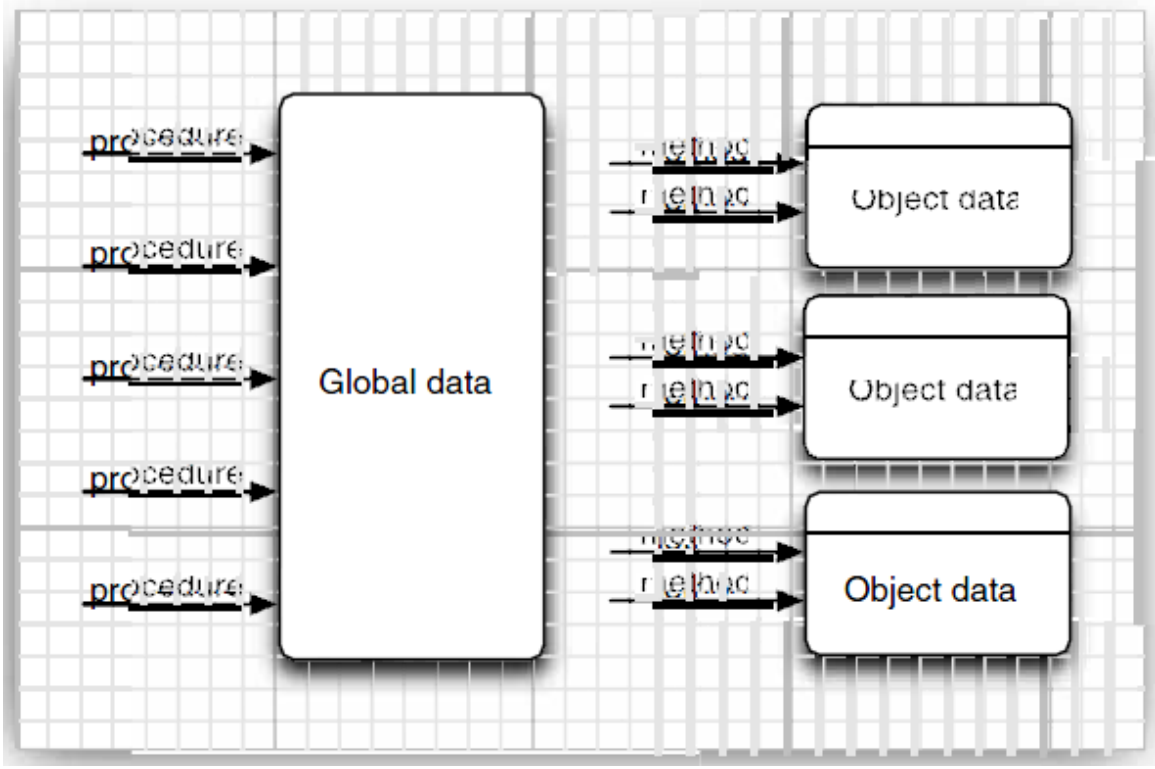
```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

UNIT-2

INTRODUCTION TO OBJECT ORIENTED PROGRAMMING

Object-oriented programming (or OOP for short) is the dominant programming paradigm these days, having replaced the "structured," procedural programming techniques that were developed in the 1970s. Java is totally object oriented, and you have to be familiar with OOP to become productive with Java.

An object-oriented program is made of objects. Each object has a specific functionality that is exposed to its users, and a hidden implementation.Traditional structured programming consists of designing a set of procedures (or algorithms)
to solve a problem. After the procedures were determined, the traditional next step was to find appropriate ways to store the data.



Procedural vs. OO programming

Classes

A class is the template or blueprint from which objects are made. Thinking about classes as cookie cutters. Objects are the cookies themselves. When you construct an object from a class, you are said to have created an instance of the class. All code that you write in Java is inside a class. The standard Java library supplies several thousand classes for such diverse purposes as user interface design, dates and calendars, and network programming. create your own classes in Java to describe the objects of the problem domains of your applications.

Encapsulation (sometimes called information hiding) is a key concept in working with objects. Formally, encapsulation is nothing more than combining data and behavior in one package and hiding the implementation details from the user of the object. The data in an object are called its instance fields, and the procedures that operate on the data are called its methods. A specific object that is an instance of a class will have specific values for its instance fields. The set of those values is the current state of the object. Whenever you invoke a method on an object, its state may change. The key to making encapsulation work is to have methods never directly access instance fields in a class other than their own. Programs should interact with object data only through the object's methods. Encapsulation is the

way to give the object its "black box" behavior, which is the key to reuse and reliability. This means a class may totally change how it stores its data, but as long as it continues to use the same methods to manipulate the data, no other object will know or care.

When you do start writing your own classes in Java, another tenet of OOP makes this easier: classes can be built by extending other classes.

Objects

All objects that are instances of the same class share a family resemblance by supporting the same behavior Next, each object stores information about what it currently looks like. This is the object's state. An object's state may change over time, but not spontaneously. A change in the state of an object must be a consequence of method calls.

However, the state of an object does not completely describe it, because each object has a distinct identity. For example, in an order-processing system, two orders are distinct even if they request identical items. Notice that the individual objects that are instances of a class always differ in their identity and usually differ in their state.

1.USING PREDEFINED CLASSES

Objects and Object Variables

To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.

In the Java programming language, you use constructors to construct new instances. A constructor is a special method whose purpose is to construct and initialize objects. Let us look at an example. The standard Java library contains a Date class. Its objects describe points in time, such as "December 31, 1999, 23:59:59 GMT".

Constructors always have the same name as the class name. Thus, the constructor for the Date class is called Date. To construct a Date object, you combine the constructor with the new operator, as follows:

```
new Date()
```

This expression constructs a new object. The object is initialized to the current date and time.

If you like, you can pass the object to a method:

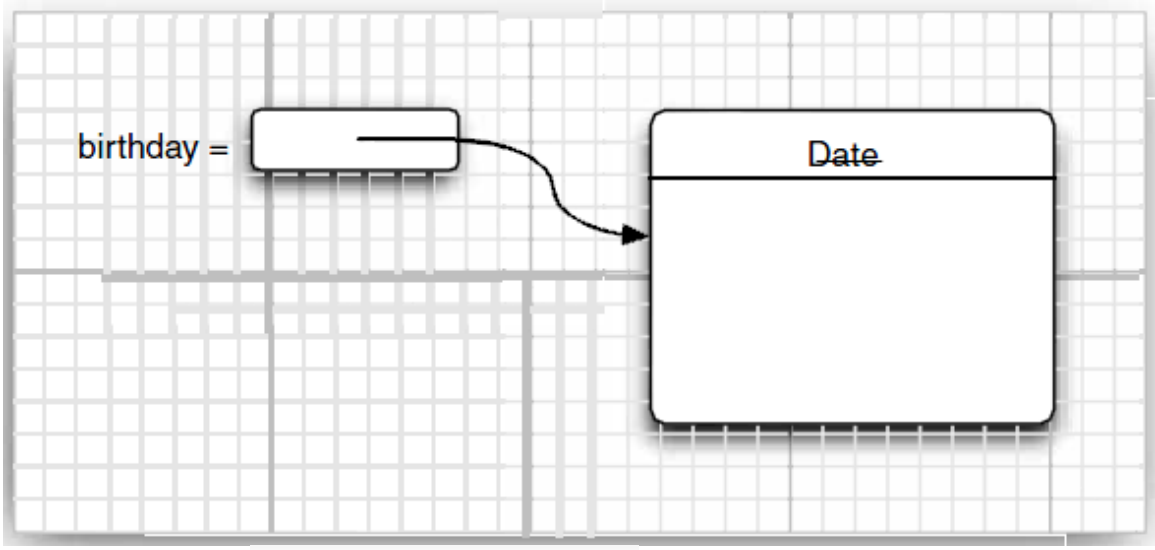
```
System.out.println(new Date());
```

Alternatively, you can apply a method to the object that you just constructed. One of the methods of the Date class is the toString method. That method yields a string representation of the date. Here is how you would apply the toString method to a newly constructed Date object:

```
String s = new Date().toString();
```


In these two examples, the constructed object is used only once. Usually, you will want to hang on to the objects that you construct so that you can keep using them. Simply store the object in a variable:

Date birthday = new Date();



Creating a new object

There is an important difference between objects and object variables. For example, the statement

Date deadline; // deadline doesn't refer to any object

defines an object variable, deadline, that can refer to objects of type Date. It is important to realize that the variable deadline is not an object and, in fact, does not yet even refer to an object. You cannot use any Date methods on this variable at this time. The statement

s = deadline.toString(); // not yet

would cause a compile-time error.

You must first initialize the deadline variable. You have two choices. Of course, you can initialize the variable with a newly constructed object:

deadline = new Date();

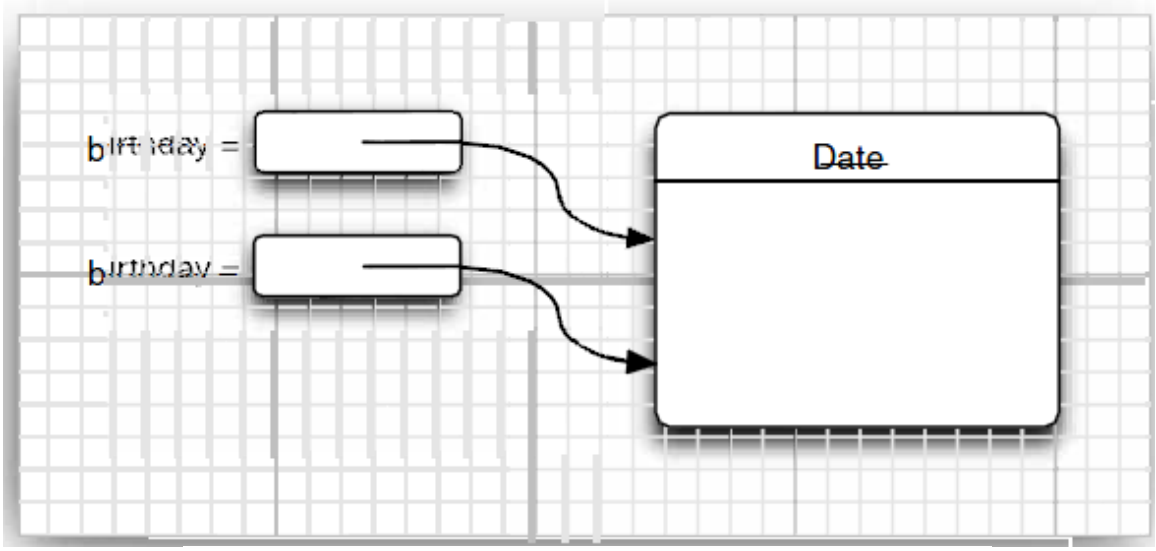
Or you can set the variable to refer to an existing object:

deadline = birthday;

Now both variables refer to the same object It is important to realize that an object variable doesn’t actually contain an object. It only refers to an object. In Java, the value of any object variable is a reference to an object that is stored elsewhere. The return value of the new operator is also a reference. A statement such as

Date deadline = new Date();

has two parts. The expression new Date() makes an object of type Date, and its value is a reference to that newly created object. That reference is then stored in the deadline variable.



Object variables that refer to the same object

User can explicitly set an object variable to null to indicate that it currently refers to no object.

```
deadline = null;
. . .
if (deadline != null)
System.out.println(deadline);
```

If user apply a method to a variable that holds null, then a runtime error occurs.

```
        birthday = null;
String s = birthday.toString(); // runtime error!
```

Variables are not automatically initialized to null. You must initialize them, either by calling new or by setting them to null.

2.DEFINING YOUR OWN CLASSES

An Employee Class

The simplest form for a class definition in Java is

```
class ClassName
{
constructor1
constructor2
. . .
method1
method2
. . .
field1
field2
. . .
}
```


Consider the following, very simplified, version of an Employee class that might be used
by a business in writing a payroll system.

```
class Employee
{
// constructor
public Employee(String n, double s, int year, int month, int day)
{
name = n;
salary = s;
GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
hireDay = calendar.getTime();
}
// a method
public String getName()
{
return name;
}
// more methods
. . .
// instance fields
private String name;
private double salary;
private Date hireDay;
}
```

In the program, we construct an Employee array and fill it with three employee objects:

```
Employee[] staff = new Employee[3];
staff[0] = new Employee("Carl Cracker", . . .);
staff[1] = new Employee("Harry Hacker", . . .);
staff[2] = new Employee("Tony Tester", . . .);
```

Next, we use the raiseSalary method of the Employee class to raise each employee's salary by
5%:

```
for (Employee e : staff)
e.raiseSalary(5);
```

Finally, we print out information about each employee, by calling the getName, getSalary,
and getHireDay methods:

```
for (Employee e : staff)
System.out.println("name=" + e.getName()
+ ",salary=" + e.getSalary()
+ ",hireDay=" + e.getHireDay());
```


Note that the example program consists of two classes: the Employee class and a class EmployeeTest with the public access specifier. The main method with the instructions is contained in the EmployeeTest class.
The name of the source file is EmployeeTest.java because the name of the file must match the name of the public class. You can have only one public class in a source file, but you can have any number of nonpublic classes. Next, when you compile this source code, the compiler creates two class files in the directory: EmployeeTest.class and Employee.class. You start the program by giving the bytecode interpreter the name of the class that contains the main method of program:
java EmployeeTest
The bytecode interpreter starts running the code in the main method in the EmployeeTest class.

Benefits of Encapsulation

```
public String getName()
{
return name;
}
public double getSalary()
{
return salary;
}
public Date getHireDay()
{
return hireDay;
}
```

The point is that the name field is a read-only field. Once you set it in the constructor, there is no method to change it. Thus, we have a guarantee that the name field will never be corrupted.

STATIC FIELDS

If user define a field as static, then there is only one such field per class. In contrast, each object has its own copy of all instance fields. For example, let's suppose we want to assign a unique identification number to each employee. We add an instance field id and
a static field nextId to the Employee class:

```
class Employee
{
. . .
private int id;
```

```
private static int nextId = 1;
}
```

Every employee object now has its own id field, but there is only one nextId field that is shared among all instances of the class. Let's put it another way. If there are 1,000 objects of the Employee class, then there are 1,000 instance fields id, one for each object. But there is a single static field nextId. Even if there are no employee objects, the static field nextId is present. It belongs to the class, not to any individual object.

Let's implement a simple method:

```
public void setId()
{
id = nextId;
nextId++;
}
```

Suppose you set the employee identification number for harry:

```
harry.setId();
```

Then, the id field of harry is set to the current value of the static field nextId, and the value of the static field is incremented:

```
harry.id = Employee.nextId;
Employee.nextId++;
```

Static Constants

Static variables are quite rare. However, static constants are more common. For example, the Math class defines a static constant:

```
public class Math
{
. . .
public static final double PI = 3.14159265358979323846;
. . .
}
```

You can access this constant in your programs as Math.PI.

If the keyword static had been omitted, then PI would have been an instance field of the Math class.

That is, you would need an object of the Math class to access PI, and every Math object would have its own copy of PI.

Another static constant that you have used many times is System.out. It is declared in the System class as follows:

```
public class System
{
. . .
public static final PrintStream out = . . .;
. . .
}
```

STATIC METHODS

Static methods are methods that do not operate on objects. For example, the pow method of the Math class is a static method. The expression

Math.pow(x, a)

computes the power xa. It does not use any Math object to carry out its task. In other words, it has no implicit parameter.

static methods as methods that don't have a this parameter. (In a non static method, the this parameter refers to the implicit parameter of the method

Because static methods don't operate on objects, you cannot access instance fields from a static method. But static methods can access the static fields in their class. Here is an example of such a static method:

```
public static int getNextId()
{
return nextId; // returns static field
}
```

To call this method, you supply the name of the class:

```
int n = Employee.getNextId();
```

Use static methods in two situations:

• When a method doesn't need to access the object state because all needed

parameters are supplied as explicit parameters (example: Math.pow)

• When a method only needs to access static fields of the class (example: Employee.getNextId)

FACTORY METHODS

Another common use for static methods. The NumberFormat class uses factory methods that yield formatter objects for various styles.

```
NumberFormat currencyFormatter =  NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

CONSTRUCTORS

Overloading occurs if several methods have the same name but different parameters. The compiler must sort out which method to call. It picks the correct method by matching the parameter types in the headers of the various methods with the types of the values used in the specific method call. A compile-time error occurs if the compiler cannot match the parameters or if more than one match is possible.

Default Field Initialization

If user don't set a field explicitly in a constructor, it is automatically set to a default value: numbers to 0, boolean values to false, and object references to null.

Default Constructors

A default constructor is a constructor with no parameters. For example, here is a default constructor for the Employee class:

```
public Employee()
{
name = "";
salary = 0;
hireDay = new Date();
}
```

If you write a class with no constructors whatsoever, then a default constructor is provided for you. This default constructor sets all the instance fields to their default values. So, all numeric data contained in the instance fields would be 0, all boolean values would
be false, and all object variables would be set to null.

If a class supplies at least one constructor but does not supply a default constructor, it is illegal to construct objects without construction parameters.

```
  Employee(String name, double salary, int y, int m, int d)
```

With that class, it was not legal to construct default employees. That is, the call

```
e = new Employee();
```

would have been an error

Explicit Field Initialization

Because you can overload the constructor methods in a class, you can obviously build in many ways to set the initial state of the instance fields of your classes. It is always a good idea to make sure that, regardless of the constructor call, every instance field is set
to something meaningful. You can simply assign a value to any field in the class definition. For example:

```
class Employee
{
. . .
private String name = "";
}
```

This assignment is carried out before the constructor executes. This syntax is particularly useful if all constructors of a class need to set a particular instance field to the same value.

The initialization value doesn't have to be a constant value. Here is an example in which a field is initialized with a method call. Consider an Employee class where each employee has an id field. You can initialize it as follows:

```
class Employee
{
. . .
static int assignId()
{
int r = nextId;
nextId++;
return r;
}
. . .
private int id = assignId();
}
```

Parameter Names

We have generally opted for single-letter parameter names:

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

However, the drawback is that you need to read the code to tell what the n and s parameters mean.

Some programmers prefix each parameter with an "a":

```
public Employee(String aName, double aSalary)
{
name = aName;
salary = aSalary;
}
```

Calling Another Constructor

The keyword this refers to the implicit parameter of a method. However, the keyword has a second meaning.

If the first statement of a constructor has the form this(. . .), then the constructor calls another constructor of the same class. Here is a typical example:

```
public Employee(double s)
{
// calls Employee(String, double)
this("Employee #" + nextId, s);
nextId++;
}
```

When you call new Employee(60000), then the Employee(double) constructor calls the

Employee(String, double) constructor.

Using the this keyword in this manner is useful—you only need to write common construction code once**.**

Initialization Blocks

You have already seen two ways to initialize a data field:

- By setting a value in a constructor
- By assigning a value in the declaration

There is actually a third mechanism in Java; it's called an initialization block. Class declarations can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed.

For example:

```
class Employee
{
public Employee(String n, double s)
{
name = n;
salary = s;
}
public Employee()
{
name = "";
salary = 0;
}
. . .
private static int nextId;
private int id;
private String name;
private double salary;
. . .
// object initialization block
{
id = nextId;
nextId++;
}
}
```

When a constructor is called:

1. All data fields are initialized to their default value (0, false, or null).
2. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. If the first line of the constructor calls a second constructor, then the body of the second constructor is executed.
4. The body of the constructor is executed.

PACKAGES

Java allows you to group classes in a collection called a package. Packages are convenient for organizing your work and for separating your work from code libraries provided by others.

The standard Java library is distributed over a number of packages, including java.lang, java.util, java.net, and so on. The standard Java packages are examples of hierarchical packages. Just as you have nested subdirectories on your hard disk, you can organize packages by using levels of nesting. All standard Java packages are inside the java and javax package hierarchies.

The main reason for using packages is to guarantee the uniqueness of class names. Suppose two programmers come up with the bright idea of supplying an Employee class. As long as both of them place their class into different packages, there is no conflict.

In fact, to absolutely guarantee a unique package name, Sun recommends that you use your company's Internet domain name (which is known to be unique) written in reverse. You then use subpackages for different projects. For example, horstmann.com is a domain that one of the authors registered. Written in reverse order, it turns into the package com.horstmann. That package can then be further subdivided into subpackages such as com.horstmann.corejava.

From the point of view of the compiler, there is absolutely no relationship between nested packages. For example, the packages java.util and java.util.jar have nothing to do with each other. Each is its own independent collection of classes.

CLASS IMPORTATION

A class can use all classes from its own package and all public classes from other packages. You can access the public classes in another package in two ways. The first is simply to add the full package name in front of every class name. For example:

java.util.Date today = new java.util.Date();

That is obviously tedious. The simpler, and more common, approach is to use the import statement. The point of the import statement is simply to give you a shorthand to refer to the classes in the package. Once you use import, you no longer have to give the classes their full names. You can import a specific class or the whole package. You place import statements at the top of your source files .

For example, you can import all classes in the java.util package with the statement

import java.util.*;

Then you can use

Date today = new Date();

without a package prefix. You can also import a specific class inside a package:
import java.util.Date;
The java.util.* syntax is less tedious. It has no negative effect on code size. However, if you import classes explicitly, the reader of your code knows exactly which classes you use.
However, note that you can only use the * notation to import a single package. You cannot use import java.* or import java.*.* to import all packages with the java prefix.
Most of the time, you just import the packages that you need, without worrying too much about them. The only time that you need to pay attention to packages is when you have a name conflict. For example, both the java.util and java.sql packages have a Date class. Suppose you write a program that imports both packages.
import java.util.*;
import java.sql.*;
If you now use the Date class, then you get a compile-time error:
Date today; // ERROR--java.util.Date or java.sql.Date?
The compiler cannot figure out which Date class you want. You can solve this problem by adding a specific import statement:
import java.util.*;
import java.sql.*;
import java.util.Date;
What if you really need both Date classes? Then you need to use the full package name with every class name.
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date(...);
Locating classes in packages is an activity of the compiler. The bytecodes in class files always use full package names to refer to other classes.

INHERITANCE

A class can inherits all the properties of another class is called inheritance.
Define a Manager class that inherits from the Employee class.User use the Java keyword extends to denote inheritance.

```
class Manager extends Employee
{
added methods and fields
}
```

The keyword extends indicates that you are making a new class that derives from an existing class. The existing class is called the

superclass, base class, or parent class. The new class is called the subclass, derived class, or child class. The terms superclass and subclass are those most commonly used by Java programmers, although some programmers prefer the parent/child analogy, which also ties in nicely with the "inheritance" them The Employee class is a superclass, but not because it is superior to its subclass or contains more functionality. In fact, the opposite is true: subclasses have more functionality than their superclasses. For example, as you will see when we go over the rest of the Manager class code, the Manager class encapsulates more data and has more functionality than its superclass Employee.

Our Manager class has a new field to store the bonus, and a new method to set it:

```
class Manager extends Employee
{
. . .
public void setBonus(double b)
{
bonus = b;
}
private double bonus;
}
```

There is nothing special about these methods and fields. If you have a Manager object, you can simply apply the setBonus method.

Manager boss = . . .; boss.setBonus(5000);

if you have an Employee object, you cannot apply the setBonus method—it is not among the methods that are defined in the Employee class.

However, you can use methods such as getName and getHireDay with Manager objects. Even though these methods are not explicitly defined in the Manager class, they are automatically inherited from the Employee superclass.

Similarly, the fields name, salary, and hireDay are inherited from the superclass. Every Manager object has four fields: name, salary, hireDay, and bonus.

When defining a subclass by extending its superclass, you only need to indicate the differences between the subclass and the superclass. When designing classes, you place the most general methods into the superclass and more specialized methods in the subclass.

Factoring out common functionality by moving it to a superclass is common in object-oriented programming.

However, some of the superclass methods are not appropriate for the Manager subclass. In particular, the getSalary method should return

the sum of the base salary and the bonus. You need to supply a new method to override the superclass method:

```
class Manager extends Employee
{
. . .
public double getSalary()
{
. . .
}
. . .
}
```

it appears to be simple—just return the sum of the salary and bonus fields:

```
public double getSalary()
{
return salary + bonus; // won't work
}
```

The getSalary method of the Manager class has no direct access to the private fields of the superclass. This means that the getSalary method of the Manager class cannot directly access the salary field, even though every Manager object has a field called salary. Only
the methods of the Employee class have access to the private fields. If the Manager methods want to access those private fields, they have to do what every other method does—use the public interface, in this case, the public getSalary method of the Employee class.

User need to call getSalary instead of simply accessing the salary field.

```
public double getSalary()
{
double baseSalary = getSalary(); // still won't work
return baseSalary + bonus;
}
```

The problem is that the call to getSalary simply calls itself, because the Manager class has a getSalary method

The consequence is an infinite set of calls to the same method, leading to a program crash. We need to indicate that we want to call the getSalary method of the Employee superclass, not the current class. You use the special keyword super for this purpose. The call

```
super.getSalary()
```

calls the getSalary method of the Employee class. Here is the correct version of the getSalary method for the Manager class:

```
public double getSalary()
{
double baseSalary = super.getSalary();
return baseSalary + bonus;
```

```
}
```

a subclass can add fields, and it can add or override methods of the superclass. However, inheritance can never take away any fields or methods.

Finally, let us supply a constructor.

```
public Manager(String n, double s, int year, int month, int day)
{
super(n, s, year, month, day);
bonus = 0;
}
```

Here, the keyword super has a different meaning. The instruction

```
super(n, s, year, month, day);
```

is shorthand for "call the constructor of the Employee superclass with n, s, year, month, and day as parameters."

Because the Manager constructor cannot access the private fields of the Employee class, it must initialize them through a constructor. The constructor is invoked with the special super syntax. The call using super must be the first statement in the constructor for the subclass. If the subclass constructor does not call a superclass constructor explicitly, then the default (no-parameter) constructor of the superclass is invoked. If the superclass has no default constructor and the subclass constructor does not call another superclass constructor explicitly, then the Java compiler reports an error.

POLYMORPHISM

The ability to use the same method name with different forms is called polymorphism.

Dynamic Binding

It is important to understand what happens when a method call is applied to an object.

1. The compiler looks at the declared type of the object and the method name. Let's say we call x.f(param), and the implicit parameter x is declared to be an object of class C.

Note that there may be multiple methods, all with the same name, f, but with different parameter types. For example, there may be a method f(int) and a method f(String).

The compiler enumerates all methods called f in the class C and all public methods called f in the superclasses of C.

Now the compiler knows all possible candidates for the method to be called.

2. Next, the compiler determines the types of the parameters that are supplied in the method call. If among all the methods called f there is a unique method whose parameter types are a best match for the

supplied parameters, then that method is chosen to be called. This process is called overloading resolution. For example, in a call x.f("Hello"), the compiler picks f(String) and not f(int). The situation can get complex because of type conversions (int to double, Manager to Employee, and so on). If the compiler cannot find any method with matching parameter types or if multiple methods all match after applying conversions, then the compiler reports an error.

Now the compiler knows the name and parameter types of the method that needs to be called.

3. If the method is private, static, final, or a constructor, then the compiler knows exactly which method to call. This is called static binding. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtimeruntime. example, the compiler would generate an instruction to call f(String) with dynamic binding.

4. When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the actual type of the object to which x refers. Let's say the actual type is D, a subclass of C. If the class D defines a method f(String), that method is called. If not, D's superclass is searched for a method f(String), and so on.

Dynamic binding has a very important property: it makes programs extensible without the need for modifying existing code. Suppose a new class Executive is added and there is the possibility that the variable e refers to an object of that class. The code containing the call e.getSalary() need not be recompiled. The Executive.getSalary() method is called automatically if e happens to refer to an object of type Executive.

FINAL CLASSES AND METHODS

Classes that cannot be extended are called final classes, and you use the final modifier in the definition of the class to indicate this. For example, let us suppose we want to prevent others from subclassing the Executive class. Then, we simply declare the class by using the final modifier as follows:

```
final class Executive extends Manager
{
. . .
}
```

You can also make a specific method in a class final. If you do this, then no subclass can override that method. (All methods in a final class are automatically final.) For example:

```
class Employee
{
```

```
. . .
public final String getName()
{
return name;
}
. . .
}
```

There is only one good reason to make a method or class final: to make sure that the semantics cannot be changed in a subclass. For example, the getTime and setTime methods of the Calendar class are final. This indicates that the designers of the Calendar class have taken over responsibility for the conversion between the Date class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the String class is a final class. That means nobody can define a subclass of String. In other words, if you have a String reference, then you know it refers to a String and nothing but a String. Some programmers believe that you should declare all methods as final unless you have a good reason that you want polymorphism. In fact, in C++ and C#, methods do not use polymorphism unless you specifically request it. That may be a bit extreme, but we agree that it is a good idea to think carefully about final methods and classes when you design a class hierarchy.

In the early days of Java, some programmers used the final keyword in the hope of avoiding the overhead of dynamic binding. If a method is not overridden, and it is short, then a compiler can optimize the method call away—a process called inlining. For example, inlining the call e.getName() replaces it with the field access e.name. This is a worthwhile improvement— CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one. However, if getName can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the overriding code may do.

Fortunately, the just-in-time compiler in the virtual machine can do a better job than a traditional compiler. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called, and not actually overridden, the just-in-time compiler can inline the method.

ABSTRACT CLASSES

To declare an abstract method, use this general form:

abstract type name(parameter-list);

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new
operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

```
// 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();
b.callme();
b.callmetoo();
}
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class A implements a
concrete method called callmetoo( ). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.
Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java’s approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

// 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());
}
}
```

CASTING

The process of forcing a conversion from one type to

another is called casting. The Java programming language has a special notation for casts. For example,
double x = 3.405;
int nx = (int) x;
converts the value of the expression x into an integer, discarding the fractional part.
To convert a floating-point number to an integer, you also need to convert an object reference from one class to another. To actually make a cast of an object reference, you use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast.
For example:
Manager boss = (Manager) staff[0];
There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the ManagerTest class, the staff array had to be an array of Employee objects because some of its entries were regular employees. We would need to cast the managerial elements of the array back to Manager to access any of its new variables. (Note that in the sample code for the first section, we made a special effort to avoid the cast. We initialized the boss variable with a Manager object before storing it in the array.

GENERIC ARRAY LISTS

In many programming languages—in particular, in C—you have to fix the sizes of all arrays at compile time. Programmers hate this because it forces them into uncomfortable trade-offs.
In Java, user can set the size of an array at runtime.
int actualSize = . . .;
Employee[] staff = new Employee[actualSize];
this code does not completely solve the problem of dynamically modifying arrays at runtime. Once you set the array size, you cannot change it easily. Instead, the easiest way in Java to deal with this common situation is to use another Java class, called ArrayList. The ArrayList class is similar to an array, but it automatically adjusts its capacity as you add and remove elements, without your needing to write any code.
As of Java SE 5.0, ArrayList is a generic class with a type parameter. To specify the type of the element objects that the array list holds, you append a class name enclosed in angle brackets, such as ArrayList<Employee>. ArrayList<Employee> staff = new ArrayList<Employee>();

User use the add method to add new elements to an array list. For example, here is how you populate an array list with employee objects:

```
staff.add(new Employee("Harry Hacker", . . .));
staff.add(new Employee("Tony Tester", . . .));
```

The array list manages an internal array of object references. Eventually, that array will run out of space. This is where array lists work their magic: If you call add and the internal array is full, the array list automatically creates a bigger array and copies all the objects from the smaller to the bigger array. If you already know, or have a good guess, how many elements you want to store, then call the ensureCapacity method before filling the array list:

```
staff.ensureCapacity(100);
```

That call allocates an internal array of 100 objects. Then, the first 100 calls to add do not involve any costly reallocation.

You can also pass an initial capacity to the ArrayList constructor:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```

The size method returns the actual number of elements in the array list. For example,

```
staff.size()
```

returns the current number of elements in the staff array list. This is the equivalent of a.length for an array a.

Once you are reasonably sure that the array list is at its permanent size, you can call the trimToSize method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory. Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use trimToSize when you are sure you won't add any more elements to the array list.

Accessing Array List Elements

The automatic growth convenience that array lists give requires a more complicated syntax for accessing the elements. The reason is that the ArrayList class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.

Instead of using the pleasant [] syntax to access or change the element of an array, you use the get and set methods.

For example, to set the ith element, you use

```
staff.set(i, harry);
```

This is equivalent to

```
a[i] = harry;
```

for an array a. (As with arrays, the index values ar**e zero-based.)**

To get an array list element, use
Employee e = staff.get(i);
This is equivalent to
Employee e = a[i];
First, make an array list and add all the elements:
ArrayList<X> list = new ArrayList<X>();
while (. . .)
{
x = . . .;
list.add(x);
}
When you are done, use the toArray method to copy the elements into an array:
X[] a = new X[list.size()];
list.toArray(a);
Sometimes, you need to add elements in the middle of an array list. Use the add method
with an index parameter:
int n = staff.size() / 2;
staff.add(n, e);
The elements at locations n and above are shifted up to make room for the new entry. If the new size of the array list after the insertion exceeds the capacity, then the array list reallocates its storage array**.**
Similarly, you can remove an element from the middle of an array list:
Employee e = staff.remove(n);
The elements located above it are copied down, and the size of the array is reduced by one.
Inserting and removing elements is not terribly efficient. It is probably not worth worrying about for small array lists. But if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead.
As of Java SE 5.0, you can use the "for each" loop to traverse the contents of an array list:
for (Employee e : staff)
do something with e
This loop has the same effect as
for (int i = 0; i < staff.size(); i++)
{
Employee e = staff.get(i);
do something with e
}

OBJECT WRAPPERS
To convert a primitive type like int to an object. All primitive types

have class counterparts. For example, a class Integer corresponds to the primitive type int.
These kinds of classes are usually called wrappers. The wrapper classes have obvious names: Integer, Long, Float, Double, Short, Byte, Character, Void, and Boolean. The wrapper classes are immutable—user cannot change a wrapped value after the wrapper has been constructed. They are also final, so user cannot subclass them.
Suppose we want an array list of integers. Unfortunately, the type parameter inside the angle brackets cannot be a primitive type. It is not possible to form an ArrayList<int>.
Here, the Integer wrapper class comes in. It is ok to declare an array list of Integer objects.
ArrayList<Integer> list = new ArrayList<Integer>();
Another Java SE 5.0 innovation makes it easy to add and get array elements. The call
list.add(3); is automatically translated to
list.add(new Integer(3));
This conversion is called autoboxing.
Conversely, when you assign an Integer object to an int value, it is automatically unboxed. That is, the compiler translates
int n = list.get(i);
into
int n = list.get(i).intValue();
Automatic boxing and unboxing even works with arithmetic expressions. For example, user can apply the increment operator to a wrapper reference:
Integer n = 3;
n++;
The compiler automatically inserts instructions to unbox the object, increment the resulting value, and box it back. The designers of Java found the wrappers a convenient place to put certain basic methods, like the ones for converting strings of digits to numbers.
To convert a string to an integer, you use the following statement:
int x = Integer.parseInt(s);

INTERFACES

In the Java programming language, an interface is not a class but a set of requirements for classes that want to conform to the interface**.**
Here is what the Comparable interface looks like:

```
public interface Comparable
{
int compareTo(Object other);
}
```


This means that any class that implements the Comparable interface is required to have a compareTo method, and the method must take an Object parameter and return an integer.
NOTE: As of Java SE 5.0, the Comparable interface has been enhanced to be a generic type.

```
public interface Comparable<T>
{
int compareTo(T other); // parameter has type T
}
```

For example, a class that implements Comparable<Employee> must supply a method
int compareTo(Employee other)
You can still use the "raw" Comparable type without a type parameter, but then you have to manually cast the parameter of the compareTo method to the desired type.
All methods All methods of an interface are automatically public. For that reason, it is not necessary to supply the keyword public when declaring a method in an interface.
When calling x.compareTo(y), the compareTo method must actually be able to compare two objects and return an indication whether x or y is larger. The method is supposed to return a negative number if x is smaller than y, zero if they are equal, and a positive number
otherwise. This particular interface has a single method. Some interfaces have more than one method. As you will see later, interfaces can also define constants. What is more important,
however, is what interfaces cannot supply. Interfaces never have instance fields, and the methods are never implemented in the interface. Supplying instance fields and method implementations is the job of the classes that implement the interface.

To declare that a class implements an interface, use the implements keyword:

class Employee implements Comparable

As of Java SE 5.0, we can do a little better. We'll decide to implement the Comparable<Employee>
interface type instead.

```
class Employee implements Comparable<Employee>
{
public int compareTo(Employee other)
{
if (salary < other.salary) return -1;
if (salary > other.salary) return 1;
return 0;
}
. . .
```



```
}
```

The reason for interfaces is that the Java programming language is strongly typed. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the sort method will be statements like this:

```
if (a[i].compareTo(a[j]) > 0)
{
// rearrange a[i] and a[j]
. . .
}
```

The compiler must know that a[i] actually has a compareTo method. If a is an array of Comparable objects, then the existence of the method is assured because every class that implements the Comparable interface must supply the method.

PROPERTIES OF INTERFACES

Interfaces are not classes. In particular, you can never use the new operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can't construct interface objects, you can still declare interface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

Next, just as you use instanceof to check whether an object is of a specific class, you can use instanceof to check whether an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called Moveable.

```
public interface Moveable
{
void move(double x, double y);
}
```

Then, you could imagine an interface called Powered that extends it:

```
public interface Powered extends Moveable
{
double milesPerGallon();
}
```

Although you cannot put instance fields or static methods in an interface, user can supply constants in them. For example:

```
public interface Powered extends Moveable
{
double milesPerGallon();
double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically public, fields are always

public static final.

Some interfaces define just constants and no methods. For example, the standard library contains an interface SwingConstants that defines constants NORTH, SOUTH, HORIZONTAL,

and so on. Any class that chooses to implement the SwingConstants interface automatically inherits these constants. Its methods can simply refer to NORTH rather than the more cumbersome SwingConstants.NORTH. However, this use of interfaces seems rather degenerate, and we do not recommend it.

While each class can have only one superclass, classes can implement multiple interfaces.

UNIT-3
EVENT HANDLING

1.BASICS OF EVENT HANDLING

Event handling is of fundamental importance to programs with a graphical user interface.

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running. Each program then decides what, if anything, to do in response to these events.

The Java programming environment takes an approach somewhat between the Visual

Basic approach and the raw C approach in terms of power and, therefore, in resulting complexity. Within the limits of the events that the AWT knows about, user completely control how events are transmitted from the event sources (such as buttons or scrollbars)

to event listeners. You can designate any object to be an event listener—user pick an object that can conveniently carry out the desired response to the event. This event delegation model gives user much more flexibility than is possible with Visual Basic, in which the listener is predetermined. Event sources have methods that allow you

to register event listeners with them. When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event. As one would expect in an object-oriented language like Java, the information about the event is encapsulated in an event object. In Java, all event objects ultimately derive from the class java.util.EventObject. Of course, there are subclasses for each event type, such as ActionEvent and WindowEvent.

Different event sources can produce different kinds of events. For example, a button can send ActionEvent objects, whereas a window can send WindowEvent objects.

An overview of how event handling in the AWT works:

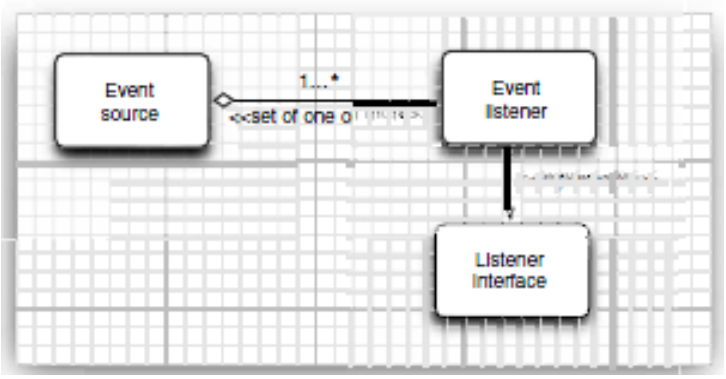
• A listener object is an instance of a class that implements a special interface called a listener interface.

• An event source is an object that can register listener objects and send them event objects.

• The event source sends out event objects to all registered listeners when that event occurs.

• The listener objects will then use the information in the event object to determinetheir reaction to the event.

Figure: shows the relationship between the event handling classes and interfaces.



Example for specifying a listener:

```
ActionListener listener = . . .;
JButton button = new JButton("Ok");
button.addActionListener(listener);
```

Now the listener object is notified whenever an “action event” occurs in the button. For buttons, as you might expect, an action event is a button click.

To implement the ActionListener interface, the listener class must have a method called actionPerformed that receives an ActionEvent object as a parameter.


```
class MyListener implements ActionListener
{
. . .
public void actionPerformed(ActionEvent event)
{
// reaction to button click goes here
. . .
}
}
```

Whenever the user clicks the button, the JButton object creates an ActionEvent object and calls listener.actionPerformed(event), passing that event object. An event source such as a button can have multiple listeners. In that case, the button calls the actionPerformed methods of all listeners whenever the user clicks the button.

NOTE: Write a Program on Push Buttons,When user press a button a particular color will be displayed.

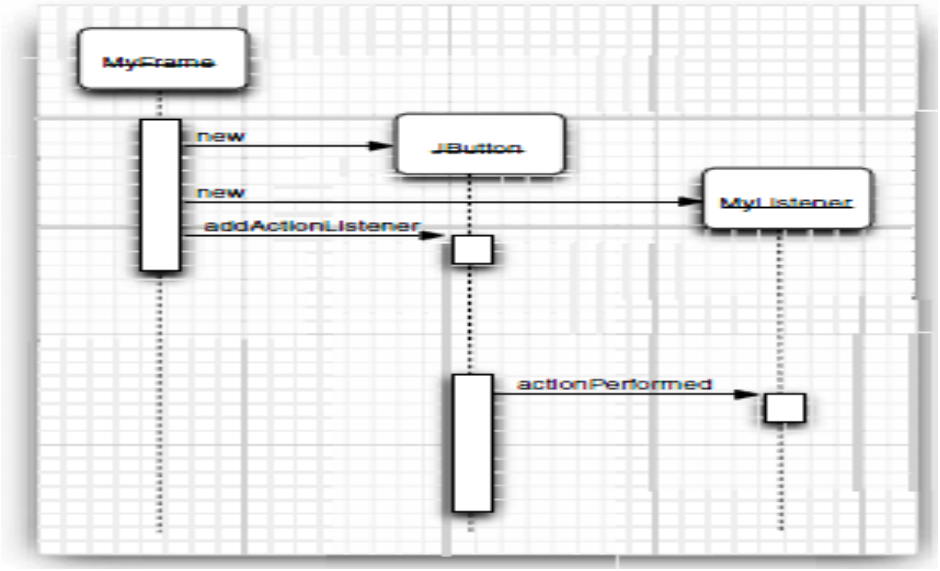


FIG : Event notification

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. 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( )
it is important to know only that all of the other classes 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.

Event Class	**Description**
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
WindowEvent	Generated when a window is activated, closed, deiconified, iconified, opened, or quit.

2.AWT EVENT HIERARCHY

The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes.

Some of the Swing components generate event objects of yet more event types; these directly extend EventObject, not AWTEvent.
The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that were passed to the listener object, as we did in the button example with the getSource and getActionCommand methods.
Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts PaintEvent objects into the event queue, but these objects are not delivered to listeners. Java programmers don’t listen to paint events; they override the paintComponent method to control repainting. The AWT also generates a number of events that are needed only by system programmers, to provide input systems for ideographic languages, automated testing robots, and so on.

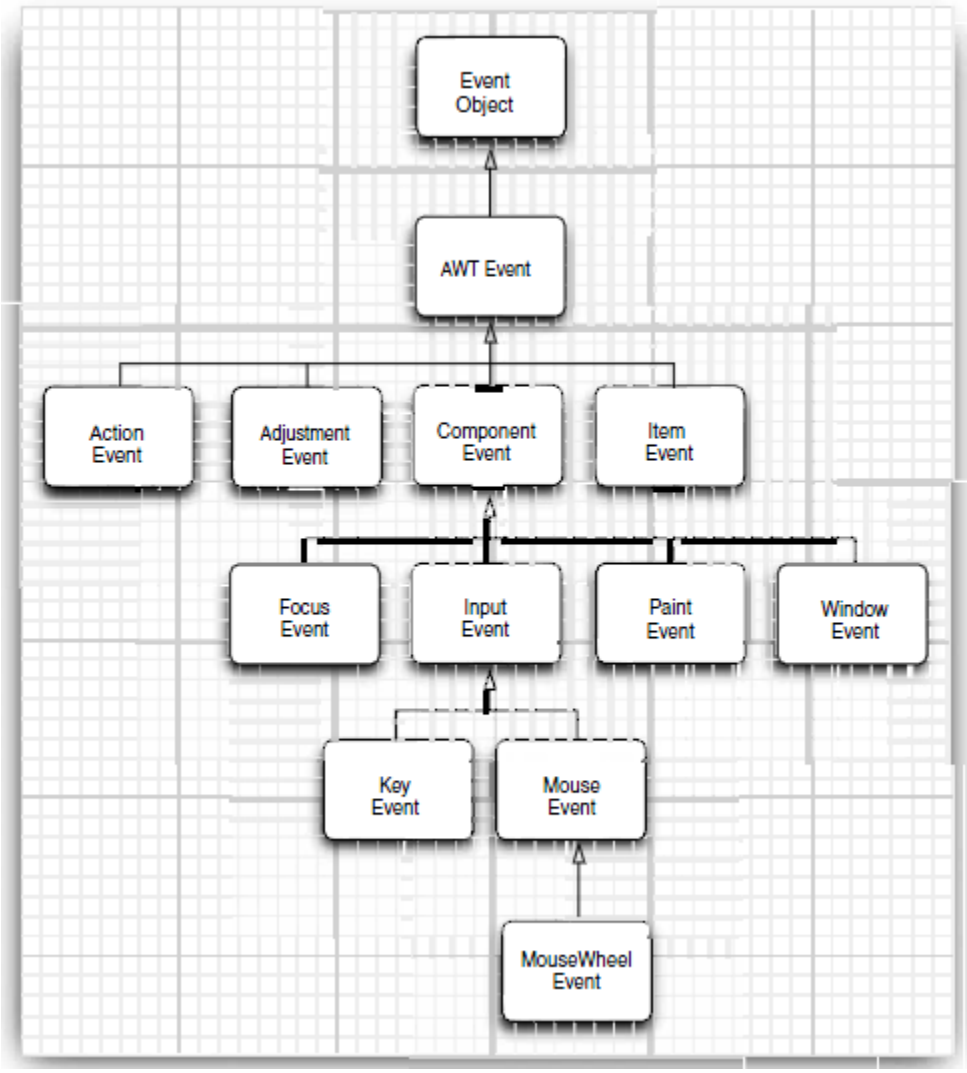


FIG : Inheritance diagram of AWT event classes

Semantic and Low-Level Events

The AWT makes a useful distinction between low-level and semantic events. A semantic event is one that expresses what the user is doing, such as "clicking that button"; hence, an Action-
Event is a semantic event. Low-level events are those events that make this possible. In the case of a button click, this is a mouse down, a series of mouse moves, and a mouse up (but only if the mouse up is inside the button area). Or it might be a keystroke, which happens if the user selects the button with the TAB key and then activates it with the space bar. Similarly, adjusting a scrollbar is a semantic event, but dragging the mouse is a low-level event.
The most commonly used semantic event classes in the java.awt.event package:

• ActionEvent (for a button click, a menu selection, selecting a list item, or ENTER typed in a text field)
• AdjustmentEvent (the user adjusted a scrollbar)
• ItemEvent (the user made a selection from a set of checkbox or list items)
Five low-level event classes are commonly used:
• KeyEvent (a key was pressed or released)
• MouseEvent (the mouse button was pressed, released, moved, or dragged)
• MouseWheelEvent (the mouse wheel was rotated)
• FocusEvent (a component got focus or lost focus)
• WindowEvent (the window state changed)
The following interfaces listen to these events:
ActionListener MouseMotionListener
AdjustmentListener MouseWheelListener
FocusListener WindowListener
ItemListener WindowFocusListener
KeyListener WindowStateListener
MouseListener
Several of the AWT listener interfaces, namely, those that have more than one method, come with a companion adapter class that implements all the methods in the interface to do nothing. (The other interfaces have only a single method each, so there is no benefit in having adapter classes for these interfaces.)
Here are the commonly used adapter classes:
FocusAdapter MouseMotionAdapter
KeyAdapter WindowAdapter
MouseAdapter

Event Handling Summary

Interface	Methods	Parameter/ Accessors	Events Generated By
ActionListener	actionPerformed	ActionEvent • getActionCommand • getModifiers	AbstractButton JComboBox JTextField Timer
AdjustmentListener	adjustmentValueChanged	AdjustmentEvent • getAdjustable • getAdjustmentType • getValue	JScrollbar

ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent	Component

3.USER INTERFACE COMPONENTS

The model-view-controller pattern is not the only pattern used in the design of AWT and Swing. Here are several additonal examples:

• Containers and components are examples of the "composite" pattern.

• The scroll pane is a "decorator."

• Layout managers follow the "strategy" pattern.

The Model-View-Controller Pattern

The procedure that make up a user interface component such as a button, a checkbox, a text field, or a sophisticated tree control.

Every component has three characteristics:

• Its content, such as the state of a button (pushed in or not), or the text in a text field

• Its visual appearance (color, size, and so on)

• Its behavior (reaction to events)

Even a seemingly simple component such as a button exhibits some moderately complex interaction among these characteristics. Obviously, the visual appearance of a button depends on the look and feel. A Metal button looks different from a Windows button or a Motif

button. In addition, the appearance depends on the button state: When a button is pushed in, it needs to be redrawn to look different. The state depends on the events that the button receives. When the user depresses the mouse inside the button, the button is pushed in.
To do this, the Swing designers turned to a well-known design pattern: the model-viewcontroller pattern. This pattern, like many other design patterns, goes back to one of the principles of object-oriented design make one object responsible for too much. Don't have a single button class do everything. Instead, have the look and feel of the component associated with one object and store the content in another object. The model-view-controller (MVC) design pattern Implement three separate classes:

- The model, which stores the content
- The view, which displays the content
- The controller, which handles user input

The pattern specifies precisely how these three objects interact. The model stores the content and has no user interface. For a button, the content is pretty trivial—just a small set of flags that tells whether the button is currently pushed in or out, whether it is active or
inactive, and so on. For a text field, the content is a bit more interesting. It is a string object that holds the current text. This is not the same as the view of the content—if the content is larger than the text field, the user sees only a portion of the text displayed.

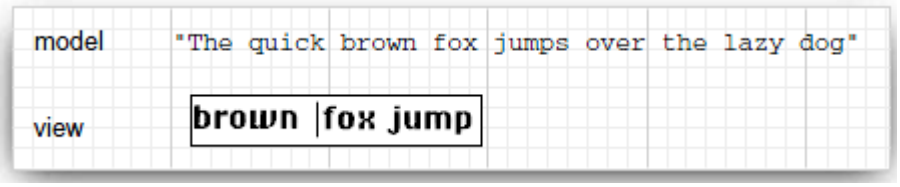


FIG :MODEL AND VIEW OF A TEXT FIELD

The model must implement methods to change the content and to discover what the content is. For example, a text model has methods to add or remove characters in the current text and to return the current text as a string. The model is completely nonvisual. It is the job of a view to draw the data that is stored in the model.
The controller handles the user-input events such as mouse clicks and keystrokes. It then decides whether to translate these events into changes in the model or the view.
For example, if the user presses a character key in a text box, the controller calls the "insert character" command of the model. The model then tells the view to update itself. The view never knows why the text changed. But if the user presses a cursor key, then

the controller may tell the view to scroll. Scrolling the view has no effect on the underlyingtext, so the model never knows that this event happened.

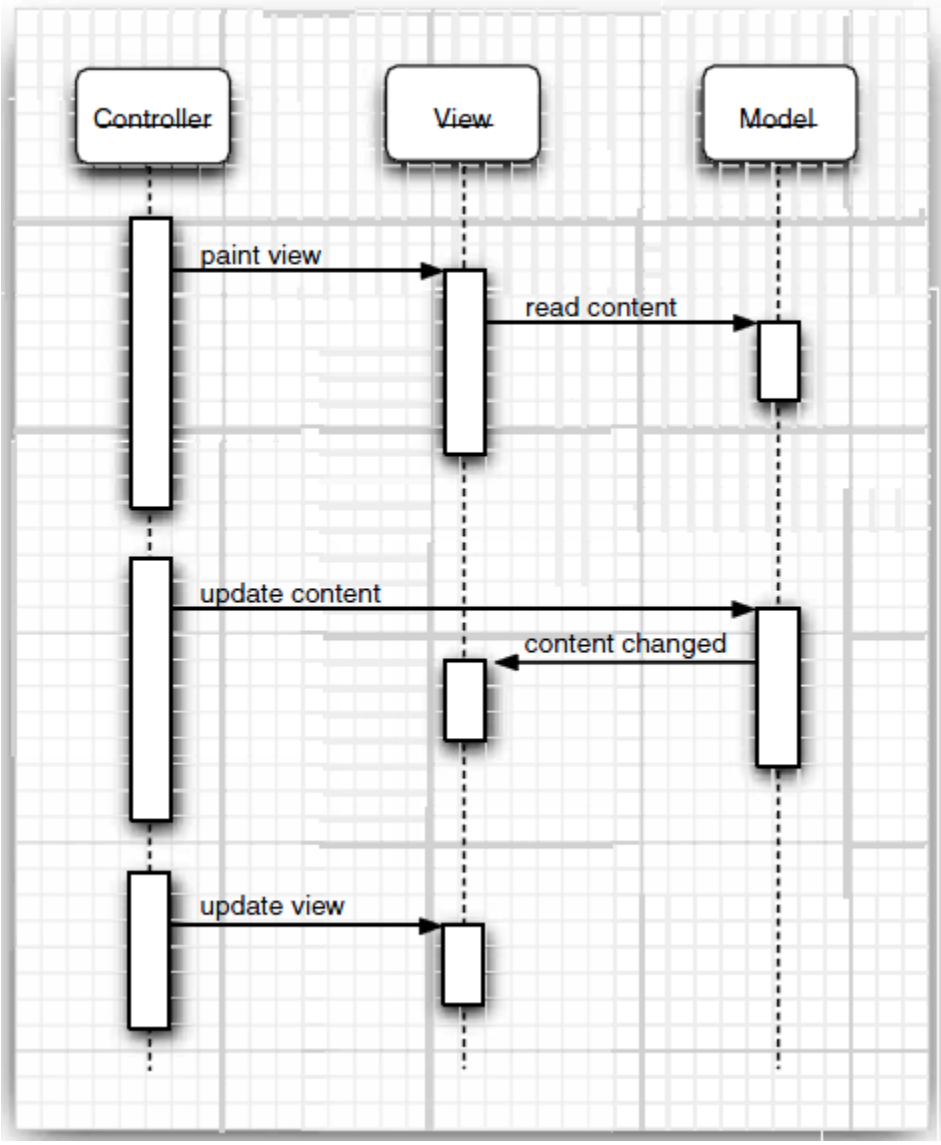


FIG : Interactions among model, view, and controller objects

4.INTRODUCTION TO LAYOUT MANAGEMENT

The buttons are contained in a JPanel object and are managed by the flow layout manager, the default layout manager for a panel,when user add more buttons to the panel. a new row is started when there is no more room,from the below fig.

Moreover, the buttons stay centered in the panel, even when the user resizes the frame.

Changing the panel size rearranges the buttons automatically

In general, components are placed inside containers, and a layout manager determines the positions and sizes of the components in the container. Buttons, text fields, and other user interface elements extend the class Component. Components can be placed inside containers such as panels. Because containers can themselves be put inside other containers, the class Container extends Component. Below Fig shows the inheritance hierarchy for Component.

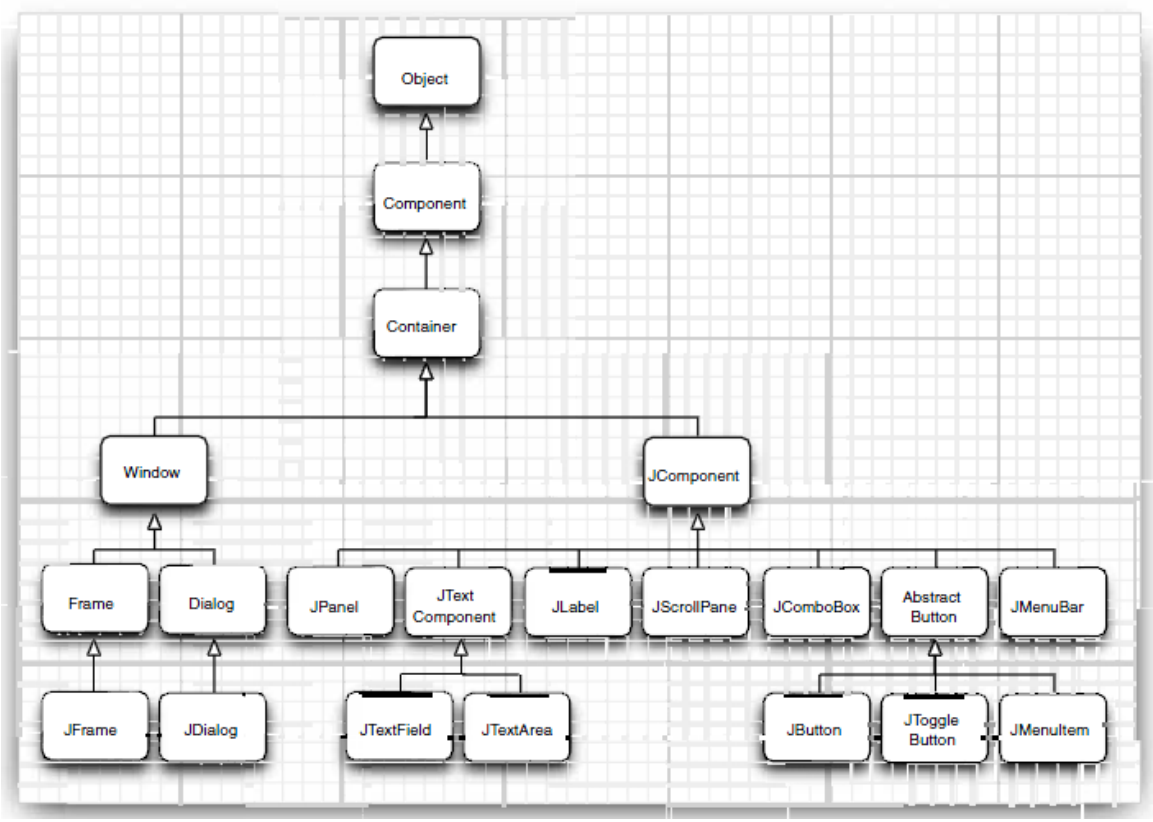


FIG : Inheritance hierarchy for the Component class

Each container has a default layout manager, but user can always set user's own. For example, the statement

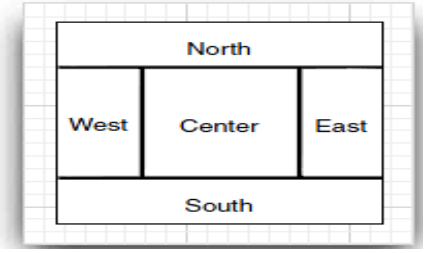
panel.setLayout(new GridLayout(4, 4));

uses the GridLayout class to lay out the components in the panel. You add components to the container. The add method of the container passes the component and any placement directions to the layout manager.

Border Layout

The border layout manager is the default layout manager of the content pane of every JFrame. Unlike the flow layout manager, which completely controls the position of each component, the border layout manager lets user choose where user want to place each component. User can choose to place the component in the center, north, south, east, or west of the content pane

Figure Border layout



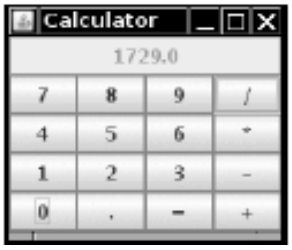
For example:
frame.add(component, BorderLayout.SOUTH);
The edge components are laid out first, and the remaining available space is occupied by the center. When the container is resized, the dimensions of the edge components are unchanged, but the center component changes its size. User add components by specifying
a constant CENTER, NORTH, SOUTH, EAST, or WEST of the BorderLayout class. Not all of the positions need to be occupied. If User don't supply any value, CENTER is assumed.
Unlike the flow layout, the border layout grows all components to fill the available space. (The flow layout leaves each component at its preferred size.) This is a problem when user add a button:
frame.add(yellowButton, BorderLayout.SOUTH); // don't

FIG : **A single button managed by a border layout**

FIG : Panel placed at the southern region of the frame

GRID LAYOUT:
The grid layout arranges all components in rows and columns like a spreadsheet. All components are given the same size. The calculator program uses a grid layout to arrange the calculator buttons. When you resize the window, the buttons grow and shrink, but all buttons have identical sizes.
In the constructor of the grid layout object, you specify how many rows and columns user need.
panel.setLayout(new GridLayout(5, 4));
user add the components, starting with the first entry in the first row, then the second entry in the first row, and so on.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```



5 .Text Input

User can use the JTextField and JText- Area components for gathering text input. A text field can accept only one line of text; a text area can accept multiple lines of text. A JPasswordField accepts one line of text without showing the contents.

All three of these classes inherit from a class called JTextComponent. User will not be able to construct a JTextComponent yourself because it is an abstract class. On the other hand, as is so often the case in Java, when you go searching through the API documentation, you may find that the methods you are looking for are actually in the parent class JTextComponent rather than in the derived class. For example, the methods that get or set the text in a text field or text area are actually methods in JTextComponent.

Text Fields

The usual way to add a text field to a window is to add it to a panel or other container—

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

This code adds a text field and initializes the text field by placing the string "Default input" inside it. The second parameter of this constructor sets the width.

6. CHOICE COMPONENTS

Checkboxes

If user want to collect just a “yes” or “no” input, use a checkbox component. Checkboxes automatically come with labels that identify them. The user usually checks the box by clicking inside it and turns off the check mark by clicking inside the box again. To toggle the check mark, the user can also press the space bar when the focus is in the checkbox.

Below Figure shows a simple program with two checkboxes, one to turn on or off the italic attribute of a font, and the other for boldface. Note that the second checkbox has focus, as indicated by the rectangle

around the label. Each time the user clicks one of the checkboxes, the screen is refreshed, using the new font attributes.

Figure:Checkboxes

Checkboxes need a label next to them to identify their purpose. You give the label text in

bold = new JCheckBox("Bold");

You use the setSelected method to turn a checkbox on or off. For example:

bold.setSelected(true);

The isSelected method then retrieves the current state of each checkbox. It is false if unchecked; true if checked.

When the user clicks on a checkbox, this triggers an action event. As always, you attach an action listener to the checkbox. In our program, the two checkboxes share the same action listener.

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

The actionPerformed method queries the state of the bold and italic checkboxes and sets the font of the panel to plain, bold, italic, or both bold and italic.

```
public void actionPerformed(ActionEvent event)
{
int mode = 0;
if (bold.isSelected()) mode += Font.BOLD;
if (italic.isSelected()) mode += Font.ITALIC;
label.setFont(new Font("Serif", mode, FONTSIZE));
}
```

NOTE : WRITE A PROGRAM TO DEMONSTRATE THE CHECK BOXES

RADIO BUTTONS

Implementing radio button groups is easy in Swing. User construct one object of type ButtonGroup for every group of buttons. Then, User

add objects of type JRadioButton to the button group. The button group object is responsible for turning off the previously set
button when a new button is clicked.

```
ButtonGroup group = new ButtonGroup();
JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);
JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

The second argument of the constructor is true for the button that should be checked initially and false for all others. Note that the button group controls only the behavior of the buttons; if you want to group the buttons for layout purposes, you also need to add them to a container such as a JPanel.

The event notification mechanism for radio buttons is the same as for any other buttons.When the user checks a radio button, the radio button generates an action event. In example an action listener that sets the font size to a particular value:

```
ActionListener listener = new
ActionListener()
{
  public void actionPerformed(ActionEvent event)
{
// size refers to the final parameter of the addRadioButton method
label.setFont(new Font("Serif", Font.PLAIN, size));
}
};
```

Each radio button gets a different listener object. Each listener object knows exactly what it needs to do—set the font size to a particular value. In the case of the checkboxes, we used a different approach. Both checkboxes have the same action listener. It called a method that looked at the current state of both checkboxes.

Borders

If user have multiple groups of radio buttons in a window, you will want to visually indicate which buttons are grouped. Swing provides a set of useful borders for this purpose.
User can apply a border to any component that extends JComponent. The most common usage is to place a border around a panel and fill that panel with other user interface elements such as radio buttons.
User can choose from quite a few borders, but User follow the same steps for all of them.
1. Call a static method of the BorderFactory to create a border. You can choose among the
following styles from below fig
• Lowered bevel
• Raised bevel
• Etched
• Line
• Matte
• Empty (just to create some blank space around the component)
2. If user like, add a title to your border by passing your border to BorderFactory.createTitledBorder.
3. If user really want to go all out, combine several borders with a call to BorderFactory.createCompoundBorder.
4. Add the resulting border to users component by calling the setBorder method of the JComponent class.
For example, here is how you add an etched border with a title to a panel:

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

FIG :TESTING BORDER TYPES

COMBO BOXES

If user have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead,

user can use a combo box. When the user clicks on the component, a list of choices drops down, and the user can then select one of them.

FIG : A COMBO BOX

If the drop-down list box is set to be editable, then user can edit the current selection as if it were a text field. For that reason, this component is called a combo box—it combines the flexibility of a text field with a set of predefined choices. The JComboBox class provides a combo box component. user call the setEditable method to make the combo box editable. Note that editing affects only the current item. It does not change the content of the list.

User can obtain the current selection or edited text by calling the getSelectedItem method.

For example, the user can choose a font style from a list of styles (Serif, Sans-Serif, Monospaced, etc.). The user can also type in another font.

User add the choice items with the addItem method. In our program, addItem is called only in the constructor, but User can call it any time.

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

This method adds the string at the end of the list. You can add new items anywhere in the list with the insertItemAt method:

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

User can add items of any type—the combo box invokes each item's toString method to display it.

If user need to remove items at runtime, you use the removeItem or removeItemAt method, depending on whether you supply the item to be removed or its position.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

The removeAllItems method removes all items at once.

When the user selects an item from a combo box, the combo box generates an action event. To find out which item was selected, call getSource on the event parameter to get a reference to the combo box that sent the event. Then call the getSelectedItem method to retrieve the currently selected item. You need to cast the returned value to the appropriate type, usually String.

```
public void actionPerformed(ActionEvent event)
{
label.setFont(new Font(
(String) faceCombo.getSelectedItem(),
Font.PLAIN,
DEFAULT_SIZE));
}
```

7.MENUS

A menu bar on top of the window contains the names of the pull-down menus. Clicking on a name opens the menu containing menu items and submenus. When the user clicks on a menu item, all menus are closed and a message is sent to the program. Figure shows a typical menu with a submenu.

Menu Building

Building menus is straightforward. You first create a menu bar:

```
JMenuBar menuBar = new JMenuBar();
```

A menu bar is just a component that you can add anywhere you like. Normally, user want it to appear at the top of a frame. You can add it there with the setJMenuBar method:

```
frame.setJMenuBar(menuBar);
```

For each menu, user create a menu object:

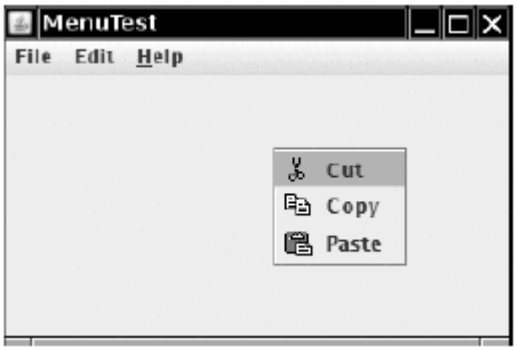
JMenu editMenu = new JMenu("Edit");
You add the top-level menus to the menu bar:
menuBar.add(editMenu);
You add menu items, separators, and submenus to the menu object:
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
You can see separators in Figure 9–19 below the "Paste" and "Read-only" menu items.
When the user selects a menu, an action event is triggered. You need to install an action
listener for each menu item:
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
The method JMenu.add(String s) conveniently adds a menu item to the end of a menu. For example:
editMenu.add("Paste");
The add method returns the created menu item, so you can capture it and then add the listener, as follows:
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
It often happens that menu items trigger commands that can also be activated through other user interface elements such as toolbar buttons. specify commands through Action objects. You define a class that implements the Action interface, usually by extending the AbstractAction convenience class. You specify the menu item label in the constructor of the AbstractAction object, and you override the
actionPerformed method to hold the menu action handler. For example:
Action exitAction = new AbstractAction("Exit") // menu item text goes here
{
public void actionPerformed(ActionEvent event)
{
// action code goes here
System.exit(0);
}
};
You can then add the action to the menu:
JMenuItem exitItem = fileMenu.add(exitAction);
This command adds a menu item to the menu, using the action name. The action object

becomes its listener. This is just a convenient shortcut for
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
set it with the setIcon method that the JMenuItem class inherits from the AbstractButton class.
Here is an example:
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
By default, the menu item text is placed to the right of the icon. If you prefer the text to be placed on the
left, call the setHorizontalTextPosition method that the JMenuItem class inherits from the AbstractButton class. For example, the call
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
moves the menu item text to the left of the icon.
You can also add an icon to an action:
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
Whenever you construct a menu item out of an action, the Action.NAME value becomes the
text of the menu item and the Action.SMALL_ICON value becomes the icon. Alternatively, you can set the icon in the AbstractAction constructor:
cutAction = new
AbstractAction("Cut", new ImageIcon("cut.gif"))
{
public void actionPerformed(ActionEvent event)
{
// action code goes here
}
};
Apart from the button decoration, you treat these menu items just as you would any others. For example, here is how you create a checkbox menu item:
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
The radio button menu items work just like regular radio buttons. You must add them to a button group. When one of the buttons in a group is selected, all others are automatically deselected.
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");

group.add(insertItem);
group.add(overtypeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypeItem);
With these menu items, you don't necessarily want to be notified at the exact moment the user selects the item. Instead, you can simply use the isSelected method to test the current state of the menu item. Use the setSelected method to set the state.

Pop-Up Menus

A pop-up menu is a menu that is not attached to a menu bar but that floats somewhere



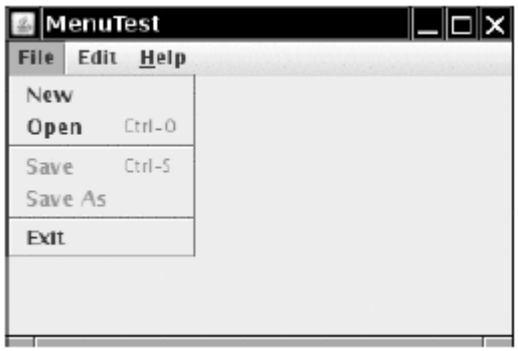
A pop-up menu

User create a pop-up menu similarly to the way you create a regular menu, but a pop-up menu has no title.
JPopupMenu popup = new JPopupMenu();
User then add menu items in the usual way:

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

Unlike the regular menu bar that is always shown at the top of the frame, user must explicitly display a pop-up menu by using the show method. User specify the parent component
and the location of the pop-up, using the coordinate system of the parent. For example:
popup.show(panel, x, y);
Usually user write code to pop up a menu when the user clicks a particular mouse button, the so-called pop-up trigger. In Windows and Linux, the pop-up trigger is the nonprimary (usually, the right) mouse button. To pop up a menu when the user clicks on a component, using the pop-up trigger, simply call the method
component.setComponentPopupMenu(popup);
Very occasionally, user may place a component inside another component that has a pop-up menu. The child component can inherit the parent component's pop-up menu by calling
child.setInheritsPopupMenu(true);

These methods were added in Java SE 5.0 to insulate programmers from system dependencies with pop-up menus. Before Java SE 5.0, user had to install a mouse
listener and add the following code to both the mousePressed and the mouseReleased listener methods:
if (popup.isPopupTrigger(event))
popup.show(event.getComponent(), event.getX(), event.getY());
Some systems trigger pop-ups when the mouse button goes down, others when the mouse button goes up.
To enable or disable a menu item, use the setEnabled method:
saveItem.setEnabled(false);
There are two strategies for enabling and disabling menu items. Each time circumstances change, user can call setEnabled on the relevant menu items or actions. For example, as soon as a document has been set to read-only mode, user can locate the Save and Save As menu items and disable them.
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)



Disabled menu items

8.DIALOG BOXES

Most windowing systems, AWT distinguishes between modal and modeless dialog boxes. A modal dialog box won't let users interact with the remaining windows of the application until he or she deals with it. You use a modal dialog box when user need information from the user before user can proceed with execution. For example,
when the user wants to read a file, a modal file dialog box is the one to pop up. The user must specify a file name before the program can begin the read operation. Only when the user closes the (modal) dialog box can the application proceed.
A modeless dialog box lets the user enter information in both the dialog box and the remainder of the application. One example of a modeless dialog is a toolbar. The toolbar

can stay in place as long as needed, and the user can interact with both the application window and the toolbar as needed.
Swing has a convenient JOptionPane class that lets user put up a simple dialog without writing any special dialog box code.
File dialogs are complex, and you definitely want to be familiar with the Swing JFileChooser for this purpose—it would be a real challenge to write your own. The JColorChooser dialog is useful when you want users to pick colors.

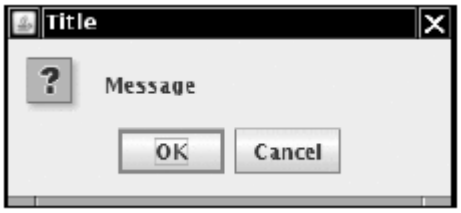
Option Dialogs

Swing has a set of ready-made simple dialogs that suffice when you need to ask the user for a single piece of information. The JOptionPane has four static methods to show these

simple dialogs:

showMessageDialog Show a message and wait for the user to click OK
showConfirmDialog Show a message and get a confirmation (like OK/Cancel)
showOptionDialog Show a message and get a user option from a set of options
showInputDialog Show a message and get one line of user input
Figure shows a typical dialog.
the dialog has the following
components:

- An icon
- A message
- One or more option buttons



An option dialog

The input dialog has an additional component for user input. This can be a text field into which the user can type an arbitrary string, or a combo box from which the user can select one item. The exact layout of these dialogs, and the choice of icons for standard message types, depend on the pluggable look and feel.
The icon on the left side depends on one of five message types:
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE

The PLAIN_MESSAGE type has no icon. Each dialog type also has a method that lets user supply his own icon instead.
For each dialog type, user can specify a message. This message can be a string, an icon, a user interface component, or any other object. Here is how the message object is displayed:
String Draw the string
Icon Show the icon
Component Show the component
Object[] Show all objects in the array, stacked on top of each other
Any other object Apply toString and show the resulting string
supplying a message string is by far the most common case. Supplying a Component gives you ultimate flexibility because you can make the paintComponent method draw anything you want.
The buttons on the bottom depend on the dialog type and the option type. When calling showMessageDialog and showInputDialog, user get only a standard set of buttons (OK and OK/ Cancel, respectively). When calling showConfirmDialog, you can choose among four option types:
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
With the showOptionDialog you can specify an arbitrary set of options. You supply an array of objects for the options. Each array element is rendered as follows:
String Make a button with the string as label
Icon Make a button with the icon as label
Component Show the component
Any other object Apply toString and make a button with the resulting string as label
The return values of these functions are as follows:
showMessageDialog None
showConfirmDialog An integer representing the chosen option
showOptionDialog An integer representing the chosen option
showInputDialog The string that the user supplied or selected
The showConfirmDialog and showOptionDialog return integers to indicate which button the user chose. For the option dialog, this is simply the index of the chosen option or the value CLOSED_OPTION if the user closed the dialog instead of choosing an option. For the confirmation dialog, the return value can be one of the following:
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION

CLOSED_OPTION
This all sounds like a bewildering set of choices, but in practice it is simple. Follow these steps:
1. Choose the dialog type (message, confirmation, option, or input).
2. Choose the icon (error, information, warning, question, none, or custom).
3. Choose the message (string, icon, custom component, or a stack of them).
4. For a confirmation dialog, choose the option type (default, Yes/No, Yes/No/ Cancel, or OK/Cancel).
5. For an option dialog, choose the options (strings, icons, or custom components) and the default option.
6. For an input dialog, choose between a text field and a combo box.
7. Locate the appropriate method to call in the JOptionPane API.

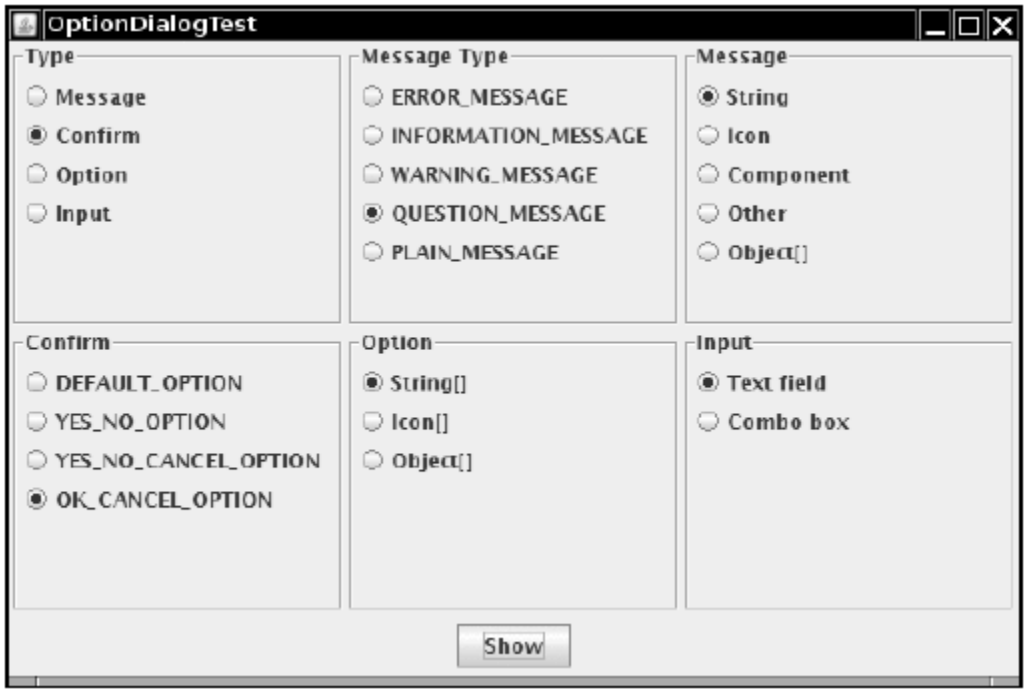


FIG :The OptionDialogTest program

APPLETS:
Applets are Java programs that are included in an HTML page. The HTML page must tell the browser which applets to load and then where to put each applet on the web page.The tag needed to use an applet must tell the browser where to get the class files, and how the applet is positioned on the web page (size, location,and so on). The browser then retrieves the class files from the Internet (or from a directory on the user's machine) and automatically runs the applet.

When applets were first developed,user had to use Sun's HotJava browser to view web pages that contained applets. Naturally, few users were willing to use a separate browser just to enjoy a new web feature. Java applets became really popular when Netscape included a Java virtual machine in its Navigator browser. Microsoft Internet Explorer soon followed suit. Unfortunately, two problems happened.
Netscape didn't keep up with more modern versions of Java, and Microsoft vacillated between reluctantly supporting outdated Java versions and dropping Java support altogether.
To overcome this problem, Sun released a tool called the "Java Plug-in." Using the various extension mechanisms, it seamlessly plugs in to a variety of browsers and enables them to execute Java applets by using an external Java runtime environment that Sun supplies. By keeping the Plug-in up-to-date, you can always take advantage of the latest and greatest features of Java.

A Simple Applet
An applet is simply a Java class that extends the java.applet.Applet class.

```
import java.awt.*;
import javax.swing.*;
public class NotHelloWorldApplet extends JApplet
{
public void init()
. {
. EventQueue.invokeLater(new Runnable()
{
public void run()
{
JLabel label = new JLabel("Not a Hello, World applet", SwingConstants.CENTER);
add(label);
}
});
}
}
```

To execute the applet, you carry out two steps:
1. Compile your Java source files into class files.
2. Create an HTML file that tells the browser which class file to load first and how to size the applet.

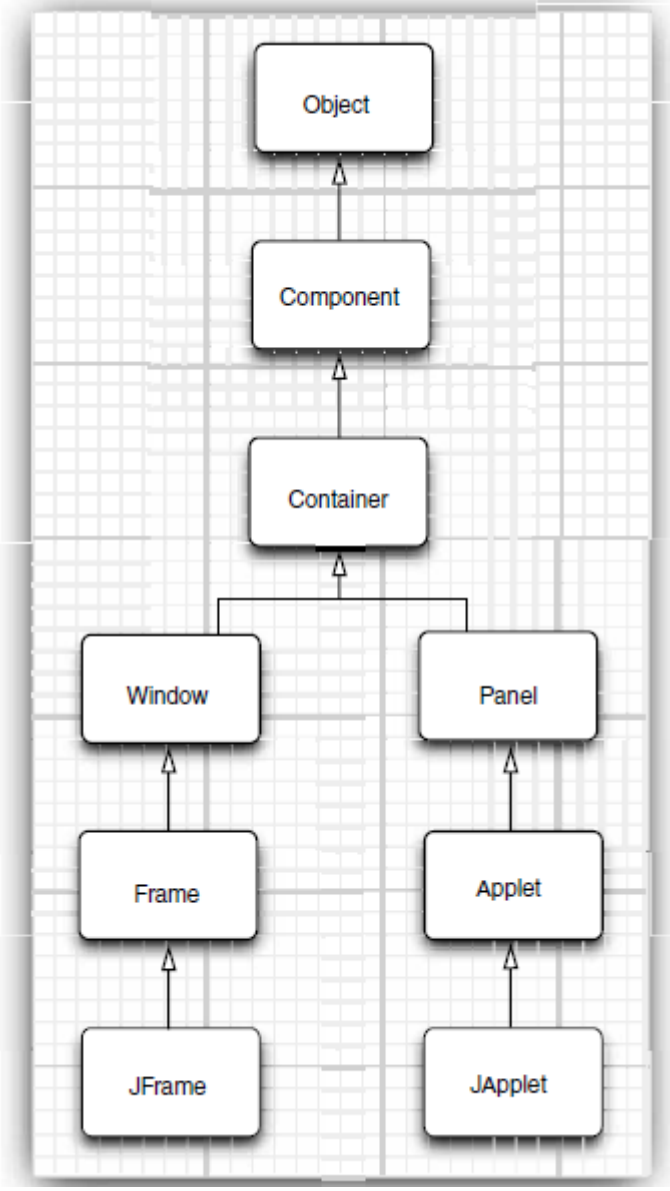


FIG : Applet inheritance diagram

It is customary (but not necessary) to give the HTML file the same name as that of the applet class inside. So, following this tradition, we call the file NotHelloWorldApplet.html.

Here are the contents of the file:

<applet code="NotHelloWorldApplet.class" width="300" height="300"> </applet>

The applet viewer is good for the first stage of testing, but at some point you need to run your applets in a browser to see them in the same way a user might use them. In particular, the applet viewer program shows you only the applet, not the surrounding
HTML text. If an HTML file contains multiple applet tags, the applet viewer pops up multiple windows.

The Applet HTML Tag and Its Attributes

In its most basic form, an example for using the applet tag looks like this:

<applet code="NotHelloWorldApplet.class" width="300" height="100">

the code attribute gives the name of the class file and must include the .class extension; the width and height attributes size the window that will hold the applet.

Both are measured in pixels. You also need a matching </applet> tag that marks the end of the HTML tagging needed for an applet. The text between the <applet> and </applet> tags is displayed only if the browser cannot show applets. The code, width, and height
attributes are required. If any are missing, the browser cannot load users applet.

All this information would usually be embedded in an HTML page that, at the very least, might look like this:

```
<html>
<head>
<title>NotHelloWorldApplet</title>
</head>
<body>
<p>The next line of text is displayed under the auspices of Java:</p>
<applet code="NotHelloWorldApplet.class" width="100" height="100">
If your browser could show Java, you would see an applet here.
</applet>
</body>
</html>
```

user can use the following attributes within the applet tag:

- width, height

These attributes are required and give the width and height of the applet, measured in pixels. In the applet viewer, this is the initial size of the applet.User can resize any window that the applet viewer creates. In a browser, user cannot resize the applet.

user will need to make a good guess about how much space your applet requires to show up well for all users.

- align

The attribute values are the same as for the align attribute of the HTML img tag.

• vspace, hspace

These optional attributes specify the number of pixels above and below the applet (vspace) and on each side of the applet (hspace).

• code

This attribute gives the name of the applet's class file. This name is taken relative to the codebase (see below) or relative to the current page if the codebase is not specified.

The path name must match the package of the applet class. For example, if the applet

class is in the package com.mycompany, then the attribute is code="com/mycompany/MyApplet.class".

The alternative code="com.mycompany.MyApplet.class" is also permitted. But you cannot use

absolute path names here. Use the codebase attribute if your class file is located elsewhere. The code attribute specifies only the name of the class that contains the applet class.

Once the browser's class loader loads the class containing the applet, it will realize that it needs more class files and will load them.

Either the code or the object attribute is required.

• codebase

This optional attribute is a URL for locating the class files. You can use an absolute URL, even to a different server. Most commonly, though, this is a relative URL that points to a subdirectory. For example, if the file layout looks like this: then you use the following tag in MyPage.html:

<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">

• archive

This optional attribute lists the JAR file or files containing classes and otherresources for the applet. These files are fetched from the web server before the applet is loaded. This technique speeds up the loading process significantly because only one HTTP request is necessary to load a JAR file that contains many smaller

files. The JAR files are separated by commas. For example:

<applet code="MyApplet.class"
archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
width="100" height="150">

LIFE CYCLE OF AN APPLET

The lifecycle methods of an Applet:
init(): This method is called to initialized an applet
start(): This method is called after the initialization of the applet.
stop(): This method can be called multiple times in the life cycle of an Applet.
destroy(): This method is called only once in the life cycle of the applet when applet is destroyed.

init () method: The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method. The init() method is called only one time in the life cycle on an applet. The init() method is basically called to read the PARAM tag in the html file. The init () method retrieve the passed parameter through the PARAM tag of html file using get Parameter() method All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method .After the initialization of the init() method user can interact with the Applet and mostly applet contains the init() method.

Start () method: The start method of an applet is called after the initialization method init(). This method may be called multiples time when the Applet needs to be started or restarted. For Example if the user wants to return to the Applet, in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.

Stop () method: The stop() method can be called multiple times in the life cycle of applet like the start () method. Or should be called at least one time. There is only miner difference between the start() method and stop () method. For example the stop() method is called by the web browser on that time When the user leaves one applet to go another applet and the start() method is called on that time when the user wants to go back into the first program or Applet.

destroy() method: The destroy() method is called only one time in the life cycle of Applet like init() method. This method is called only on that time when the browser needs to Shut down.

THREADS :

a thread is a program's path of execution. Most programs written today run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this.

Multithreaded applications deliver their potent power by running many threads concurrently within a single program. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time, however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system is treating the programs as two separate and distinct processes. Under Unix, forking a process creates a child process with a different address space for both code and data. However, fork() creates a lot of overhead for the operating system, making it a very CPU-intensive operation. By starting a thread instead, an efficient path of execution is created while still sharing the original data area from the parent

Creating threads

Java's creators have graciously designed two ways of creating threads: implementing an interface and extending a class. Extending a class is the way Java inherits methods and variables from a parent class. In this case, one can only extend or inherit from a single parent class. This limitation within Java can be overcome by implementing interfaces, which is the most common way to create threads. (Note that the act of inheriting merely allows the class to be run as a thread. It is up to the class to start() execution, etc.)

Interfaces provide a way for programmers to lay the groundwork of a class. They are used to design the requirements for a set of classes to implement. The interface sets everything up, and the class or classes that implement the interface do all the work. The different set of classes that implement the interface have to follow the same rules.

There are a few differences between a class and an interface. First, an interface can only contain abstract methods and/or static final variables (constants). Classes, on the other hand, can implement methods and contain variables that are not constants. Second, an interface cannot implement any methods. A class that implements an

interface must implement all methods defined in that interface. An interface has the ability to extend from other interfaces, and (unlike classes) can extend from multiple interfaces. Furthermore, an interface cannot be instantiated with the new operator; for example, Runnable a=new Runnable(); is not allowed.

The first method of creating a thread is to simply extend from the Thread class. Do this only if the class you need executed as a thread does not ever need to be extended from another class. The Thread class is defined in the package java.lang, which needs to be imported so that our classes are aware of its definition.

```
import java.lang.*;
public class Counter extends Thread
{
	public void run()
	{
	....
	}
}
```

The above example creates a new class Counter that extends the Thread class and overrides the Thread.run() method for its own implementation. The run() method is where all the work of the Counter class thread is done. The same class can be created by implementing Runnable:

```
import java.lang.*;
public class Counter implements Runnable
{
	Thread T;
	public void run()
	{
	....
	}
}
```

Here, the abstract run() method is defined in the Runnable interface and is being implemented. Note that we have an instance of the Thread class as a variable of the Counter class. The only difference between the two methods is that by implementing Runnable, there is greater flexibility in the creation of the class Counter. In the above example, the opportunity still exists to extend the Counter class, if needed. The majority of classes created that need to be run as a thread will

implement Runnable since they probably are extending some other functionality from another class.

Do not think that the Runnable interface is doing any real work when the thread is being executed. It is merely a class created to give an idea on the design of the Thread class. In fact, it is very small containing only one abstract method. Here is the definition of the Runnable interface directly from the Java source:

```
package java.lang;
public interface Runnable {
	public abstract void run();
}
```

That is all there is to the Runnable interface. An interface only provides a design upon which classes should be implemented. In the case of the Runnable interface, it forces the definition of only the run() method. Therefore, most of the work is done in the Thread class. A closer look at a section in the definition of the Thread class will give an idea of what is really going on:

```
public class Thread implements Runnable {
...
	public void run() {
		if (target != null) {
			target.run();
		}
	}
...
}
```

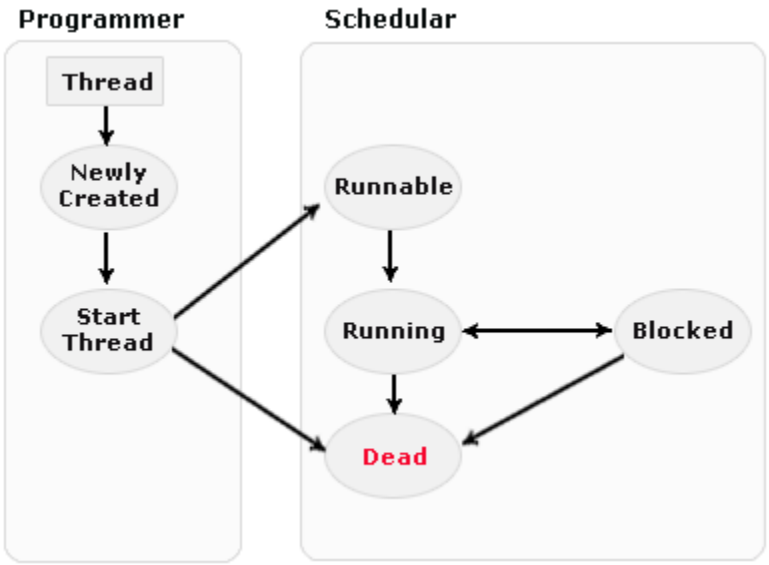
From the above code snippet it is evident that the Thread class also implements the Runnable interface. Thread.run() checks to make sure that the target class (the class that is going to be run as a thread) is not equal to null, and then executes the run() method of the target. When this happens, the run() method of the target will be running as its own thread.

LIFE CYCLE OF A THREAD

When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking start() method, it doesn't mean that the thread has

access to CPU and start executing straight away. Several factors determine how it will proceed.

Different states of a thread are :



New state – After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

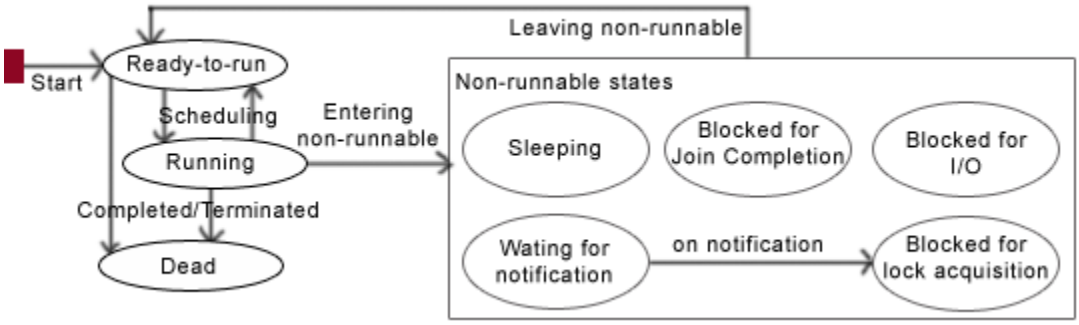
Runnable (Ready-to-run) state – A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

Running state – A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

Dead state – A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.

Blocked - A thread can enter in this state because of waiting the resources that are hold by another thread.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

Sleeping – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method sleep( ) to stop the running state of a thread.

static void sleep(long millisecond) throws InterruptedException

Waiting for Notification – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

final void wait(long timeout) throws InterruptedException

final void wait(long timeout, int nanos) throws InterruptedException

final void wait() throws InterruptedException

Blocked on I/O – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.

Blocked for joint completion – The thread can come on this state because of waiting the completion of another thread.

Blocked for lock acquisition – The thread can come on this state because of waiting to acquire the lock of an object.

MULTITHREADING

Multitasking :

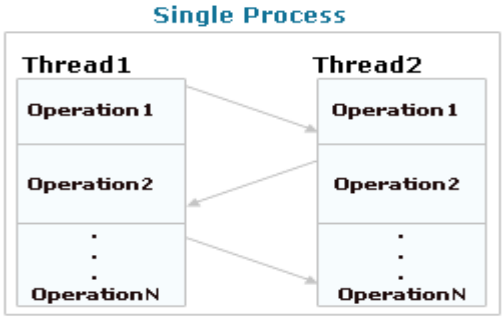
Multitasking allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single OS.

Multithreading :

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, When you use a word processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.

In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

For example, look at the diagram shown as:



In the above diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

Advantages of multithreading over multitasking :

Reduces the computation time.

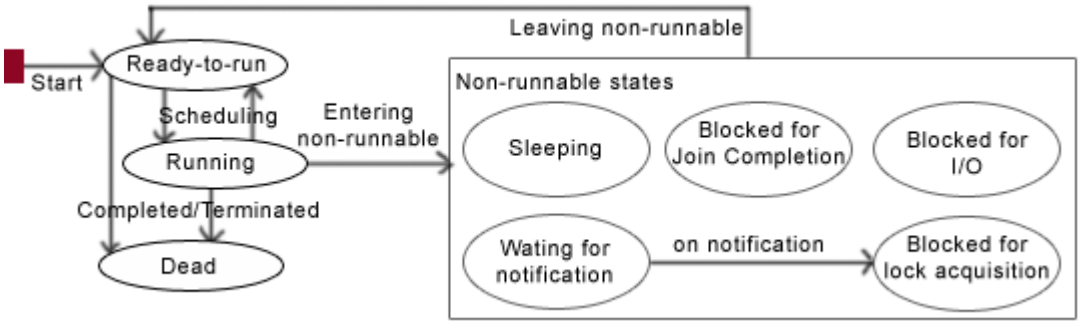
Improves performance of an application.

Threads share the same address space so it saves the memory.

Context switching between threads is usually less expensive than between processes.

Cost of communication between threads is relatively low.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

Sleeping – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method sleep( ) to stop the running state of a thread.

static void sleep(long millisecond) throws InterruptedException

Waiting for Notification – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

final void wait(long timeout) throws InterruptedException

final void wait(long timeout, int nanos) throws InterruptedException

final void wait() throws InterruptedException

Blocked on I/O – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.

Blocked for joint completion – The thread can come on this state because of waiting the completion of another thread.

Blocked for lock acquisition – The thread can come on this state because of waiting to acquire the lock of an object.

UNIT-4
EXCEPTION HANDLING

CLASSIFICATION OF EXCEPTIONS

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language
or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method. Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that user want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw.

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

This is the general form of an exception-handling block:

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

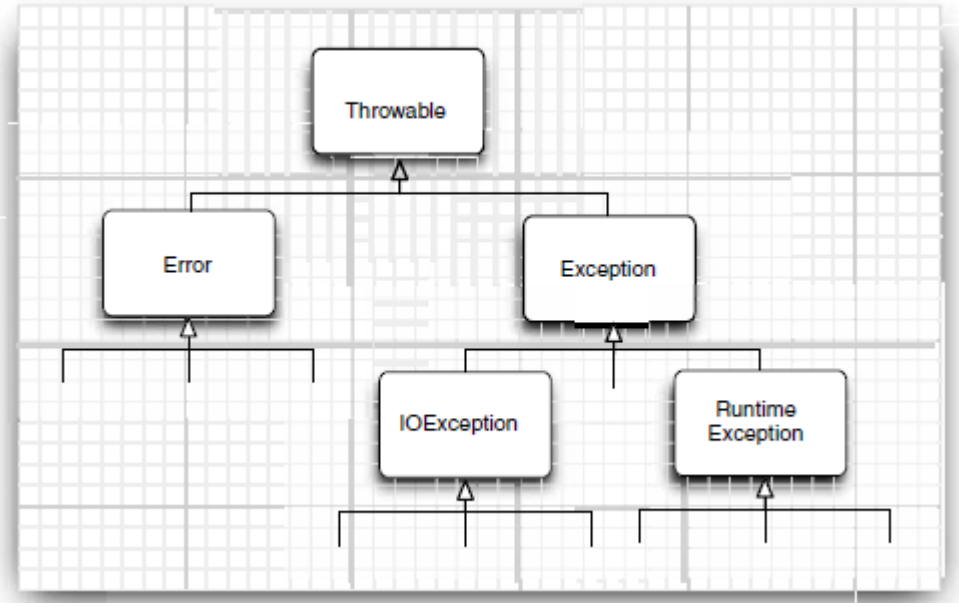


FIG : EXCEPTION HIERARCHY IN JAVA

The Error hierarchy describes internal errors and resource exhaustion inside the Java runtime system. You should not throw an object of this type. There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully. These situations are quite rare. When doing Java programming, you

focus on the Exception hierarchy. The Exception hierarchy also splits into two branches: exceptions that derive from RuntimeException and those that do not. The general rule is this: A RuntimeException happens because you made a programming error. Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.

Exception Types

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

2 .CREATION OF EXCEPTION CLASSES

User's code may run into a problem that is not adequately described by any of the standard exception classes. In this case, it is easy enough to create own exception class. Just derive it from Exception or from a child class of Exception such as IOException. It is customary to give both a default constructor and a constructor that contains a detailedmessage. (The toString method of the Throwable superclass prints that detailed message, which is handy for debugging.)

```
class FileFormatException extends IOException
{
public FileFormatException() {}
public FileFormatException(String gripe)
{
super(gripe);
}
}
```

Now you are ready to throw your very own exception type.

```
String readData(BufferedReader in) throws FileFormatException
{
. . .
```

```
while (. . .)
{
if (ch == -1) // EOF encountered
{
if (n < len)
throw new FileFormatException();
}
. . .
}
return s;
}
```

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows user to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused if the program stopped running and printed a stack trace whenever an error occurred. It is quite easy to prevent this.

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

```
class Exc2 {
public static void main(String args[]) {
int d, a;
try { // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
} catch (ArithmeticException e) { // catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

Division by zero.

After catch statement. Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. Put differently, catch

is not "called," so execution never "returns" to the try block from a catch. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.

3.CATCHING EXCEPTIONS

If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace. Graphics programs (both applets and applications) catch exceptions, print stack trace messages, and then go back to the user interface processing loop. (When you are
debugging a graphically based program, it is a good idea to keep the console available on the screen and not minimized.)
To catch an exception, you set up a try/catch block. The simplest form of the try block is
as follows:

```
try
{
code
more code
more code
}
catch (ExceptionType e)
{
handler for this type
}
```

If any of the code inside the try block throws an exception of the class specified in the catch clause, then
1. The program skips the remainder of the code in the try block.
2. The program executes the handler code inside the catch clause.
If none of the code inside the try block throws an exception, then the program skips the catch clause.
If any of the code in a method throws an exception of a type other than the one named in the catch clause, this method exits immediately. To show this at work, we show some fairly typical code for reading in data:

```
public void read(String filename)
{
try
{
InputStream in = new FileInputStream(filename);
int b;
while ((b = in.read()) != -1)
{
```

```
process input
}
}
catch (IOException exception)
{
exception.printStackTrace();
}
}
```

Catching Multiple Exceptions

User can catch multiple exception types in a try block and handle each type differently. User use a separate catch clause for each type as in the following example:

```
try
{
code that might throw exceptions
}
catch (MalformedURLException e1)
{
emergency action for malformed URLs
}
catch (UnknownHostException e2)
{
emergency action for unknown hosts
}
catch (IOException e3)
{
emergency action for all other I/O problems
}
```

The exception object (e1, e2, e3) may contain information about the nature of the exception. To find out more about the object, try e3.getMessage() to get the detailed error message (if there is one), or e3.getClass().getName() to get the actual type of the exception object.

NOTE : WRITE THE PROGRAM TO HANDLE MULTIPLE EXCEPTIONS

STREAMS AND FILES

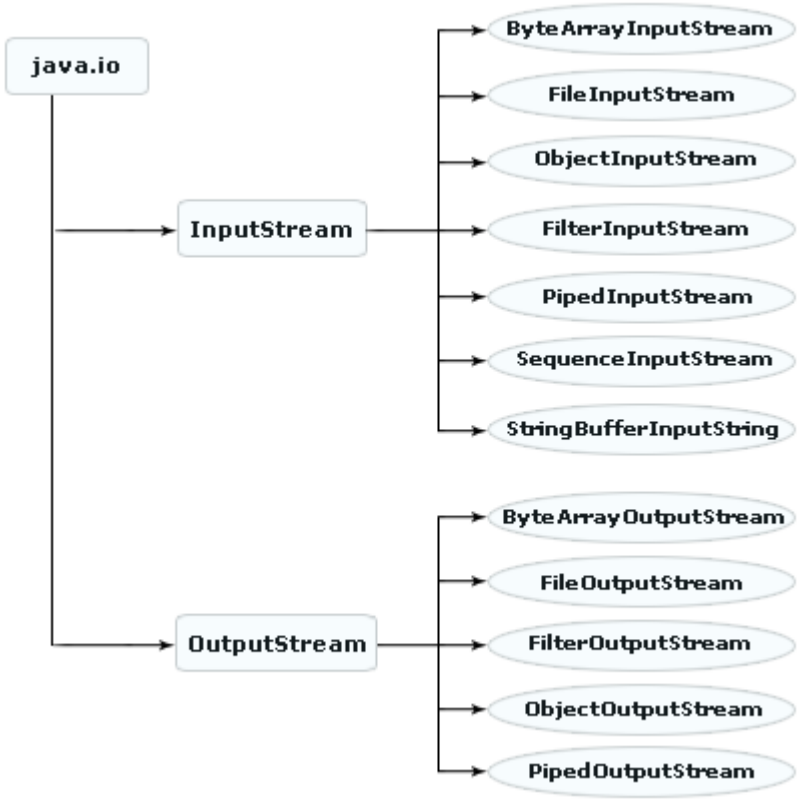
The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

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 does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionlity related to streams and I/O. Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

The Java Input/Output (I/O) is a part of java.io package. The java.io package contains a relatively large number of classes that support input and output operations. The classes in the package are primarily abstract classes and stream-oriented that define methods and subclasses which allow bytes to be read from and written to files or other input and output sources. The InputStream and OutputStream are central classes in the package which are used for reading from and writing to byte streams, respectively.

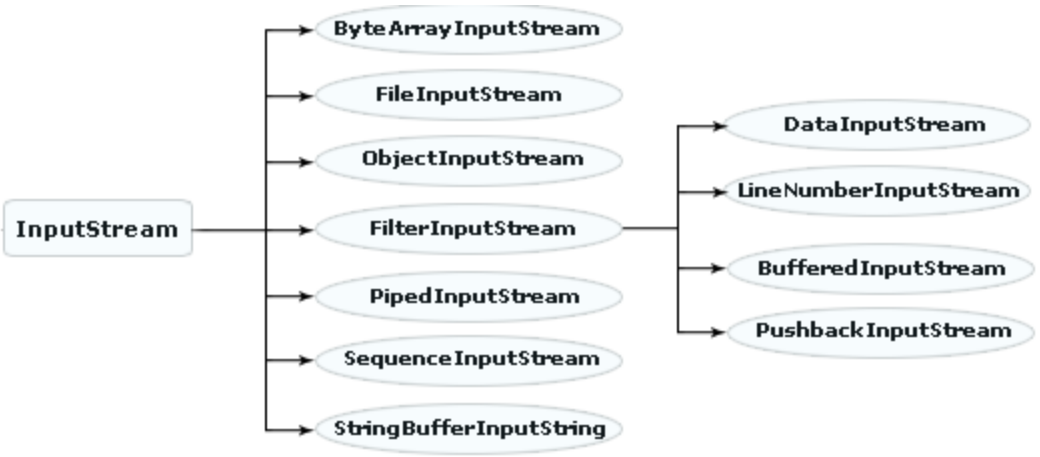
The java.io package can be categories along with its stream classes in a hierarchy structure shown below:



InputStream:

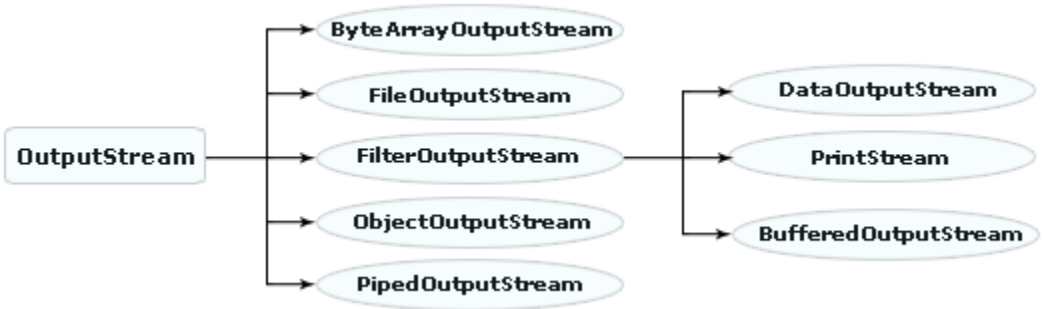
The InputStream class is used for reading the data such as a byte and array of bytes from an input source. An input source can be a file, a string, or memory that may contain the data. It is an abstract class that defines the programming interface for all input streams that are inherited from it. An input stream is automatically opened when you create it. You cans explicitly close a stream with the close( ) method, or let it be closed implicitly when the object is found as a garbage.
The subclasses inherited from the InputStream class can be seen in a hierarchy manner shown below:

InputStream is inherited from the Object class. Each class of the InputStreams provided by the java.io package is intended for a different purpose.



OutputStream:

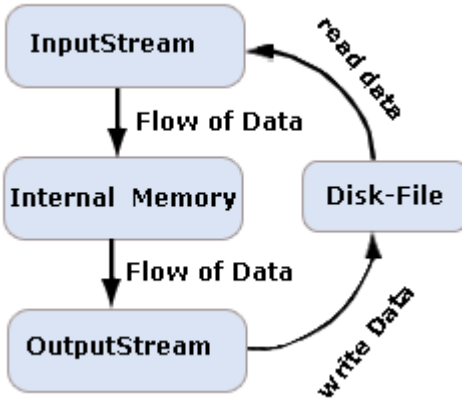
The OutputStream class is a sibling to InputStream that is used for writing byte and array of bytes to an output source. Similar to input sources, an output source can be anything such as a file, a string, or memory containing the data. Like an input stream, an output stream is automatically opened when you create it. You can explicitly close an output stream with the close( ) method, or let it be closed implicitly when the object is garbage collected.
The classes inherited from the OutputStream class can be seen in a hierarchy structure shown below:
OutputStream is also inherited from the Object class. Each class of the OutputStreams provided by the java.io package is intended for a different purpose.



How Files and Streams Work:

Java uses streams to handle I/O operations through which the data is flowed from one location to another. For example, an InputStream can flow the data from a disk file to the internal memory and an OutputStream can flow the data from the internal memory to a disk

file. The disk-file may be a text file or a binary file. When we work with a text file, we use a character stream where one character is treated as per byte on disk. When we work with a binary file, we use a binary stream.
The working process of the I/O streams can be shown in the given diagram.



String Tokenizers And Delimited Text :

The processing of text often consists of parsing a formatted input string. Parsing is the division of text into a set of discrete parts, or tokens, which in a certain sequence can convey a semantic meaning. The StringTokenizer class provides the first step in this parsing process, often called the lexer (lexical analyzer) or scanner. StringTokenizer implements the Enumeration interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using StringTokenizer.
To use StringTokenizer, you specify an input string and a string that contains delimiters. Delimiters are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, ",;:" sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, newline, and carriage return.

The StringTokenizer constructors are shown here:

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

In all versions, str is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, delimiters is a string that specifies the delimiters. In the third

version, if delimAsToken is true, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned.

Delimiters are not returned as tokens by the first two forms. Once you have created a StringTokenizer object, the nextToken( ) method is used to extract consecutive tokens. The hasMoreTokens( ) method returns true while there are more tokens to be extracted. Since StringTokenizer implements Enumeration, the hasMoreElements( ) and nextElement( ) methods are also implemented, and they act the same as hasMoreTokens( ) and nextToken( ), respectively.

NOTE : WRITE AN EXAMPLE ON STRING TOKENIZIER

READING DELIMITED INPUT

A simple text scanner which can parse primitive types and strings using regular expressions.

A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in);
   int i = sc.nextInt();
```

As another example, this code allows long types to be assigned from entries in a file myNumbers:

```
Scanner sc = new Scanner(new File("myNumbers"));
while (sc.hasNextLong()) {
   long aLong = sc.nextLong();
}
```

NOTE : WRITE AN EXAMPLE ON SCANNER

FILE MANAGEMENT :

The File class deals with the machine dependent files in a machine-independent manner i.e. it is easier to write platform-independent code that examines and manipulates files using the File class. This class is available in the java.lang package.

The java.io.File is the central class that works with files and directories. The instance of this class represents the name of a file or directory on the host file system. When a File object is created, the

system doesn't check to the existence of a corresponding file/directory. If the file exist, a program can examine its attributes and perform various operations on the file, such as renaming it, deleting it, reading from or writing to it.
The constructors of the File class are shown in the table:

Constructor	Description
File(path)	Create File object for default directory (usually where program is located).
File(dirpath,fname)	Create File object for directory path given as string.
File(dir, fname)	Create File object for directory.

Thus the statement can be written as:

File f = new File("<filename>");
The methods that are used with the file object to get the attribute of a corresponding file shown in the table.

Method	Description
f.exists()	Returns true if file exists.
f.isFile()	Returns true if this is a normal file.
f.isDirectory()	true if "f" is a directory.
f.getName()	Returns name of the file or directory.
f.isHidden()	Returns true if file is hidden.
f.lastModified()	Returns time of last modification.
f.length()	Returns number of bytes in file.
f.getPath()	path name.
f.delete()	Deletes the file.
f.renameTo(f2)	Renames f to File f2. Returns true if successful.
f.createNewFile()	Creates a file and may throw IOException.

Eg:

```
import java.io.*;
public class CreateFile1{
   public static void main(String[] args) throws IOException{
     File f;
     f=new File("myfile.txt");
     if(!f.exists()){
     f.createNewFile();
     System.out.println("New file \"myfile.txt\" has been created
                    to the current directory");
     }
   }}
```


APPLETS:

Applets are Java programs that are included in an HTML page. The HTML page must tell the browser which applets to load and then where to put each applet on the web page.The tag needed to use an applet must tell the browser where to get the class files, and how the applet is positioned on the web page (size, location,and so on). The browser then retrieves the class files from the Internet (or from a directory on the user's machine) and automatically runs the applet.

When applets were first developed,user had to use Sun's HotJava browser to view web pages that contained applets. Naturally, few users were willing to use a separate browser just to enjoy a new web feature. Java applets became really popular when Netscape included a Java virtual machine in its Navigator browser. Microsoft Internet Explorer soon followed suit. Unfortunately, two problems happened.

Netscape didn't keep up with more modern versions of Java, and Microsoft vacillated between reluctantly supporting outdated Java versions and dropping Java support altogether.

To overcome this problem, Sun released a tool called the "Java Plug-in." Using the various extension mechanisms, it seamlessly plugs in to a variety of browsers and enables them to execute Java applets by using an external Java runtime environment that Sun supplies. By keeping the Plug-in up-to-date, you can always take advantage of the latest and greatest features of Java.

A Simple Applet

An applet is simply a Java class that extends the java.applet.Applet class.

```
import java.awt.*;
import javax.swing.*;
public class NotHelloWorldApplet extends JApplet
{
public void init()
. {
. EventQueue.invokeLater(new Runnable()
{
public void run()
{
JLabel label = new JLabel("Not a Hello, World applet", SwingConstants.CENTER);
add(label);
}
});
```



```
}
}
```

To execute the applet, you carry out two steps:
1. Compile your Java source files into class files.
2. Create an HTML file that tells the browser which class file to load first and how to size the applet.

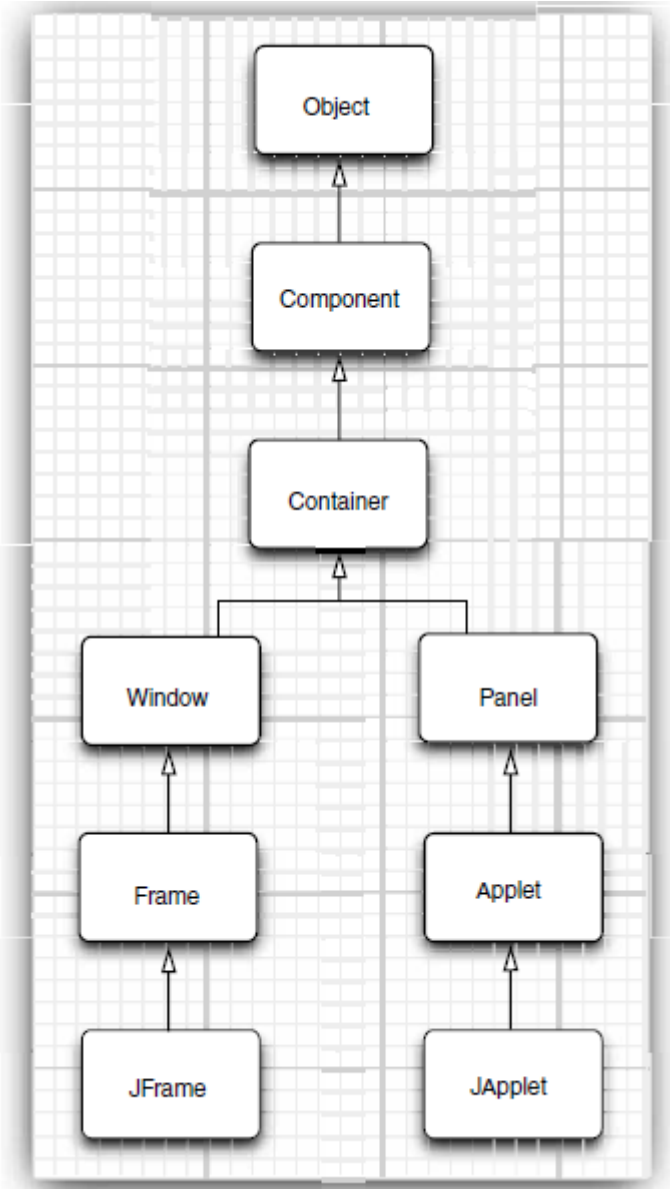


FIG : Applet inheritance diagram

It is customary (but not necessary) to give the HTML file the same name as that of the applet class inside. So, following this tradition, we call the file NotHelloWorldApplet.html.
Here are the contents of the file:

<applet code="NotHelloWorldApplet.class" width="300" height="300"> </applet>

The applet viewer is good for the first stage of testing, but at some point you need to run your applets in a browser to see them in the same way a user might use them. In particular, the applet viewer program shows you only the applet, not the surrounding
HTML text. If an HTML file contains multiple applet tags, the applet viewer pops up multiple windows.

The Applet HTML Tag and Its Attributes

In its most basic form, an example for using the applet tag looks like this:

<applet code="NotHelloWorldApplet.class" width="300" height="100">

the code attribute gives the name of the class file and must include the .class extension; the width and height attributes size the window that will hold the applet.

Both are measured in pixels. You also need a matching </applet> tag that marks the end of the HTML tagging needed for an applet. The text between the <applet> and </applet> tags is displayed only if the browser cannot show applets. The code, width, and height
attributes are required. If any are missing, the browser cannot load users applet.

All this information would usually be embedded in an HTML page that, at the very least, might look like this:

```
<html>
<head>
<title>NotHelloWorldApplet</title>
</head>
<body>
<p>The next line of text is displayed under the auspices of Java:</p>
<applet code="NotHelloWorldApplet.class" width="100" height="100">
If your browser could show Java, you would see an applet here.
</applet>
</body>
</html>
```

user can use the following attributes within the applet tag:

• width, height

These attributes are required and give the width and height of the applet, measured in pixels. In the applet viewer, this is the initial size of the applet.User can resize any window that the applet viewer creates. In a browser, user cannot resize the applet.

user will need to make a good guess about how much space your applet requires to show up well for all users.

• align

The attribute values are the same as for the align attribute of the HTML img tag.

• vspace, hspace

These optional attributes specify the number of pixels above and below the applet (vspace) and on each side of the applet (hspace).

• code

This attribute gives the name of the applet's class file. This name is taken relative to the codebase (see below) or relative to the current page if the codebase is not specified.

The path name must match the package of the applet class. For example, if the applet

class is in the package com.mycompany, then the attribute is code="com/mycompany/MyApplet.class".

The alternative code="com.mycompany.MyApplet.class" is also permitted. But you cannot use

absolute path names here. Use the codebase attribute if your class file is located elsewhere. The code attribute specifies only the name of the class that contains the applet class.

Once the browser's class loader loads the class containing the applet, it will realize that it needs more class files and will load them.

Either the code or the object attribute is required.

• codebase

This optional attribute is a URL for locating the class files. You can use an absolute URL, even to a different server. Most commonly, though, this is a relative URL that points to a subdirectory. For example, if the file layout looks like this: then you use the following tag in MyPage.html:

<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">

• archive

This optional attribute lists the JAR file or files containing classes and otherresources for the applet. These files are fetched from the web server before the applet is loaded. This technique speeds up the loading process significantly because only one HTTP request is necessary to load a JAR file that contains many smaller

files. The JAR files are separated by commas. For example:

```
<applet code="MyApplet.class"
archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
width="100" height="150">
```

LIFE CYCLE OF AN APPLET

The lifecycle methods of an Applet:
init(): This method is called to initialized an applet
start(): This method is called after the initialization of the applet.
stop(): This method can be called multiple times in the life cycle of an Applet.
destroy(): This method is called only once in the life cycle of the applet when applet is destroyed.

init () method: The life cycle of an applet is begin on that time when the applet is first loaded into the browser and called the init() method. The init() method is called only one time in the life cycle on an applet. The init() method is basically called to read the PARAM tag in the html file. The init () method retrieve the passed parameter through the PARAM tag of html file using get Parameter() method All the initialization such as initialization of variables and the objects like image, sound file are loaded in the init () method .After the initialization of the init() method user can interact with the Applet and mostly applet contains the init() method.

Start () method: The start method of an applet is called after the initialization method init(). This method may be called multiples time when the Applet needs to be started or restarted. For Example if the user wants to return to the Applet, in this situation the start Method() of an Applet will be called by the web browser and the user will be back on the applet. In the start method user can interact within the applet.

Stop () method: The stop() method can be called multiple times in the life cycle of applet like the start () method. Or should be called at least one time. There is only miner difference between the start() method and stop () method. For example the stop() method is called by the web browser on that time When the user leaves one applet to go another applet and the start() method is called on that time when the user wants to go back into the first program or Applet.

destroy() method: The destroy() method is called only one time in the life cycle of Applet like init() method. This method is called only on that time when the browser needs to Shut down.

AWT EVENT HIERARCHY

The EventObject class has a subclass AWTEvent, which is the parent of all AWT event classes.

Some of the Swing components generate event objects of yet more event types; these directly extend EventObject, not AWTEvent.

The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that were passed to the listener object, as we did in the button example with the getSource and getActionCommand methods.

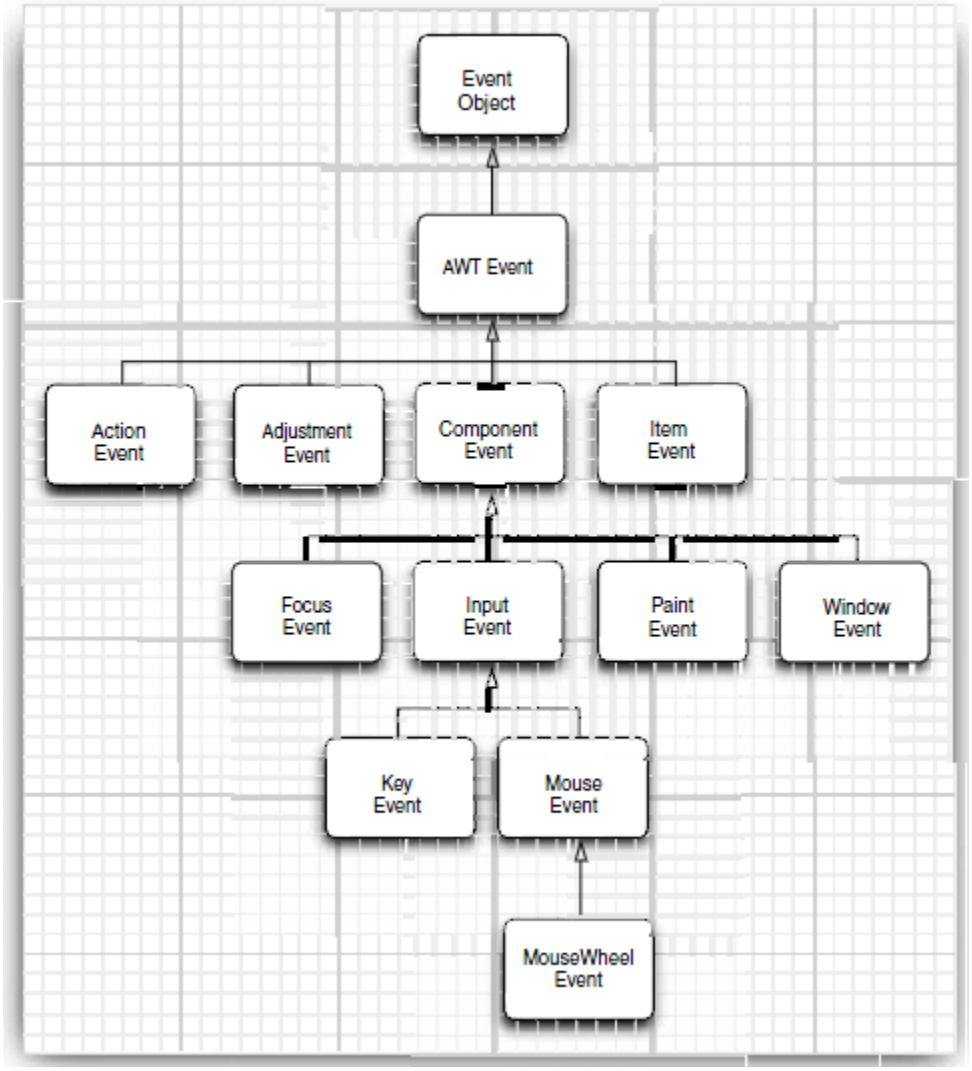


FIG : Inheritance diagram of AWT event classes

Semantic and Low-Level Events

Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts PaintEvent objects into the event queue, but these objects are not delivered to listeners. Java programmers don't listen to paint events; they override the paintComponent method to control repainting. The AWT also generates a number of events that are needed only by system programmers, to provide input systems for ideographic languages, automated testing robots, and so on.

The AWT makes a useful distinction between low-level and semantic events. A semantic event is one that expresses what the user is doing, such as "clicking that button"; hence, an Action-
Event is a semantic event. Low-level events are those events that make this possible. In the case of a button click, this is a mouse down, a series of mouse moves, and a mouse up (but only if the mouse up is inside the button area). Or it might be a keystroke, which happens if the user selects the button with the TAB key and then activates it with the space bar. Similarly, adjusting a scrollbar is a semantic event, but dragging the mouse is a low-level event.
The most commonly used semantic event classes in the java.awt.event package:
• ActionEvent (for a button click, a menu selection, selecting a list item, or ENTER typed in a text field)
• AdjustmentEvent (the user adjusted a scrollbar)
• ItemEvent (the user made a selection from a set of checkbox or list items)
Five low-level event classes are commonly used:
• KeyEvent (a key was pressed or released)
• MouseEvent (the mouse button was pressed, released, moved, or dragged)
• MouseWheelEvent (the mouse wheel was rotated)
• FocusEvent (a component got focus or lost focus)
• WindowEvent (the window state changed)
The following interfaces listen to these events:
ActionListener MouseMotionListener
AdjustmentListener MouseWheelListener
FocusListener WindowListener
ItemListener WindowFocusListener
KeyListener WindowStateListener
MouseListener
Several of the AWT listener interfaces, namely, those that have more than one method, come with a companion adapter class that implements all the methods in the interface to do nothing. (The other

interfaces have only a single method each, so there is no benefit in having adapter classes for these interfaces.)
Here are the commonly used adapter classes:
FocusAdapter MouseMotionAdapter
KeyAdapter WindowAdapter
MouseAdapter

THREADS :
a thread is a program's path of execution. Most programs written today run as a single thread, causing problems when multiple events or actions need to occur at the same time. Let's say, for example, a program is not capable of drawing pictures while reading keystrokes. The program must give its full attention to the keyboard input lacking the ability to handle more than one event at a time. The ideal solution to this problem is the seamless execution of two or more sections of a program at the same time. Threads allows us to do this.

Multithreaded applications deliver their potent power by running many threads concurrently within a single program. From a logical point of view, multithreading means multiple lines of a single program can be executed at the same time, however, it is not the same as starting a program twice and saying that there are multiple lines of a program being executed at the same time. In this case, the operating system is treating the programs as two separate and distinct processes. Under Unix, forking a process creates a child process with a different address space for both code and data. However, fork() creates a lot of overhead for the operating system, making it a very CPU-intensive operation. By starting a thread instead, an efficient path of execution is created while still sharing the original data area from the parent

Creating threads

Java's creators have graciously designed two ways of creating threads: implementing an interface and extending a class. Extending a class is the way Java inherits methods and variables from a parent class. In this case, one can only extend or inherit from a single parent class. This limitation within Java can be overcome by implementing interfaces, which is the most common way to create threads. (Note that the act of inheriting merely allows the class to be run as a thread. It is up to the class to start() execution, etc.)

Interfaces provide a way for programmers to lay the groundwork of a class. They are used to design the requirements for a set of classes to implement. The interface sets everything up, and the class or classes

that implement the interface do all the work. The different set of classes that implement the interface have to follow the same rules.

There are a few differences between a class and an interface. First, an interface can only contain abstract methods and/or static final variables (constants). Classes, on the other hand, can implement methods and contain variables that are not constants. Second, an interface cannot implement any methods. A class that implements an interface must implement all methods defined in that interface. An interface has the ability to extend from other interfaces, and (unlike classes) can extend from multiple interfaces. Furthermore, an interface cannot be instantiated with the new operator; for example, `Runnable a=new Runnable();` is not allowed.

The first method of creating a thread is to simply extend from the `Thread` class. Do this only if the class you need executed as a thread does not ever need to be extended from another class. The `Thread` class is defined in the package java.lang, which needs to be imported so that our classes are aware of its definition.

```
import java.lang.*;
public class Counter extends Thread
{
	public void run()
	{
	....
	}
}
```

The above example creates a new class `Counter` that extends the `Thread` class and overrides the `Thread.run()` method for its own implementation. The `run()` method is where all the work of the `Counter` class thread is done. The same class can be created by implementing Runnable:

```
import java.lang.*;
public class Counter implements Runnable
{
	Thread T;
	public void run()
	{
	....
	}
}
```


Here, the abstract run() method is defined in the Runnable interface and is being implemented. Note that we have an instance of the Thread class as a variable of the Counter class. The only difference between the two methods is that by implementing Runnable, there is greater flexibility in the creation of the class Counter. In the above example, the opportunity still exists to extend the Counter class, if needed. The majority of classes created that need to be run as a thread will implement Runnable since they probably are extending some other functionality from another class.

Do not think that the Runnable interface is doing any real work when the thread is being executed. It is merely a class created to give an idea on the design of the Thread class. In fact, it is very small containing only one abstract method. Here is the definition of the Runnable interface directly from the Java source:

```
package java.lang;
public interface Runnable {
	public abstract void run();
}
```

That is all there is to the Runnable interface. An interface only provides a design upon which classes should be implemented. In the case of the Runnable interface, it forces the definition of only the run() method. Therefore, most of the work is done in the Thread class. A closer look at a section in the definition of the Thread class will give an idea of what is really going on:

```
public class Thread implements Runnable {
...
	public void run() {
		if (target != null) {
			target.run();
		}
	}
...
}
```

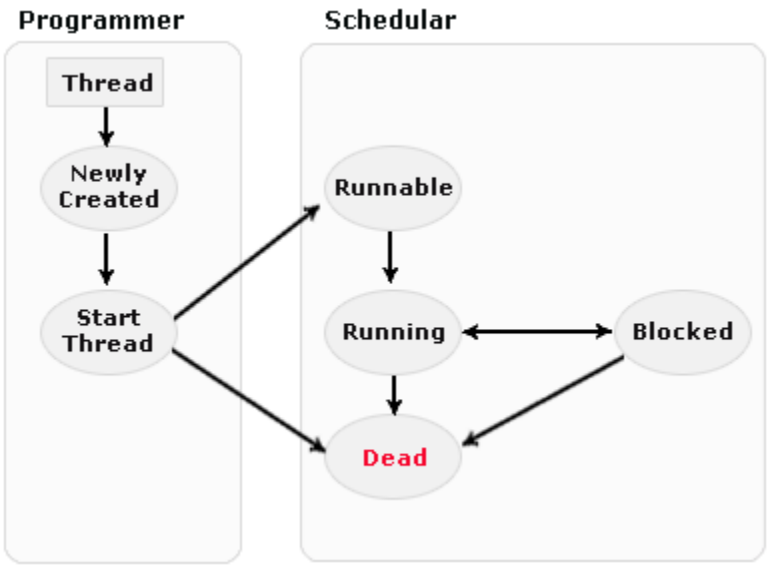
From the above code snippet it is evident that the Thread class also implements the Runnable interface. Thread.run() checks to make sure that the target class (the class that is going to be run as a thread) is not equal to null, and then executes the run() method of the target.

When this happens, the run() method of the target will be running as its own thread.

LIFE CYCLE OF A THREAD

When you are programming with threads, understanding the life cycle of thread is very valuable. While a thread is alive, it is in one of several states. By invoking start() method, it doesn't mean that the thread has access to CPU and start executing straight away. Several factors determine how it will proceed.

Different states of a thread are :



New state – After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.

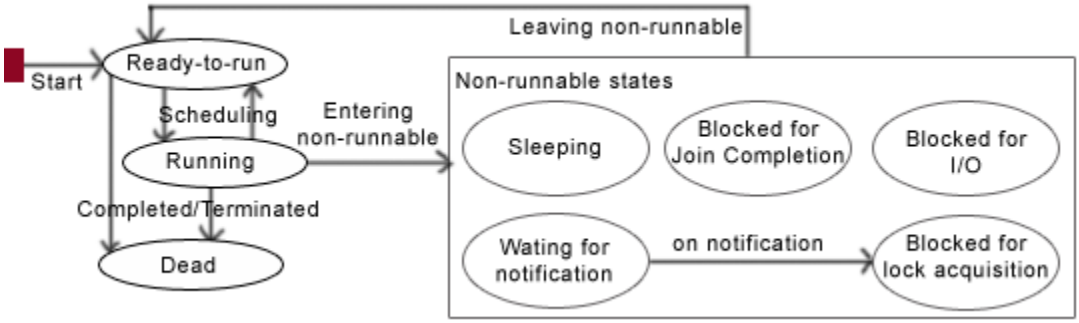
Runnable (Ready-to-run) state – A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.

Running state – A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.

Dead state – A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.

Blocked - A thread can enter in this state because of waiting the resources that are hold by another thread.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

Sleeping – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method sleep( ) to stop the running state of a thread.

static void sleep(long millisecond) throws InterruptedException

Waiting for Notification – A thread waits for notification from another thread. The thread sends back to runnable state after sending notification from another thread.

final void wait(long timeout) throws InterruptedException

final void wait(long timeout, int nanos) throws InterruptedException

final void wait() throws InterruptedException

Blocked on I/O – The thread waits for completion of blocking operation. A thread can enter on this state because of waiting I/O resource. In that case the thread sends back to runnable state after availability of resources.

Blocked for joint completion – The thread can come on this state because of waiting the completion of another thread.

Blocked for lock acquisition – The thread can come on this state because of waiting to acquire the lock of an object.

MULTITHREADING

Multitasking :

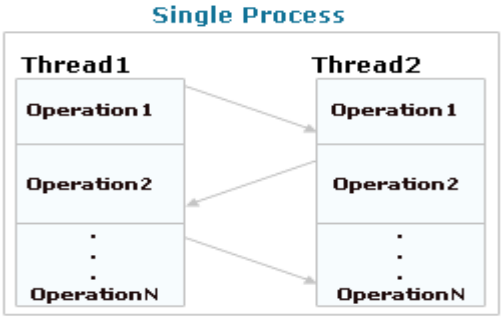
Multitasking allow to execute more than one tasks at the same time, a task being a program. In multitasking only one CPU is involved but it can switches from one program to another program so quickly that's why it gives the appearance of executing all of the programs at the same time. Multitasking allow processes (i.e. programs) to run concurrently on the program. For Example running the spreadsheet program and you are working with word processor also. Multitasking is running heavyweight processes by a single OS.

Multithreading :

Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, When you use a word processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.

In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.

For example, look at the diagram shown as:



In the above diagram, two threads are being executed having more than one task. The task of each thread is switched to the task of another thread.

Advantages of multithreading over multitasking :

Reduces the computation time.

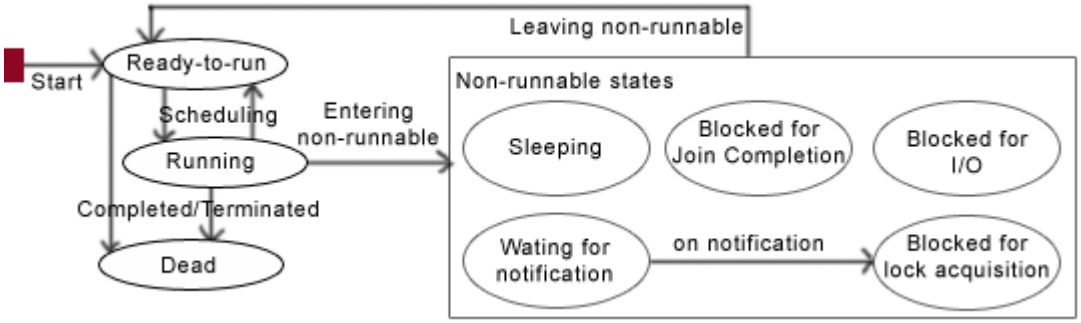
Improves performance of an application.

Threads share the same address space so it saves the memory.

Context switching between threads is usually less expensive than between processes.

Cost of communication between threads is relatively low.

Different states implementing Multiple-Threads are:



As we have seen different states that may be occur with the single thread. A running thread can enter to any non-runnable state, depending on the circumstances. A thread cannot enters directly to the running state from non-runnable state, firstly it goes to runnable state. Now lets understand the some non-runnable states which may be occur handling the multithreads.

Sleeping – On this state, the thread is still alive but it is not runnable, it might be return to runnable state later, if a particular event occurs. On this state a thread sleeps for a specified amount of time. You can use the method sleep( ) to stop the running state of a thread.

static void sleep(long millisecond) throws InterruptedException

