JAVA

What is Java? Why it is called Object Oriented?

Java is a computer language but with a difference. Java has adopted the best features of many existing languages such as C and C++ and added few new features to form a simple and easy to learn language. It has now emerged as the language of choice of the world, computing community due to its simplicity, portability and security. Java is used in almost all applications from simple home appliances control systems to compile space control statements. It has also revolutionised application development for intranet and the internet.

Object oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of each module on demand. Java is a true object oriented language. All program codes and Java resides with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance. The object model in Java is simple and easy to extend itself.

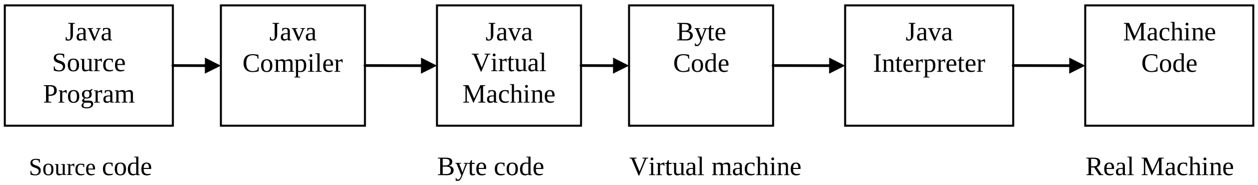
Java as a Platform Independent.

Platform Independent

Java is a platform independent language. Java programs can be easily moved from one computer system to another system anywhere and at any time.Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. Changes and upgrades in operating system processes and system resources will not force any changes in the java programs. Therefore. Java has become a popular language for programming on internet which interconnects dif6ferent kinds of systems of world wide.

JAVA Virtual Machine

JVM stands for Java Virtual Machine. All language compilers translate source code into machine code for a specific computer. Java compiler also does the same thing. It provides an intermediate code known as BYTE CODE for a machine that does not exist. This machine is called Java Virtual Machine and resides only inside the computer's memory.



The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by Java Interpreter by acting as intermediately between virtual machine and real machine as shown if fig. above.

ByteCode:

Bytecode is the key that allows java to solve both security and portability problems. Bytecode is a highly optimisd set of instructions designed to be executed by java runtime system (i.e. JVM). In essence, JVM is an interpretor for Bytecode. As the Bytecode is highly optimized, the use of bytecode enables JVM to execute programs much faster than we expect.

Translating a java program into Bytecode makes it much easier to run a program in a wide variety of environments. Bytecode is also reffered to as Virtual Machine code.

Difference between C++ and JAVA:-

1. Java has no preprocessor. If you want to use classes in another library, you say Import and the name of the library. There are no preprocessors-like macros.
2. there are no pointers in the sense of C and C++. When you create an object with new, you get back a reference.
3. there are no destructors in Java. There is no “scope” of a variable per seen, to indicate when the object’s lifetime is ended-the lifetime of an object is determined instead by the garbage collection.
4. There is no GOTO statement in JAVA.
5. No INLINE methods. The java compiler might decide it’s own to inline a method, but you don’t have much control over this. You can suggest inlining in java by using FINAL keyword for a method. However , inline functions are only suggestions to the C++ compiler as well.
6. Java has method overloading that works virtually identically to C++ function overloading.
7. Java doesn’t create exe file after the execution of a program.
8.
9. > In Java, the sizes of int, long etc. are rigidly defined in terms of
10. > bits. In C++ they are platform−dependent.
11. >
12. > In Java, the JVM behaves at if it were big endian, even if internally
13. > it is actually little−endian. In C++, the endianness is platform
14. > dependent.
15. >
16. > In Java, garbage collection of unreferenced objects is automatic. In
17. > C++, you manually manage memory.
18. >
19. > In Java, references are constrained to point only to the beginnings of
20. > objects. In C++, you can do arithmetic on pointers and make pointers
21. > point anywhere in the address space.
22. >
23. > In Java you cannot overload operators. In C++, you can.
24. >
25. > In Java, by default methods are virtual (overrideable). In C++, by
26. > default, methods are non−virtual.
27. >
28. > Java object code (class files containing JVM byte codes) will run
29. > unmodified on any platform. C++ object code must be first linked to

30. > produce an executable containing platform−specific machine
31. > instructions. It will run only on one platform.
32. >
33. > Java checks all subscripts that they are in bounds and all casts for
34. > validity. C++ does not.
35. >
36. > Java requires a JVM to execute. C++ programs are usually freestanding.
37. > Java does not use a preprocessor. C++ makes extensive use of a macro
38. > preprocessor.

Constants and Variables

A quantity, which may vary during a program execution, is called a **variable.**
A quantity whose value cannot be changed is called **Constant**.
e.g. in expression
c = a + b - 5;
In above expression a, b, c are variables and 5 is constant.

Rules for naming variables in JAVA

(i) A variable name is any combination of alphabets digits or underscore (_). Some compilers allow variable names up to 40-characters in length.
(ii) The first character of the variable name must be an alphabet i.e. it must not be a digit.
(iii) There should not be any blank space or comma within variable name.
(iv) No special symbol other than (_) underscore can be used within variable name e.g. ?,*,+,' etc.
(v) It should not be a keyword.

Keywords:

Keywords are the words whose meaning has already been explained to the JAVA-Compiler. The keyword cannot be used as variable name because doing so will try to assign a new meaning to keyword, which is not allowed by the compiler. e.g. float, if, int, do, for, break, while, else, switch, class, private, public etc.

Tokens :

Token is the smallest part of statement, it may be an operator, variable, constant, punctuation mark or keyword. E.g. the statement

if (a >5)

Above statement has 6 tokens as

if , **(** , **a** , > , 5 , **)**

Identifiers :

Identifiers refer to the names of variables, functions, arrays, classes etc. created by the programmer. Each language has its own rules for naming identifiers. The rules for naming identifiers in C / C++ has been given above with heading "**Rules naming variables in JAVA**"

Writing JAVA Program:

Each JAVA program defines a class. The name of class is same as that of name of .java file in which this source program is to be stored. For core java, there must exist a public function named "main", receiving command-line-arguments as an array of String class. This member method will be executed when java program is executed. As every java program is definition of a class, therefore, during the execution an object of this class is automatically created and executed "main" method on that object. We cannot execute a java program without class which makes it "pure-object-oriented". In C++, it supported the object-oriented techniques, but itself C++

was not a pure-object-oriented language because C++ used the procedure “main ( )” as startup method/function, which was not a member of any class. Therefore, it was a procedural language because the program starts from a procedure, now, objects can be declared and used within that procedure but the still it a procedure that invoked the code.

In JAVA, a set of statements are written in a method of class, that are executed sequentially to accomplish the task.

Following rules are applicable on all JAVA statements

(i) Blank spaces may be inserted between two words/keywords to improve the readability of the statement. However no blank spaces are allowed within variable, constant or keywords.

(ii) Writing statements in Java language is case sensitive.

(iii) Any Java-Statement always ends with a semicolon

(iv) Remarks can be written using double backslashes (//) on the same line
Where ever // appears, whole line after that is ignored by the compiler

If we want to add multi line remarks then we use /* ---------- */ to write remarks in between them

```
e.g.  /*…………………………………………………
        …………………………………………………
        …………………………………………………
        */
```

Types of Statements/instructions in JAVA

Now, let us see how constants, variables and expressions are combined together to form instructions
There are basically 4-types of instructions:

1. Type Declaration Instructions
2. Input / Output Instructions
3. Arithmetic and logical instructions (Expressions)
4. Control Instructions

Data Types

One of the fundamental concepts of any programming language is that of data types. Data types define the storage methods available for representing information, along with how the information is interpreted. Data types are linked tightly to the storage of variables in memory because the data type of a variable determines how the compiler interprets the contents of the memory. You already have received a little taste of data types in the discussion of literal types.

To create a variable in memory, you must declare it by providing the type of the variable as well as an identifier that uniquely identifies the variable. The syntax of the Java declaration statement for variables follows:

```
Type Identifier [, Identifier];
```

The declaration statement tells the compiler to set aside memory for a variable of type Type with the name Identifier. The optional bracketed Identifier indicates that you can make multiple declarations of the same type by separating them with commas. Finally, as in all Java statements, the declaration statement ends with a semicolon.

Java data types can be divided into two categories: simple and composite. Simple data types are core types that are not derived from any other types. Integer, floating-point, Boolean, and character types are all simple types. Composite types, on the other hand, are based on simple types, and include strings, arrays, and both classes and interfaces in general. You’ll learn about arrays later in this chapter. Classes and interfaces are covered in Chapter 14, “Classes, Packages, and Interfaces.”

In C++, Data Types are further divided into categories :

(i) Built-in (basic) data types : int, float, char, long int etc.

(ii) use defined data types : class, interface
(iii) derived data types : array, function

Integer Data Types

Integer data types are used to represent signed integer numbers. There are four integer types: byte, short, int, and long. Each of these types takes up a different amount of space in memory, as shown in Table 12.4.

Table 12.4. Java integer types. *

*Type	*Size
byte	1 byte
short	2 byte
int	4 byte
long	8 byte

To declare variables using the integer types, use the declaration syntax mentioned previously with the desired type. The following are some examples of declaring integer variables:

```
int i;
short rocketFuel;
long angle, magnitude;
byte red, green, blue;
```

Floating-Point Data Types

Floating-point data types are used to represent numbers with fractional parts. There are two floating-point types: float and double. The float type reserves storage for a 32-bit single-precision number and the double type reserves storage for a 64-bit double-precision number.

Declaring floating-point variables is very similar to declaring integer variables. The following are some examples of floating-point variable declarations:

```
float temperature;
double windSpeed, barometricPressure;
```

*Type	*Size
single	4 bytes
Double	8 bytes

boolean Data Type

The boolean data type is used to store values with one of two states: true or false. You can think of the boolean type as a 1-bit integer value, because 1 bit can have only two possible values: 1 or 0. However, instead of using 1 and 0, you use the Java keywords true and false. true and false aren't just conveniences in Java; they are actually the only legal Boolean values. This means that you can't interchangeably use Booleans and integers like in C/C++. To declare a Boolean value, just use the boolean type declaration:

```
boolean gameOver;
```

Character Data Type

The character data type is used to store single Unicode characters. Because the Unicode character set is composed of 16-bit values, the char data type is stored as a 16-bit unsigned integer. You create variables of type char as follows:

```
char firstInitial, lastInitial;
```

Remember that the char type is useful only for storing single characters. If you come from a C/C++ background, you might be tempted to try to fashion a string by creating an array of chars. In Java this isn’t necessary because the String class takes care of handling strings. This doesn’t mean that you should never create arrays of characters, it just means that you shouldn’t use a character array when you really want a string. C and C++ do not distinguish between character arrays and strings, but Java does.

Operators in JAVA

An operator is a symbol that specifies certain operation to be performed on data. The data on which operation is to be performed is/are called operand(s). operator specifies what operation is to be performed on data. There are following type of operators in JAVA.

1. Unary operators :

Operators which operates on single data. i.e. operators which need only one operand to perform the operations e.g. ++, --

++ is also called increment operator and -- as decrement operator

e.g.

```
int a=5;
a++;     // this statement will increment a by 1 i.e. add 1 to a   i.e. a=a+1
System.out.println(“ ”+a);     // this will display the value of a as 6
```

Unary operator can be placed either before or after operand.

When it is placed before operand then it may be called as pre-increment or pre-decrement.

When it is placed after operand then it may be called as post-increment or post-decrement.

It behaves differently in both cases

int a=5;

Pre-increment	**Post increment**
b = ++ a;	b = a++;
First INCREMENT Then USE	First USE Then INCREMENT
System.out.println(b+” , “+a); // 6,6	System.out.println(b+” , “+a); // 5,6

2. Binary operators :

Operators, which need two operands to operate, are called binary operators. These can be further categorized according to the type of operands they use or the result they produce

(i) Arithmatic operators :

+, - , * , / , %

Above all operators can be used with int, char, float data types except % that cannot be used with float data type because it gives the remainder obtained by division between the operands. Arithmetic expression produces a numeric result.

(ii) Relational operators :

> , < , >= , <= . = =

These operators can work on any numeric / arithmetic expressions value as operands and Produce a result, which is logical value (true/false).

(iii) Logical operators (AND , OR , NOT):

&& , ||

these operators use logical values as operands, and result of && (AND) is True when both operands are True. Result of || (OR) is True any at least one operand is True.

(iv) Bitwise logical operators:

These operators are used to manipulate the bits levels of operands. Some of these operators are << (left shift), >> (right shift), & (bitwise AND), | (bitwise OR), >>>, <<<

3. Ternary Operaros:

The operators which use 3 operands to operate on. E.g. ? :

(test condition) **?** (true-statement) **:** (false-statement);

the operator above is also called **conditional operator**.

CONTROL STRUCTURES

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

- If else statement

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.

- **Nested Ifs**

A *nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, 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.

➢ If-else-if Ladder

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

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

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will 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 other conditions 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 statement

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

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

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

The **switch** statement works like this: The value of the expression is compared with each 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**.

Here is a simple example that uses a **switch** statement:

```
// 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.
```

As you can see, each time through the loop, the statements associated with the **case** constant that matches **i** are executed. All others are bypassed. After **i** is greater than 3, no **case** statements match, so the **default** statement is executed.

The **break** statement is optional. If you omit the **break**, execution will continue on into the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between them. For example, consider the following program:

Iteration Statements

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

❖ While

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

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

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

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
```

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is false to begin with. For example, in the following fragment, the call to **println( )** is never executed:

```
int a = 10, b = 20;
while(a > b)
System.out.println("This will not be displayed");
```

The body of the **while** (or any other of Java’s loops) can be empty. This is because a *null statement* (one that consists only of a semicolon) is syntactically valid in Java.

❖ Do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the **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.

Here is a reworked version of the “tick” program that demonstrates the **do-while** loop. It generates the same output as before.

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

The loop in the preceding program, while technically correct, can be written more efficiently as follows:

```
do {
System.out.println("tick " + n);
} while(--n > 0);
```

In this example, the expression **(– –n > 0)** combines the decrement of **n** and the test for zero into one expression. Here is how it works. First, the **– –**n statement executes, decrementing **n** and returning the new value of **n**. This value is then compared with zero. If it is greater than zero, the loop continues; otherwise it terminates.

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

❖ For

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

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

If only one statement is being repeated, there is no need for the curly braces.
The **for** loop operates as follows. When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *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.
Here is a version of the “tick” program that uses a **for** loop:

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

ARRAY (LIST):

Array is a collection of homogeneous elements (elements of same data type) stored in continuous memory locations under one name. Each element of array can be accessed independently using its index number. This index number is sometimes also called subscript number and hence array elements are also called subscripted variables.

Declaration:

```
int a[]=new int[10];     // 20 bytes will be reserved for A to store 10 integer values
```

// a										
	0	1	2	3	4	5	6	7	8	9

assigning value to an element

```
a[3]=56;          // element at index number 3 will contain value 57
```

a				57						
	0	1	2	3	4	5	6	7	8	9

rray values can be initialised during declaration like

```
int a[10]={5,8,4,5,9,3,2,1,5,7};          //
```

a	5	8	4	5	9	3	2	1	5	7
	0	1	2	3	4	5	6	7	8	9

```
          or
int a[]={2,5,8,4,6}                // here length of array is automatically assumed to be 5
```

Display array

```
for(i=0; i<10; i++)
```

```
System.out.println("   " + a[i]);
```

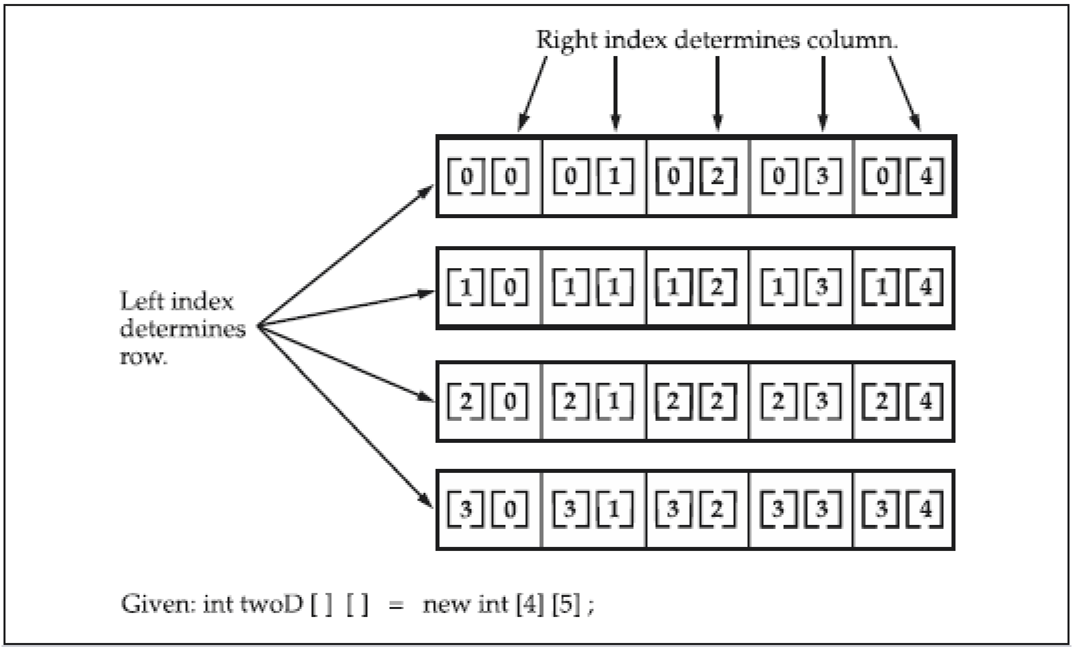
Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see,

there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

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

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1. A conceptual view of a 4 by 5, two-dimensional array



Given: int twoD [ ] [ ] = new int [4] [5] ;

Figure 3-1. *A conceptual view of a 4 by 5, two-dimensional array*

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// 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
```

FUNCTIONS/METHOD of class

Function is a self-contained block of statements that perform a specific task. This task may be performed on a class object and class members may need to be modified. In other words, a function is a subprogram /subroutine which has all the properties of a program and is invoked whenever called. It must be called either with class name or object of class using DOT (.) operator.

It behaves like a child process/program and the program calling/invoking it is called parent of this child. After finishing the child program/process the control is returned back to the parent process. On the other hand, the parent process will be blocked and waiting for the child process to be finished, so that parent process may be continued after the return of control back to parent process who invoked this child process/function/subroutine.

JAVA also provide multithreading programming environment. With the help of Threads (class that provide multi threading implementations) we can create methods that executes simultaneously as different processes.

A function can have any name that is valid for a variable
A method/function is defined as

Access-type return-type function-name (arg1, arg2, … , argn)

Access-type gives the visibility scope, it may be **private, public or protected**. Private methods of class can be used only within class or by other methods of class but cannot be used directly by class objects. Only public methods can be accessed outside the class by class objects.

Return-type represents the type of data that is returned by the function. The value returned by the function can be used where it will be called. If function does not return any value than keyword **“void”** can be use as return type, which may indicate that function will not return any value.

arg1, arg2, arg3, …. , argn are the arguments or parameters which are received by the function. This is also called massage passing. If function will not receive any arguments then ( ) may be kept empty.

Public void accept( )

Above declaration means that accept is the name of function which will not return any value and will mnot receive any value. Public indicates that it can be called by any object of this class, declared either outside or inside class.

Public void accept(int a,int b)

Above declarations means that function accept will not return any value but receive 2 int-type values and store them into local variables a and b respectively.

Public fact(int n)

above declaration implies that function named fact will return an int-value and will receive an int-value which will be stores to variable named n.

Its not necessary that all functions receive or return values, function may or may not return value, function may or may not receive value(s).

Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**. For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

Garbage collection:

In Java objects are dynamically allocated by using **new** operator, how these objects can be destroyed? In C++, **delete** operator is provided to manually destroy dynamically allocated objects. If these objects are not destroyed and their memory is not de-allocated then memory occupied by them will be wasted. The memory space, which is allocated but is not referenced by any object and is supposed to be waste, is called **garbage collection**. In Java, there is no need to worry about such non-referenced memory space, it handles de-allocation automatically. When no reference to an object exists, that object is assumed to be no longer needed, and the memory occupied by that object can be reclaimed.

Writing JAVA Program:

Each JAVA program defines a class. The name of class is same as that of name of .java file in which this source program is to be stored. For core java, there must exist a public function named "main", receiving command-line-arguments as an array of String class. This member method will be executed when java program is executed. As every java program is definition of a class, therefore, during the execution an object of this class is automatically created and executed "main" method on that object. We cannot execute a java program without class which makes it "pure-object-oriented". In C++, it supported the object-oriented techniques, but itself C++ was not a pure-object-oriented language because C++ used the procedure "main ( )" as startup method/function, which was not a member of any class. Therefore, it was a procedural language because the program starts from a procedure, now, objects can be declared and used within that procedure but the still it a procedure that invoked the code.

In JAVA, a set of statements are written in a method of class, that are executed sequentially to accomplish the task.

Following rules are applicable on all JAVA statements

(v) Blank spaces may be inserted between two words/keywords to improve the readability of the statement. However no blank spaces are allowed within variable, constant or keywords.

(vi) Writing statements in Java language is case sensitive.

(vii) Any Java-Statement always ends with a semicolon

(viii) Remarks can be written using double backslashes (//) on the same line
Where ever // appears, whole line after that is ignored by the compiler

If we want to add multi line remarks then we use /* ---------- */ to write remarks in between them

```
e.g.   /*……………………………………………………
        ……………………………………………………
        ……………………………………………………
        */
```

```
class fact
{
 public static void main(String arg[])
 {
  int num[]={1,2,3,4,5,6,7,8,9,10};
  for(int i=0;i<num.length;i++)
  {
   int f=1;
   for(int j=1;j<=num[i];j++)
   {
    f=f*j;
   }
   System.out.println("Factorial of "+num[i]+"=" +f);

  }
 }
}
```

New operator

new operator is used to create a memory space for an object. In C, there are special functions like malloc(), calloc, alloc to do this. C++ use **new** operator to allocate dynamic memory (run-time allocation). It returns the address of new object created. If space is not successfully allocated it returns ZERO (0).

```
float *ptr;
int *a;
ptr=new float(10.675);   // ptr will get a address where float number 10.675 will be stored
a=new int [10];          // a will geta address where 10 integers can be stores (i.e. 20 bytes in
                         //continous are available)
```

the memory allocated by new operator cannot be released unless we use **delete operator.**

Constructors

Constructor is one of the special type of methods which have the same name as that of class but with no-return-type. They cannot specify return-type, not even void. This is because it return the instance of class itself. Constructor enables an object to initialize itself when it is created. We can pass parameters to constructors and therefore sontructors can be overloaded.

CONTRUCTORS are of two types:

1. Default constructor
2. Parameterized ocnstructor

1. Default constructor:

constructor that do not receive any parameters are called default constructors. These type of constructors are used for default initialization of class objects.

2. Parameterized constructors:

constructor that receive at least one parameter is called parameterized constructor. Mostly the initial values of object are passed to constructors so that object may be initialized with those values. By the technique of polymorphism, constructors can be designed to receive different number or type of parameters and hence can be overloaded.

Note : There may be many parameterized constructors but only one default constructor.

e.g.

```
class rectangle
{
        int length,breadth;

rectangle()        // default constructor
        {
        length = 0;
         breadth = 0;
         }

rectangle(int x,int y)  // parameterized constructor
        {
        length=x;
        breadth=y;
        }


int area()
        {
        return(length*breadth);
        }

}

class rectarea
{
        public static void main(String args[])
        {
        rectangle r=new rectangle(6,3);
        System.out.println(“Area = “ + r.area());
        }
}
```

STRINGS

Methods /functions of string class:-

➢ String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:
int length( )
The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

➢ charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:
char charAt(int *where*)
Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value "**b**" to **ch**.

➢ startsWith( ) and endsWith( )

String defines two routines that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:
boolean startsWith(String *str*)
boolean endsWith(String *str*)
Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.
A second form of **startsWith( )**, shown here, lets you specify a starting point:
boolean startsWith(String s*tr*, int *startIndex*)
Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

➢ compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to,* or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo( )** serves this purpose. It has this general form:
int compareTo(String *str*)
Chapter 13: String Handling 359

360 Java™ 2: The Complete Reference
Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted as shown here:

Value	Meaning
Less than zero	The invoking string is less than *str.*
Greater than zero	The invoking string is greater than *str.*
Zero	The two strings are equal.

➢ Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:
_ **indexOf( )** Searches for the first occurrence of a character or substring.
_ **lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.
To search for the first occurrence of a character, use
int indexOf(int *ch*)
To search for the last occurrence of a character, use
int lastIndexOf(int *ch*)
Here, *ch* is the character being sought.
To search for the first or last occurrence of a substring, use
int indexOf(String *str*)
int lastIndexOf(String *str*)
Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:
int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)
int indexOf(String *str*, int *startIndex*)
int lastIndexOf(String *str*, int *startIndex*)
Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero

➢ substring( )

You can extract a substring using **substring( )**. It has two forms. The first is
String substring(int *startIndex*)
Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.
The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:
String substring(int *startIndex*, int *endIndex*)
Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

➢ concat( )

You can concatenate two strings using **concat( )**, shown here:
String concat(String *str*)
This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as **+**. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

➢ trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:
String trim( )
Here is an example:

```
String s = " Hello World ".trim();
```

This puts the string “Hello World” into **s**.
The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state’s capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

```
/*Program  to check whether the given string is a Palindrome or not*/
class rrr
{
 public static void main(String s[])
 {
  int n,i;
  String ar=new String("ashsa");
  n=ar.length();
  for( i=0;i<n/2;i++)
  {
  if(ar.charAt(i)!=ar.charAt(n-1-i))
  break;
  }
 if(i>=n/2)
 {
 System.out.println("Palindrome");
 }
 else
 {
 System.out.println(" Not palindrome");
 }
}
}
```

```
/* Program  to Sort the  list of strings*/
class rr23
{
 public static void main(String s[]) Throws ArrayIndexOutOfBoundException;
 {
  int n,i,j;
  String ar=new String();
  String a[]={"resham","esha","akash","dilpreet","harman","rajdeep"};
  n=a.length;
  for(i=0;i<n;i++)
  for(j=i+1;i<n;j++)
  if(a[i].compareTo(a[j])>0)
  {
  ar=a[i];
  a[i]=a[j];
  a[j]=ar;
  }
 System.out.println("SORTED LIST IS");
 for(i=0;i<n;i++)
 System.out.println(a[i]);
 }
 }
```

Static methods of class:

Normally the methods of class can be called in conjunction with an object of this class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such members, **static** keyword is used before its declaration. When a member is declared static, it can be called without creation of any object of this class. We can declare both methods and variables to be static.

Instance variables declared as static acts as global variables for all the objects of class because when objects of class are declared, no copy of static members is created. All instances of the class share the same static variable.

Methods declared as static have several restrictions:

1. They can only call static methods

2. They can only access static data
3. They cannot refer to **this** or **super** is any way.

e.g.

```
class height
{
            int feet,inch;
            static int c=0;

    height()
      {
            feet=0;
            inch=0;
            c++;
      }

finalize()
            {
            c--;
            }
public static void showcount()
            {
            System.out.println("counter is at "+c);
            }
};

class testheight
{
public static void main()
            {
            height h=new height();
            height q=new height();
            height.showcount();
            }
}
```

Final keyword:

final keyword is provided in java to say “this cannot be changed”.

If data is declared as final then it cannot be changed. It tells the compiler that this piece of data is constant and will not ever change. Final variable must be initialized at the time of declaration.

If a method is declared as final then it cannot be overridden. It turns off dynamic binding i.e. it tells the compiler that dynamic binding is not necessary.

If a class is declared as final then it means that this class definition can never be changed and no subclasses can be defined from it. In other words it prevents inheritance and do not allow other classes to inherit final class. However, objects of final class may be created in other classes. In other words, final class cannot be subclassed.

Simply we can say that final indicates that class, variable or method is not changeable

e.g. final int max_pages=5;

OR

```
    Final int max_pages;
    max_pages=5;
```

finalize() method:

JAVA provides constructors for the initialization of class objects. Finalize method is used to destroy the object. Sometimes we want to execute a set of statements at the time when an object is being vanished or when an object loses its life/scope. This set of statements may be written in a special method named finalize(). Java run-time automatically frees up the memory resources used by the objects. But, the objects may hold other non-object resources such as file description or windows system fonts. The garbage collection cannot free these resources. In other words, to free these type of resources finalize() method is used and is also called destructor.

Super keyword:

Super keyword is used by subclasses constructors to invoke the constructor method of super class.
Use of super keyword subject to following conditions

- super may only be used within a subclass's constructor mehod.
- The call to super class constructor must appear as the first statement within the subclass constructor.
- The parameters in the super class must match the order and type of the instance variable declared in super class.

Finally statement:

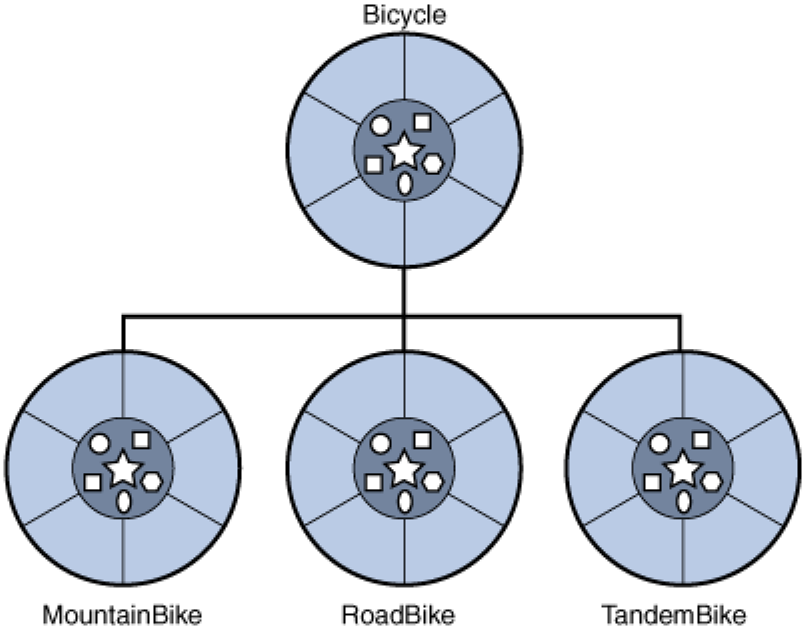
Finally statement can be used to handle an exception htat is not caught by any of the previous catch statements. Finally statement can be used to handle any exception generated within try block. It may be added immediately after the try block or after the last catch block.

When finally block is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown. We can use it to perform certain necessary operations such as closing files and releasing system resources.

INHERITANCE

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass.* The class that does the inheriting is called a *subclass.* Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

In this example, `Bicycle` now becomes the *superclass* of `MountainBike`, `RoadBike`, and `TandemBike`. In the Java programming language, each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of *subclasses*:



A hierarchy of bicycle classes.

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

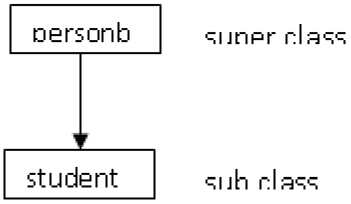
```
class MountainBike extends Bicycle {

     // new fields and methods defining a mountain bike would go here

}
```

This gives `MountainBike` all the same fields and methods as `Bicycle`, yet allows its code to focus exclusively on the features that make it unique. This makes code for your subclasses easy to read. However, you must take care to properly document the state and behavior that each superclass defines, since that code will not appear in the source file of each subclass.

TYPES OF INHERITANCE



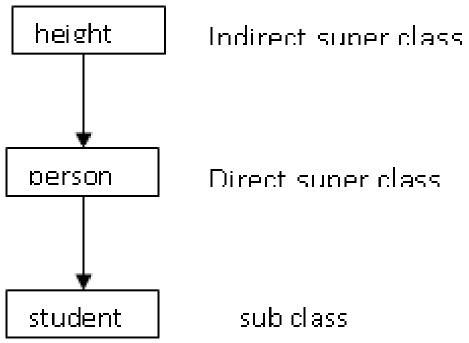
Single inheritance : If a class is derived from only one super class.

Program to show the concept of single inheritance

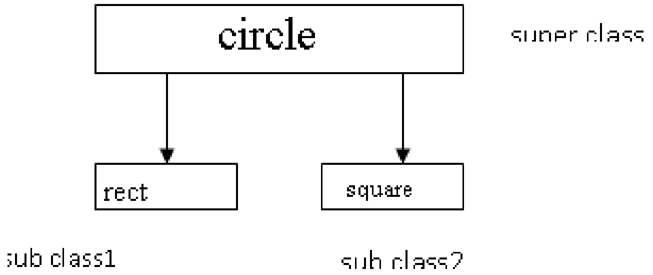
```
class height
{
 private int feet,inch;
 public height()
 {
  feet=inch=0;
 }
```

```
public height(int a,int b)
{
 feet=a;
 inch=b;
 adjust();
}
private void adjust()
{
 if(inch>=12)
 {
 feet=feet+inch/12;
 inch=inch%12;
 }
}
public void input(int a,int b)
{
 feet=a;
 inch=b;
 adjust();
 }
 public void shwheight()
 {
  System.out.println(feet+ """+inch+"""");
  }
 }

 class person extends height
 {
  String name;
  int age;
  public person(String s,int a,int f,int i)
  {
  super(f,i);
  name=s;
  age=a;
  }
  public void shwperson()
  {
   System.out.println("Name= "+name);
   System.out.println("Age= "+age);
   System.out.print("Height=");
   shwheight();
   }
  }
  class xxx
  {
  public static void main(String s[])
  {
   Student p=new student("reshu",18,5,3);
   p.shwperson();
}
```



Multilevel inheritance: If the super class is also derived from some other super class



Hierarchical inheritance : When from one super class more than one sub classes are defined.

```
//Program using hierarchical inheritence
 class circle
{
int radii;
double area;
public void get(int  r)
{
radii=r;
}
public void dis()
{
area=3.14*radii*radii;
System.out.println("Area of circle="+area);
}
}
class rect extends circle
{
int len,bre;
public void enter(int r,int b)
{
len=r;
get(r);
bre=b;
}
public void display()
{
dis();
System.out.println("Area of rectangle="+(len*bre));
}
}

class square extends circle
{
int side;
public void input(int r)
{
side=r;
}
public void displ()
{
System.out.println("Area of square="+(side*side));
}
}

class hieinherit
{
public static void main(String args[])
{
square t=new square();
rect p=new rect();
t.input(3);
t.displ();
p.enter(4,5);
p.display();
}
}
```

Levels of Access protection / Visibility mode of class members / Accessibility modifiers

There are three levels of access protection for hiding data and function members internal to the class

1. **private :**

   In private section, a member data can only be accessed by the member functions or friend functions of this class. Private data is not accessible to outside world. This modifier restricts the access to class members only.

2. **default :**

   When there is no modifier (private, public or protected) present, this is called default access. A member with default access provides access to class members to other classes within the same package.

3. **public :**

   The members which are declared as public modifier can be accessed by any function in outside world through the objects of this class. Public modifier does not restrict access to anyone. These functions are the only way to access the private members of the class by objects. Therefore, public members are also called interface to outside world. Everyone can access member classes, methods and variable which are declared as public modifier.

4. **protected :**

   The members which are declared in protected modifier, can only be accessed by member and friends functions of this class, also these functions can be accessed by member and friend functions of derived classes. It is not accessible to the outside world. It gives preferred access to subclasses in other packages.

Difference between overloading and overriding

The terms overloading and overriding are applied to situations in which you have multiple java methods with the same name. within a particular class, you can have more than one method with the same name as long as they differ if number, type and order of the input parameters. This is described as overloading. For java, these are two different methods. The duplicate names are just for the convenience of the programmer. Compiler will not allow methods with same signature but with different return type, even if one method is declared in parent class and the other in the subclass.

If a subclass method has the same name, parameter list and return type as super class method, you say that the subclass method overrides the super class method.

Overloading methods are resolved at compiler time, based on the arguments you supply. Overridden methods are resolved at run time.

Applying Method Overriding

Let’s look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of various two-dimensional objects. It also defines a method called **area( )** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

// Using run-time polymorphism.

```
class Figure
 {
double dim1;
double dim2;
Figure(double a, double b)
{
dim1 = a;
dim2 = b;
}
double area()
 {
System.out.println("Area for Figure is undefined.");
return 0;
}
}
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 FindAreas
{
public static void main(String args[])
{
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9, 5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " + figref.area());
figref = t;
System.out.println("Area is " + figref.area());
figref = f;
System.out.println("Area is " + figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( )**. The interface to this operation is the same no matter what type of figure is being used.

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation.

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

Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem.

They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

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

The syntax for creatring interfaces is shown below:

```
    interface  identifier
               {
               interfaceBody
               }
```

Here interface is the keyword, identifier is the name of the interface. Interface body consists of the static final variables and abstract methods.

```
interface intr
{
 double pi=3.1427;
 public double area();
 public double per();
 }
 class circle implements intr
 {
  int r;
  int x,y;
  public double area()
  {
  return (pi*r*r);
  }
   public circle()
   {
    r=0;
    x=y=0;
    }
    public circle(int a)
    {
     r=a;
     x=y=0;
     }
     public double per()
     {
      return(2*pi*r);
      }
     }
     class intre
     {
      public static void main(String s[])
      {
       circle c=new circle(4);
       System.out.println(c.area());
       System.out.println(c.per());
       }
       }
```

Interfaces Can Be Extended

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

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

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

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

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs and applets that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

Define error and explain its types

A program rarely executes successfully at the first attempt. It is common to make mistakes while developing as well as typing a program. Mistake might lead to an error causing the program to produce unexpected results. Errors are the wrongs that make a program go wrong.

An error may produce an incorrect output or may cause the system to crash. It is therefore important to detect and manage properly all the possible error conditions in the program so that the program will not crash during the execution.

Types of errors

1. compile-time errors
2. run-time errors

1. Compile time errors :

   All the syntax errors which are detected and displayed by the Java compiler are known as compile-time errors.

   Most of the compile time errors are due to typing mistakes. Typographic errors are hard to find. We have to check word by word or even character by character.

   The most common compile-time errors are :

   (i) missing semicolons

   (ii) missing brackets in classes or methods

(iii) Misspelled identifiers or keywords.
(iv) Use of undeclared variables.
(v) Use of = in place or = = operator

2. Run-time errors

Some programs may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic. So the errors created by logical mistakes are known as run-time errors.
Most commonly used run-time errors are:

(i) dividing an integer by 0
(ii) trying to store a value into an array of incompatible type or class
(iii) Attempting to use negative size of array
(iv) Trying to illegally change the state of thread.
(v) Converting invalid string to number.

What is Exceptions and Exception Handling?

An exception is a condition that can cause run-time error in a program. When the Java interpreter encounters an error, it creates an exception object and throws it (i.e. inform us that an error has occurred). If error (exception object) is not handled properly, then error massage is displayed and no further execution is done.

If we want the program to continue the execution of remaining code then we must catch the error (exception object) and then display appropriate massage or write appropriate statements that handle the error occurred and hence take corrective actions. This is known as Exception handling.

The purpose of exception handling mechanism is to provide a means to detect and report Exception circumstance so that appropriate action can be taken.
Exception handling method follows the following steps:

2. find the problem (hit the exception)
3. inform that error has occurred (throws exception)
4. receive the error informatio (catch the exception)
5. take corrective steps (handle the exception)

Types of Exception

- Arithmetic Exception	Caused by math error such as divide by ZERO
- Arraystore Exception	Caused when a programmer tries to store the wrong type of data in array.
- File not found Exception	Caused by an attempt to access a non-existed file

-	ArrayIndexOutOfException	Caused when Array index is out of bound
-		

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

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

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
}
```

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

Handling Arithmetic exception using try and catch block

```
Public class sample
{
Public static void main(String ab[])
{
```

```
Try
{
Int a=100,b=0,c;
System.out.println("Yes");
C=a /b ;
System.out.println("No");
} catch(Arithmetic Exception e)
{
System.out.println("error occurs");
}
}
```

Multiple catch Clauses

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

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

Throw:-

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

throw *ThrowableInstance*;

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

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

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

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

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

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java’s standard exception objects. Pay close attention to this line:

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

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

Throws:-

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

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

type method-name(parameter-list) throws *exception-list*
{
// body of method
}

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

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a **throws** clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.
class ThrowsDemo
{
static void throwOne()
{
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[])
{
throwOne();
}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception.

The corrected example is shown here:

```
/
/ This is now correct.
class ThrowsDemo
{
static void throwOne() throws IllegalAccessException
{
```

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

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Packages

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

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

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

package *pkg*;

Here, *pkg* is the name of the package. For example, the following statement creates apackage called **MyPackage**.

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly.
More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

A Short Package Example

:

```
// A simple package
package MyPack;
class Balance
{
String name;
double bal;
Balance(String n, double b)
 {
name = n;
bal = b;
}
void show()
{
if(bal<0)
System.out.print("--> ");

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

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

```
java MyPack.AccountBalance
```

Access Protection

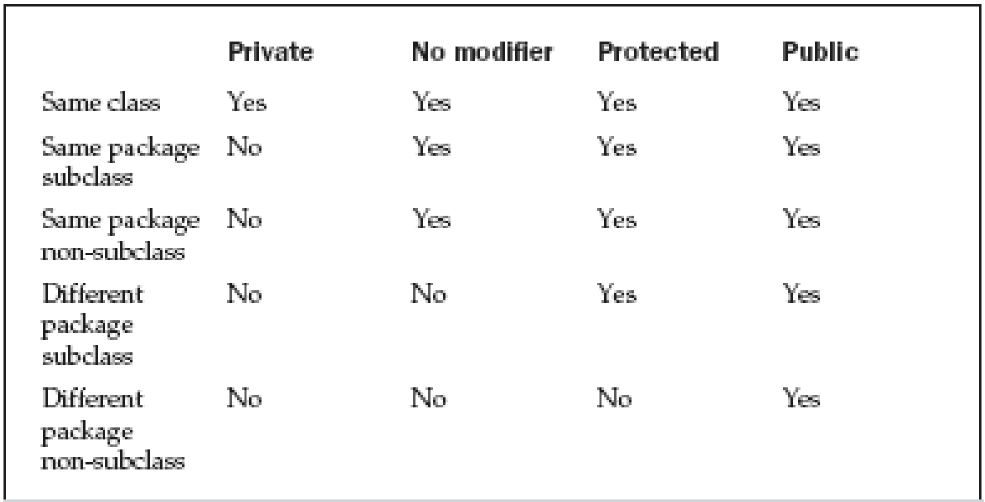
Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java’s smallest unit of abstraction. Because of the interplay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

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

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

While Java’s access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

Table 9-1 applies only to members of classes. A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.



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

Table 9-1. *Class Member Access*

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use.
For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.
In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

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

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

```
import java.io.*;
```

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely

no effect on the run-time performance or size of your classes.
All of the standard Java classes included with Java are stored in a package called **java**.

when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

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

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

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

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

Threads and Life cycle of thread (Multithreading)

A thread is a smallest unit of code in a program which is dispatched by the scheduler. A thread is similar to the program having the basic code. It is also known as light weight and the program is a heavy weight process. A thread is a similar to a program that has a single flow of control. It has a beginning, a body, and, an end and executes commands sequentially. Every program will have at least one thread.

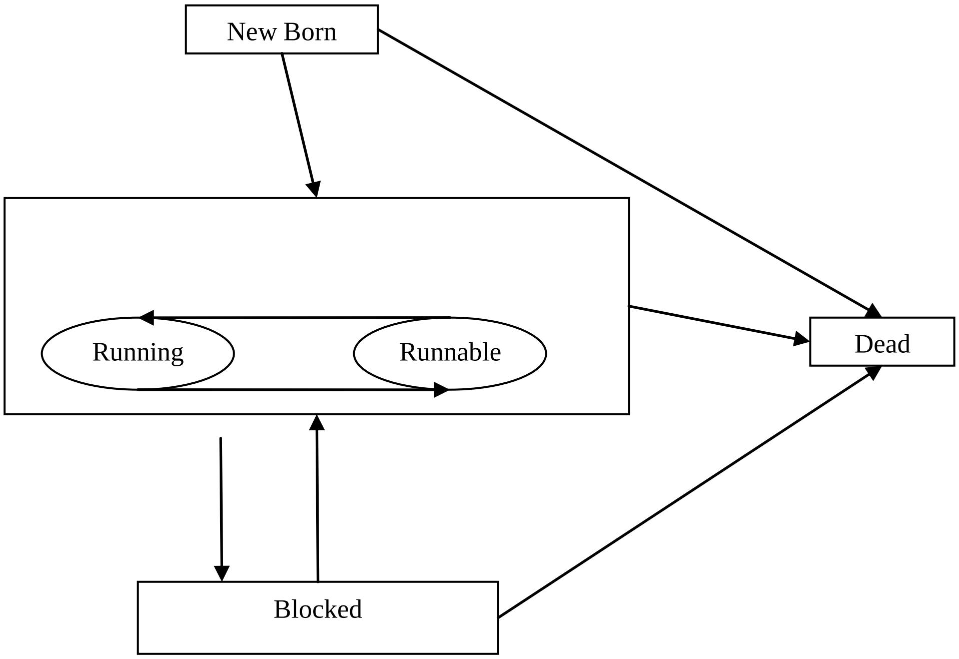
Advantages of thread

1. They can be created easily.
2. Requires fewer overheads.
3. Inter process communication is faster.
4. Context switching is faster
5. Maximum use of CPU time

Life Cycle of thread

During the life time of a thread, there are may states it can enter:

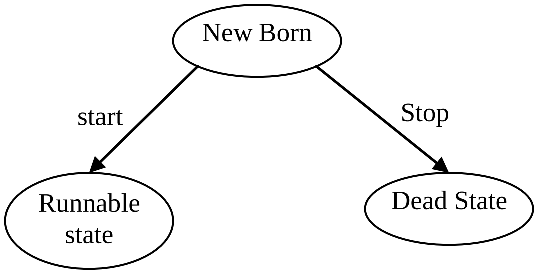
2. New born state
3. Run able state
4. Running state
5. Blocked state
6. Dead state



1. new born state

when we create a thread object, the thread is born and is said to be a new born state. The thread is not yet scheduled for running. At this stage it can do only one of the following:

- schedule it for running using start() method
- kill it using stop() methods



2. Run able state

Run able state means that the thread is ready for execution and is waiting for the availability of the processor. That is the thread has joined the queue of threads that are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round robin fashion. That means first-come-first-serve manner.

3. Running state

Running means that the processor has given its time to the thread for its execution .the thread rums until it relinquishes control on its own or it is pre-empted by a higher priority thread. A running thread may relinquish its control in one of the following situations:

- it has been suspended using suspend() methods
- it has been made to sleep. We can put a thread to sleep for a specific time period using sleep(time) method.
- It has been told to wait until some event occurs. This is done by wait() method.

4. Blocked state:

A thread is said to be blocked when it is prevented for entering into run able state and subsequently the running state. This happens when the thread is suspended, sleeping or waiting in order to satisfying some requirements. A blocked thread is considered to be “not run able “, but not dead, therefore fully qualified to run again.

5. Dead State

Every thread has a life cycle. A running thread ends its life when it has completed executing run() method. It is a natural death. However, we can kill it by sending the stop massage to it at any state thus causing premature death to it. A thread can be killed as soon as it is born or while it is running, or even when it is blocked state.

The Thread Class and the Runnable Interface

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

The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

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

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

```
class a extends Thread
{
public void run()
{
int i;
for(i=1;i<10;i++)
{
System.out.println("in A" +i);
System.out.println("finished");
}
} }
class b extends Thread
{
public void run()
{

int i;
for(i=1;i<10;i++)
{
System.out.println("in b" +i);
System.out.println("finished");
}
}
}
class c extends Thread
{
public void run()
{
int i;
for(i=1;i<10;i++)
{
System.out.println("in c" +i);
System.out.println("finished");
}
}
}
class resu
{
public static void main(String arg[])
{
a x=new a();
b y=new b();
c z=new c();
x.start();
y.start();
z.start();
System.out.println("MaIN FINISHED");
}
}
```

Implementing Runnable

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

public void run( )

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

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

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

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

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

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

void start( )

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

```
class NewThread implements Runnable
{
Thread t;

 NewThread()
 {
 t=new Thread(this,"Demo Thread");
 System.out.println("Child Thread:" +t);
 t.start();
 }

 public void run()
 {
  try
  {
   for(int i=5;i>0;i--)
    {
     System.out.println("child Thread: "+i);
     Thread.sleep(500);
    }
  }
  catch(InterruptedException e)
   {
    System.out.println("Child interrupted");

   }
  System.out.println("Exiting child thread");

 }
} //end of class NewThread.......
```

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

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area( )** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method.*

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. 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**.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// 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.

You will see this feature put to use in the next example.
Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an undefined two-dimensional figure, the following version of the program declares **area( )** as abstract inside **Figure**. This, of course, means that all classes derived from **Figure** must override **area( )**.

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

As the comment inside **main( )** indicates, it is no longer possible to declare objects of type **Figure**, since it is now abstract. And, all subclasses of **Figure** must override **area( )**. To prove this to yourself, try creating a subclass that does not override **area( )**. You will receive a compile-time error.
Although it is not possible to create an object of type **Figure**, you can create a reference variable of type **Figure**. The variable **figref** is declared as a reference to **Figure**, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

Events (Members: getSource(),getActionCommand(), getWhen())

ActionEvent (object is clicked)

Listener interface: ActionListener

Methods : actionPerformed(…)

ComponentEvent (size, visibility, position of object is changed)

Listener interface : Componentlistener

Methods: component Resize(), componentMoved(), componentShown(). componentHidden()

MouseEvent (clicked,dragged,enter,exited,moved,pressed,released,wheel)

Listener interface: MouseListener

Methods: mouseClicked(), mouseEntered(), moueExited(), mousePressed(), mouseReleased()

TextEvent (text in textfield etc is changed)

Listener interface: TextListener

Methods: textValueChanged()

WindowEvent (window activated,deavticated, closed, closing…)

Listener Interface: WindowListener

Methods: windowActivated(),windowClosed(), windowClosing(), windoeDeactivated(), windowOpened(), windowiconified(), windowDeiconified()

KeyEvent (key is pressed, released, keyTyped)

Listener interface: KeyListener

Methods: keyPressed(), keyReleased(), keyTyped()

FocusEvent (focus)

Litener interface : FocusListener

Methods: focusGained(), focusLost()