UNIT-I

OPERATORS:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

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

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0

++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20, then:

Operator	**Description**	**Example**
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Operator	**Description**	**Example**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000

>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	**Description**	**Example**
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

There are following assignment operators supported by Java language:

Operator	**Description**	**Example**
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators:

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

This would produce the following result:

```
Value of b is : 30
Value of b is : 20
```

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is written as:

```
( Object reference variable ) instanceof  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is the example:

```
public class Test {

   public static void main(String args[]){
      String name = "James";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle
{
public class Car extends Vehicle {
  public static void main(String args[]){
    Vehicle a = new Car();
    boolean result =  a instanceof Car;
    System.out.println( result );
  }
}
```

This would produce the following result:

```
true
```

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right

Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

PROGRAM CONTROL STATEMENTS:

Control statements are the statements which alter the flow of execution and provide better control to the programmer on the flow of execution. In Java control statements are categorized into selection control statements, iteration control statements and jump control statements.

- **Java's Selection Statements:**

Java supports two selection statements: if and switch. These statements allow us to control the flow of program execution based on condition.

if Statement:

if statement performs a task depending on whether a condition is true or false.

Syntax: 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.

Program : Write a program to find biggest of three numbers.

```
//Biggest of three numbers

class BiggestNo

{  public static void main(String args[])

  {  int a=5,b=7,c=6;

    if ( a > b && a>c)

      System.out.println ("a is big");

    else if ( b > c)

      System.out.println ("b is big");

    else

      System.out.println ("c is big");

  }

}
```

Output:

```
D:/kumar>javac BiggestNo.java

D:/kumar>java BiggestNo

b is big
```

<u>Switch Statement:</u>

When there are several options and we have to choose only one option from the available ones, we can use switch statement.

Syntax: switch (expression)

```
    {  case value1:  //statement sequence

         break;

      case value2:  //statement sequence
```

```
          break;
        ……………..
      case valueN:  //statement sequence
          break;
      default:  //default statement sequence
    }
```

Here, depending on the value of the expression, a particular corresponding case will be

executed.

Program : Write a program for using the switch statement to execute a particular task depending on color value.

```
//To display a color name depending on color value
class ColorDemo
{  public static void main(String args[])
  {  char color = 'r';
    switch (color)
    {  case 'r': System.out.println ("red");     break;
      case 'g': System.out.println ("green");  break;
      case 'b': System.out.println ("blue");     break;
      case 'y': System.out.println ("yellow");  break;
      case 'w': System.out.println ("white");  break;
      default: System.out.println ("No Color Selected");
    }
  }
}
```

Output:

```
D:/kumar>javac ColorDemo.java
D:/kumar>java ColorDemo
```

red

<u>Java's Iteration Statements</u>:

Java's iteration statements are for, while and do-while. These statements are used to repeat same set of instructions specified number of times called loops.

A loop repeatedly executes the same set of instructions until a termination condition is met.

<u>while Loop:</u>

while loop repeats a group of statements as long as condition is true. Once

the condition is false, the loop is terminated. In while loop, the condition is tested first; if

it is true, then only the statements are executed. while loop is called as entry control loop.

Syntax: while (condition)

```
    {
      statements;
    }
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main(String args[])
  {  int i=1;
    while (i <= 20)
    {  System.out.print (i + "\t");
      i++;
    }
  }
}
```

Output:

```
D:/kumar>javac Natural.java

D:/kumar>java Natural

1       2       3       4       5       6       7       8       9       10

11      12      13      14      15      16      17      18      19      20
```

do-while Loop:

do…while loop repeats a group of statements as long as condition is true. In do...while loop, the statements are executed first and then the condition is tested. do…while loop is also called as exit control loop.

Syntax: do

```
   {
     statements;
   } while (condition);
```

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.

class Natural1

{  public static void main(String args[])

  {  int i=1;

   do

   {  System.out.print (i + "\t");

     i++;

   } while (i <= 20);

  }

}
```

Output:

```
D:/kumar>javac Natural1.java

D:/kumar>java Natural1

1       2       3       4       5       6       7       8       9       10
```

```
11      12      13      14      15      16      17      18      19      20
```

for Loop:

The for loop is also same as do…while or while loop, but it is more compact syntactically. The for loop executes a group of statements as long as a condition is true.

Syntax: for (expression1; expression2; expression3)

```
   {    statements;
   }
```

Here, expression1 is used to initialize the variables, expression2 is used for condition checking and expression3 is used for increment or decrement variable value.

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural2
{  public static void main(String args[])
  {  int i;
    for (i=1; i<=20; i++)
      System.out.print (i + "\t");
  }
}
```

Output:

```
D:/kumar>javac Natural2.java
D:/kumar>java Natural2
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
D:/kumar>
```

Java's Jump Statements:

Java supports three jump statements: break, continue and return.

These statements transfer control to another part of the program.

break:

Ø break can be used inside a loop to come out of it.

Ø break can be used inside the switch block to come out of the switch block.

Ø break can be used in nested blocks to go to the end of a block. Nested blocks represent a block written within another block.

Syntax: break;   (or) break label;//here label represents the name of the block.

Program : Write a program to use break as a civilized form of goto.

```
//using break as a civilized form of goto

class BreakDemo

{  public static void main (String args[])

  {  boolean t = true;

    first:

{

  second:

{

        third:

{

            System.out.println (“Before the break”);

            if (t) break second; // break out of second block

              System.out.println (“This won’t execute”);

          }

          System.out.println (“This won’t execute”);

        }

        System.out.println (“This is after second block”);

      }
    }
  }
```

Output:

```
D:/kumar>javac BreakDemo.java
D:/kumar>java BreakDemo
Before the break
This is after second block
D:/kumar>
```

__**continue:**__

This statement is useful to continue the next repetition of a loop/ iteration. When continue is executed, subsequent statements inside the loop are not executed.

Syntax: continue;

Program : Write a program to generate numbers from 1 to 20.

```
//Program to generate numbers from 1 to 20.
class Natural
{  public static void main (String args[])
  {  int i=1;
    while (true)
    {  System.out.print (i + "\t");
      i++;
      if (i <= 20 )
        continue;
      else
        break;
    }
  }
}
```

Output:

```
D:/kumar>javac Natural.java

D:/kumar>java Natural

1       2       3       4       5       6       7       8       9       10

11      12      13      14      15      16      17      18      19      20

D:/kumar>
```

return statement:

Ø return statement is useful to terminate a method and come back to the calling method.

Ø return statement in main method terminates the application.

Ø return statement can be used to return some value from a method to a calling method.

Syntax: return; (or)

return value; // value may be of any type

Program : Write a program to demonstrate return statement.

```
//Demonstrate return

class ReturnDemo

{  public static void main(String args[])

  {  boolean t = true;

    System.out.println (“Before the return”);

    if (t)

      return;

    System.out.println (“This won’t execute”);

  }

}
```

Output:

```
D:/kumar>javac ReternDemo.java

D:/kumar>java ReturnDemo

Before the return
```

Introducing Classes:

Concepts of classes, objects:

The class is at the core of Java. **It is the logical construct** upon which the entire Java language is built because it defines the shape and nature of an object.

As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java.

Class Fundamentals:

The classes created in the preceding chapters primarily exist simply to encapsulate the main( )method, which has been used to demonstrate the basics of the Java syntax.

Perhaps the most important thing to understand about a class is that it defines a new **data type,** defined, this new type can be used to **create objects of that type**.

Thus, **a class is a template for an object**, and **an object is an instance of a class**.

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

The General Form of a Class:

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data. While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data.

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

```
class classname {

type instance-variable1;

type instance-variable2;

// ...

type instance-variableN;

type methodname1(parameter-list) {

// body of method

}

type methodname2(parameter-list) {

// body of method

}

// ...

type methodnameN(parameter-list) {

// body of method

}

}
```

The data, or variables, defined within a class are called **instance variables**. The code is contained within methods. Collectively, the methods and variables defined within a class are called **members of the class.** In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, as a general rule, it is the methods that determine how a class' data can be used.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

All methods have the same general form as main( ), which we have been using thus far. However, most methods will not be specified as static or public.

Notice that the general form of a class does not specify a main( )method. Java classes do not need to have a main( ) method. You only specify one if that class is the starting point for your program. Further, applets don’t require a main( ) method at all.

Declaring Member Variables

There are several kinds of variables:

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

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

A Simple Class:

Let’s begin our study of the class with a simple example. Here is a class called Box that defines

three instance variables: width, height, and depth.

```
// This program declares two Box objects.

class Box {
```

```
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
mybox2.width = 3;
mybox2.height = 6;
mybox2.depth = 9;
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume is 3000.0

Volume is 162.0
```

- As you can see, mybox1's data is completely separate from the data contained in mybox2.
- As stated, a class defines a new type of data. In this case, the new data type is called Box.

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

- After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement.
- To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

For example, to assign the width variable of mybox the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100.

- In general, you use the dot operator to access both the instance variables and the methods within an object.
- You should call the file that contains this program BoxDemo2.java, because the main() method is in the class called BoxDemo2, not the class called Box.
- When you compile this program, you will find that two .class files have been created, one for Box and one for BoxDemo2.
- The Java compiler automatically puts each class into its own .class file. It is not necessary for both the Box and the BoxDemo2 class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo2.java, respectively.

To run this program, you must execute BoxDemo2.class. When you do, you will see the

following output:

```
Volume is 3000.0

Volume is 162.0
```

- As stated earlier, each object has its own copies of the instance variables.

This means that if you have two Box objects, each has its own copy of depth, width, and height. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another.

Declaring Objects

An object is an instance of a class. An object is known by a name and every object contains a state. The state is determined by the values of attributes (variables). The state of an object can be changed by calling methods on it. The sequence of state changes represents the behavior of the object.

An object is a software entity (unit) that combines a set of data with a set of operations to manipulate that data.

As just explained**, when you create a class, you are creating a new data type**. You can use this type to declare objects of that type.

However, **obtaining objects of a class is a two-step process**.

- **First,** you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- **Second**, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator.

- **Declaration**: The code set in variable declarations that associate a variable name with an object type.
- **Instantiation**: The new keyword is a Java operator that creates the object.

- **Initialization**: The new operator is followed by a call to a constructor, which initializes the new object.

The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.

In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

Box mybox = new Box();

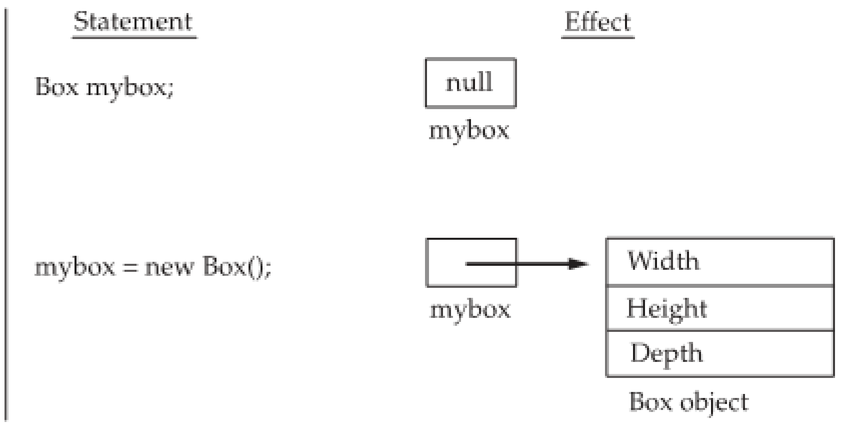
This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

mybox = new Box(); // allocate a Box object

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object.

Any attempt to use mybox at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object. The effect of these two lines of code is depicted in Figure.

FIGURE 6-1 Declaring an object of type **Box**



Assigning Object Reference Variables:

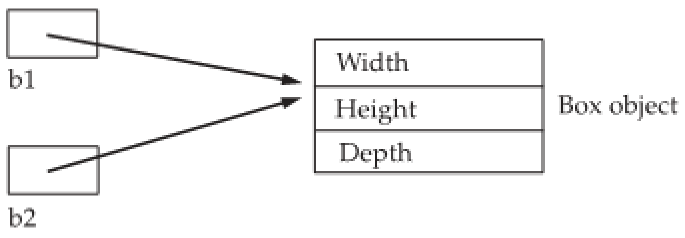
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();

Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although b1 and b2 both refer to the same object, they are not linked in any other way.

For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2. For example:

```
Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.

REMEMBER: When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

Methods :

Here is an example of a typical method declaration:

```
public double calculateAnswer(double width, int numberofitems,
                              double length, double height) {
   //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

Modifiers, return types, and parameters will be discussed later in this lesson. Exceptions are discussed in a later lesson.

Definition: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized.

Here are some examples:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

Returning a Value from a Method

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

returnValue;

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The getArea() method in the Rectangle Rectangle class that was discussed in the sections on objects returns an integer:

```
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
```

This method returns the integer that the expression width*height evaluates to.

Parameters:

Parameters refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

Constructors:

It can be tedious to initialize all of the variables in a class each time an instance is created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.

You can rework the Box example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace setDim( ) with a constructor.

Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the dimensions of a box.*/

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box() {

System.out.println("Constructing Box");
```

```
width = 10;

height = 10;

depth = 10;

}

// compute and return volume

double volume() {

return width * height * depth;

}

}

class BoxDemo6 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box();

Box mybox2 = new Box();

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();
```

```
System.out.println("Volume is " + vol);

}

}
```

When this program is run, it generates the following results:

```
Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0
```

As you can see, both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both mybox1 and mybox2 will have the same volume. The println( ) statement inside Box( ) is for the sake of illustration only.

Parameterized Constructors

While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor.

As you can probably guess, this makes them much more useful. For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created.

```
/* Here, Box uses a parameterized constructor to initialize the dimensions of a box.*/

class Box {
```

```
double width;

double height;

double depth;

// This is the constructor for Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// compute and return volume

double volume() {

return width * height * depth;

}

}

class BoxDemo7 {

public static void main(String args[]) {

// declare, allocate, and initialize Box objects

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box(3, 6, 9);

double vol;
// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}
```

The output from this program is shown here:

```
Volume is 3000.0

Volume is 162.0
```

As you can see, each object is initialized as specified in the parameters to its constructor.

For example, in the following line,

```
Box mybox1 = new Box(10, 20, 15);
```

the values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object. Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

Overloading Constructors:

Since Box( ) requires three arguments, it's an error to call it without them.

This raises some important questions.

What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions?

As the Box class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described. Here is a program that contains an improved version of Box that does just that:

```
/* Here, Box defines three constructors to initialize

the dimensions of a box various ways.

*/

class Box {

double width;

double height;

double depth;

// constructor used when all dimensions specified

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions specified

Box() {

width = -1;  // use -1 to indicate

height = -1; // an uninitialized

depth = -1;  // box

}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0

Volume of mybox2 is -1.0

Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the parameters specified when new is executed.

The this Keyword
Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. To better understand what **this** refers to, consider the following version of **Box( )**:

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

This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object.

2.6 Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically.

Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector so we don't need to make extra efforts.

Finalize () method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

```
        protected void finalize(){}
```

A Stack Class

As we have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class. In a sense, a class is like a "data engine." No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class.
To see a practical application, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out
ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop*. To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism.
Here is a class called **Stack** that implements a stack for up to ten integers:

```
// This class defines an integer stack that can hold 10 values
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
```

Stack class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack( )** constructor initializes **tos** to –1, which indicates an empty stack. The method **push( )** puts an item on the stack. To retrieve an item, call **pop( )**. Since access to the stack is through **push( )** and **pop( )**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push( )** and **pop( )** would remain the same.
The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
```

```
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

As you can see, the contents of each stack are separate.
One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

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

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

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Note: Overloaded methods should be used sparingly, as they can make code much less readable.

parameter passing:

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

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

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

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

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

For example, consider the following program:

```
// Primitive types are passed by value.

class Test {

void meth(int i, int j) {

i *= 2;

j /= 2;

}

}

class CallByValue {

public static void main(String args[]) {

Test ob = new Test();

int a = 15, b = 20;

System.out.println("a and b before call: " +

a + " " + b);

ob.meth(a, b);

System.out.println("a and b after call: " +

a + " " + b);

}

}
```

The output from this program is shown here:

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

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

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

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.

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

For example, consider the following program:

```
// Objects are passed by reference.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// pass an object

void meth(Test o) {

o.a *=  2;

o.b /= 2;

}

}

class CallByRef {

public static void main(String args[]) {

Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +ob.a + " " + ob.b);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " +ob.a + " " + ob.b);

}

}
```

This program generates the following output:

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

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

As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Note: When a primitive type is passed to a method, it is done by use of call-by-value. Objects are implicitly passed by use of call-by-reference.

Recursion:

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

A method that calls itself is said to be recursive.

The classic **example** of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

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

```
// A simple example of recursion.

class Factorial {

// this is a recursive method

int fact(int n) {

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

The output from this program is shown here:

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

Access Modifiers:

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc.

1.private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

2.default:

If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package.

3.protected:

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

4.public:

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	**within class**	**within package**	**outside package by subclass only**	**outside package**
Private	Y	N	N	N

Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

UNDERSTANDING STATIC:

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

why java main method is static?

Because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

Example of static method, variable:

```
//Program of changing the common property of all objects(static field)
.

  class Student9{
     int rollno;
     String name;
     static String college = "ITS";

     static void change(){
     college = "BBDIT";
     }

     Student9(int r, String n){
     rollno = r;
     name = n;
     }

     void display (){System.out.println(rollno+" "+name+" "+college);
}

    public static void main(String args[]){
    Student9.change();

    Student9 s1 = new Student9 (111,"Karan");
    Student9 s2 = new Student9 (222,"Aryan");
    Student9 s3 = new Student9 (333,"Sonoo");

    s1.display();
    s2.display();
    s3.display();
    }
}
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

```
Output:static block is invoked
       Hello main
```

2.8 Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Nested classes are divided into two categories: static and non-static. Nested classes that are declared static are called *static nested classes*. Non-static nested classes are called *inner classes*.

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

STATIC NESTED CLASSES:

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Non Static Nested Classes(Inner class):

Non static nested class is also known as inner class. It has access to all variables and methods of outer class and may refer to them directly. But the reverse is not true, that is outer class cannot directly access members of inner class. One more thing is an inner class must created and instantiated within the scope of outer class only.

Using Command-Line Arguments

Sometimes we will want to pass information into a program when we 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 a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on. 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
```

Varargs: Variable-Length Arguments

Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*. A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.
Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided. In this situation, it would be convenient to pass only the arguments to which the defaults did not apply.
Prior to JDK 5, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.
In cases where the maximum number of potential arguments was larger, or unknowable, second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach is illustrated by the following program:

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
static void vaTest(int v[]) {
System.out.print("Number of args: " + v.length + " Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how an array must be created to
// hold the arguments.
int n1[] = { 10 };
int n2[] = { 1, 2, 3 };
int n3[] = { };
vaTest(n1); // 1 arg
vaTest(n2); // 3 args
vaTest(n3); // no args
}
}
```

The output from the program is shown here:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

In the program, the method **vaTest( )** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest( )** to take an arbitrary
number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling **vaTest( )**. The varargs feature offers a simpler, better option.
A variable-length argument is specified by three periods (**...**). For example, here is how **vaTest( )** is written using a vararg:

```
static void vaTest(int ... v) {
```

This syntax tells the compiler that **vaTest( )** can be called with zero or more arguments. As a result, **v** is implicitly declared as an array of type **int[ ]**. Thus, inside **vaTest( )**, **v** is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

```
// Demonstrate variable-length arguments.
class VarArgs {
// vaTest() now uses a vararg.
static void vaTest(int ... v) {
System.out.print("Number of args: " + v.length + " Contents: ");
for(int x : v)
System.out.print(x + " ");
System.out.println();
}
public static void main(String args[])
{
// Notice how vaTest() can be called with a
// variable number of arguments.
vaTest(10); // 1 arg
vaTest(1, 2, 3); // 3 args
vaTest(); // no args
}
}
```

The output from the program is the same as the original version.