**History of Java:**

Before starting to learn Java, let us plunge into its history and see how the language originated. In 1990, Sun Microsystems Inc. (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote. This project was called *Stealth Project* but later its name was changed to *Green Project*.

In January of 1991, Bill Joy, **James Gosling**, Mike Sheradin, Patrick Naughton, and several others met in Aspen, Colorado to discuss this' project. Mike Sheradin was to focus on business development; Patrick Naughton was to begin work on the graphics system; and James Gosling was to identify the proper programming language for the project. Gosling thought C and C++ could be used to develop the project.

But the problem he faced with them is that they were system dependent languages and hence could not be used on various processors, which the electronic devices might use. So he started developing a new language, which was completely system independent. This language was initially called *Oak.* Since this name was registered by some other company, later it was changed to *Java.*

**Why the name Java?** James Gosling and his team members were consuming a lot of tea while developing this language. They felt that they were able to develop a better language because of the good quality tea they had consumed. So the tea also had its own role in developing this language and hence, they fixed the name for the language as *Java.* Thus, the symbol for *Java* is tea cup and saucer.


- *James Gosling was born on **May 19, 1955**.*
- *James Gosling received a Bachelor of Science from the **University of Calgary** and his M.A. and Ph.D. from **Carnegie Mellon University**. He built a multi-processor version of Unix for a 16-way computer system while at Carnegie Mellon University, before joining Sun Microsystems.*

By September of 1994, Naughton and Jonathan Payne started writing *WebRunner-a* Java-based Web browser, which was later renamed as *HotJava.* By October 1994, HotJava was stable and was demonstrated to Sun executives. HotJava was the fIrst browser, having the capabilities of executing *applets,* which are programs designed to run dynamically on Internet. This time, Java's potential in the context of the World Wide Web was recognized.

Sun formally announced Java and HotJava at SunWorld conference in 1995. Soon after, Netscape Inc. announced that it would incorporate Java support in its browser Netscape Navigator. Later, Microsoft also announced that they would support Java in their Internet Explorer Web browser, further solidifying Java's role in the World Wide Web. On **January 23rd 1996, JDK 1.0** version was released**.** Today more than 4 million developers use Java and more than 1.7 5 billion devices run Java. Thus, Java pervaded the world.

**Features of Java:**

- **Simple:** Java is a simple programming language. Rather than saying that this is the feature of Java, we can say that this is the design aim of Java. JavaSoft (the team who developed Java is called with this name) people maintained the same syntax of C and C++ in Java, so that a programmer who knows C or C++ will find Java already familiar.

- **Object-oriented:** Java is an object oriented programming language. This means Java programs use objects and classes. What is an object? An object is anything that really exists in the world and can be distinguished from others.

- **Distributed:** Information is distributed on various computers on a network. Using Java, we can write programs, which capture information and distribute it to the clients. This is possible because Java can handle the protocols like TCP/IP and UDP.
- **Robust:** Robust means *strong.* Java programs are strong and they don't crash easily like a C or C++ program. There are two reasons -for this. Firstly, Java has got excellent inbuilt exception handling features.
- **Secure:** Security problems like eavesdropping, tampering, impersonation, and virus threats can be eliminated or minimized by using Java on Internet.
- **System independence:** Java's byte code is not machine dependent. It can be run on any machine with any processor and any operating system.
- **Portability:** If a program yields the same result on every machine, then that program is called *portable.* Java programs are portable. This is the result of Java's *System independence* nature.
- **Interpreted:** Java programs are compiled to generate the byte code. This byte code can be downloaded and interpreted by the interpreter in JVM. If we take any other language, only an interpreter or a compiler is used to execute the programs. But in Java, we use .both compiler and interpreter for the execution.
- **High Performance:** The-problem with interpreter inside the JVM is that it is slow. Because of this, Java programs used to run slow. To overcome this problem, along with the interpreter, JavaSoft people have introduced JIT (Just In Time) compiler, which enhances the speed of execution. So now in JVM, both interpreter and JIT compiler work together to run the program.
- **Multithreaded:** A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different blocks of code. Creating multiple threads is called 'multithreaded'.
- **Scalability:** Java platform can be implemented on a wide range of computers with varying levels of resources-from embedded devices to mainframe computers. This is possible because Java is compact and platform independent.
- **Dynamic:** Before the development of Java, only static text used to be displayed-in the browser. But when James Gosling demonstrated an animated atomic molecule where the rays are moving and stretching, the viewers are dumbstruck. This animation was done using an *applet* program, which are the dynamically interacting programs on Internet.

**Java Virtual Machine:**

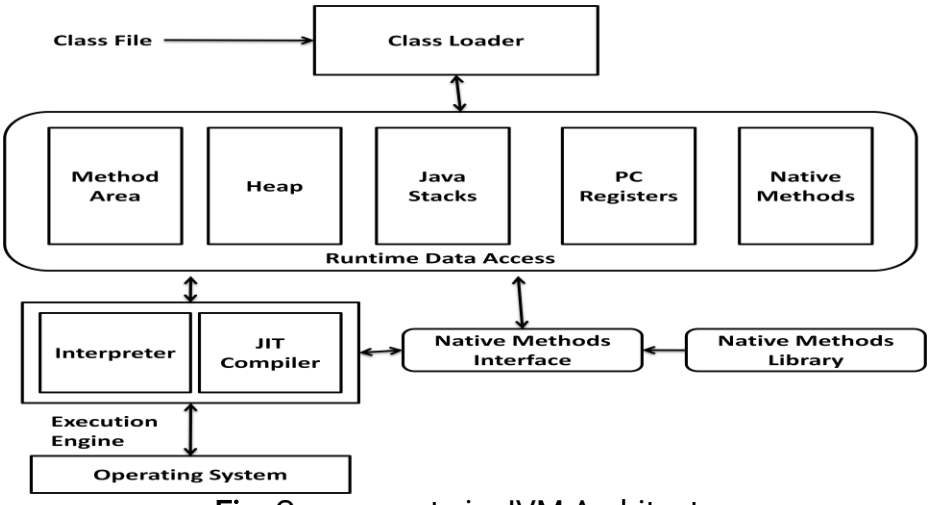



**Fig:** Components in JVM Architecture

First of all, the .java program is converted into a .class file consisting of byte code instructions by the java compiler. Remember, this java compiler is outside the JVM: Now this
.class file is given to the JVM. In JVM, there is a module (or program) called *class loader sub system,* which performs the following functions:

- First of all, it loads the .class file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program.

This memory is divided into 5 parts, called *run time data areas,* which contain the data and results while running the program. These areas are as follows:

- **Method area:** Method area is the memory block, which stores the class code, code of the variables, and code of the methods in the Java program. (Method means functions written in a class)
- **Heap:** This is the area where objects are created. Whenever JVM loads a class, a method and a heap area are immediately created in it.
- **Java Stacks:** Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java stacks.
- **PC (Program Counter) registers:** These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.
- **Native method stacks:** Java methods are executed on Java stacks. Similarly, native methods (for example *C/C++* functions) are executed on Native method stacks. To execute the native methods, generally native method libraries (for example *C/C++* header files) are required. These header files are located and connected to JVM by a program, called *Native method interface.*

Execution engine contains interpreter and JIT (Just In Time) compiler, which are responsible for converting the byte code instructions into machine code so that the processor will execute them. Most of the JVM implementations use both the interpreter and JIT compiler simultaneously to convert the byte code. This technique is also called *adaptive optimizer.* Generally, any language (like C/C++, Fortran, COBOL, etc.) will use either an interpreter or a compiler to translate the source code into a machine code. But in JVM, we got interpreter and JIT compiler both working at the same time on byte code to translate it into machine code.

**Java Program Structure:**

- Documentation Section
- Package Statement
- Import Statements
- Interface Statement
- Class Definition
- Main Method Class
    - o Main Method Definition

| | |
|---|---|
| Documentation Section | It comprises of a comment line which gives the names program, the programmer's name and some other brief details. In addition to the 2 other styles of comments i.e. /**/ and //, Java also provides another style of comment i.e. /**….*/ |
| Package statement | The first statement allowed in Java file is the Package statement which is used to declare a package name and it informs the compiler that the classes defined within the program belong to this package. package package_name; |
| Import statements | The next is the number of import statements, which is equivalent to<br>#include statement in C++.<br>Example:<br><br>import calc.add; |
| Interface statement | Interfaces are like class that includes a group of method declarations. This is an optional section and can be used only when programmers want to implement multiple inheritance(s) within a program. |
| Class Definition | A Java program may contain multiple class definitions. Classes are the main and important elements of any Java program. These classes are used to plot the objects of real world problem. |
| Main Method Class | Since every Java stand alone program requires a main method as the starting point of the program. This class is essentially a part of Java program. A simple Java program contains only this part of the program. |

Sample Code of Java "Hello Java"
Program Example:

```
// Hello java program
import java.lang.System;
import java.lang.String;
class Hello
{
   public static void main(String[ ] args)
   {
      System.out.println("Hello Java");
   }
}
```

Program Output:

Hello Java

**Comments:**

- **Single line comments:** These comments are for marking a SL'1gle line as a comment. These comments start with double slash symbol / / and after this, whatever is written till the end of the line is taken as a comment. For example,

```
// This is single line comment
```

- **Multi-line comments:** These comments are used for representing several lines as comments. These comments start with / * and end with * /. In between / * and * /, whatever is written is treated as a comment. For example,

```
/* This is Multi line
comment, Line 2 is here
Line 3 is here */
```

**Importing Classes:**

In C/C++ compiler to include the header file <stdio.h>. What is a header file? A *header file* is a file, which contains the code of functions that will be needed in a program. In other words, to be able to use any function in a program, we must first include the header file containing that function's code in our program. For example, <stdio.h> is the header me that contains functions, like printf (), scanf (), puts (), gets (), etc. So if we want to use any of these functions, we should include this header file in our *C/C++* program.

A similar but a more efficient mechanism, available in case of Java, is that of importing classes. First, we should decide which classes are 'needed in our program. Generally, programmers are interested in two things:

- Using the classes by creating objects in them.
- Using the methods (functions) of the classes.

In Java, methods are available in classes or interfaces. What is an interface? An *interface* is similar to a class that contains some methods. Of course, there is a lot of difference between an interface and a class, which we will discuss later. The main point to be kept in mind, at this stage, is that a class or an interface contains methods. A group of classes and interfaces are contained in a package. A *package* is a kind of directory that contains a group of related Classes and interfaces and Java has several such packages in its library,

Java library
|
Packages
|
Classes and interfaces
|
Method
s

In our first Java program, we are using two classes namely, System and String. These classes belong to a package called java .lang (here lang represents language). So these two classes must be imported into our program, as shown below:

```
import java.lang.System;
import java.lang.String;
```

Whenever we want to import several classes of the same package, we need not write several import statements, as shown above; instead, we can write a single

```
import java.lang.*;
```

statement as:

Here, * means all the classes and interfaces of that package, i.e. java.lang, are imported made available) into our program. In import statement, the package name that we have written acts like a reference to the JVM to search for the classes there. JVM will not copy any code from the classes or packages. On the other hand, when a class name or a

method name is used, JVM goes to the Java library, executes the code there, comes back, and substitutes the result in that place of the program. Thus, the Java program size will not be increased.

After importing the classes into the program, the next step is to write a class, Since Java is purely an object-oriented programming language and we cannot write a Java program without having at least one class or object. So, it is mandatory that every Java program should have at least one class in it, How to write a class? We should use class keyword for this purpose and then write the class name.

```
class
  Hello{ sta
  tements;
```

A class code starts with a { and ends with a }. We know that a class or an object contains variables and methods (functions), 'So we can create any number of variables and methods inside the class. This is our first program, so we will create only one method, i.e. compulsory, main () method.

The main method is written as follows:

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

- Here **args[ ]** is the array name and it is of String type. This means that it can store a group of strings. Remember, this array can also store a group of numbers but in the form of strings only. The values passed to main( ) method are called *arguments.* These arguments are stored into args[ ] array, so the name args[ ] is generally used for it.
- If a method is not meant to return any value, then we should write *void* before that method's name. ***void*** means *no value.* main() method does not return any value, so void should be written before that method's name.
- ***static*** methods are the methods, which can be called and executed without creating the objects. Since we want to call main()method without using an object, we should declare main( ) method as static. Then, how is the main( ) method called and executed? The answer is by using the classname.methodname(). JVM calls main() method using its class name as Hello.main() at the time of running the program.
- JVM is a program written by *JavaSoft* people (Java development team) and main() is the method written by us. Since, main() method should be available to the JVM, it should be declared as ***public***. If we don't declare main() method as public, then. It doesn't make itself available to JVM and JVM cannot execute it:

So, the main()method should always be written as shown here:

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

If at all we want to make any changes, we can interchange public and static and write it as follows:

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

Or, we can use a different name for the string type array and write it as:

**public static void main(String x[])**

**Q)** When main() method is written without String args[args]:

**public static void main ()**

**Ans:** The code will compile but JVM cannot run the code because it cannot recognize the main () method as the method from where it should start execution of the Java program. Remember JVM always looks for

main () method with string type array as parameter.

## print method in java:

In Java, print() method is used to display something on the monitor. So, we can write it as:

**print("Welcome to java");**

But this is not the correct way of calling a method in Java. A method should be called by using objectname .methodname( ). So, to call print() method, we should create an object to the class to which print() method belongs. print() method belongs to PrintStream class. So, we should call print() method by creating an object to PrintStream class as:

**PrintStream obj.print("Welcome to java");**

But as it is not possible to create the object to PrintStream class directly, an alternative is given to us, i.e. ***System.out***. Here, ***System*** is the class name and ***out*** is a static variable in *System* class. *Out* is called a field in *System* class. When we call this field, a PrintStream class object will be created internally. So, we can call the print() method as shown below:

**System.out.print("Welcome to java");**

System.out gives the PrintStream class object. This object, by default, represents the standard output device, i.e. the monitor. So, the string "Welcome to Java" will be sent to the monitor.

```
class Hello
{
  public static void main(String[] args)
  {
        System.out.print("Welcome to java");
  }
}
```

Save the above program in a text editor like *Notepad* with the name Hello.java.
Now, go to System prompt and compile it using *javac* compiler as:

```
javac Hello.java
```

The compiler generates a file called Hello.class that contains byte code instructions. This file is executed by calling the JVM as:

```
java Hello
```

Then, we can see the result.

**Note:** In fact, JVM is written in C language.

**Output:**

```
C:\> javac Hello.java
C:\> java Hello
Welcome to Java
```

**Program:** Write a program to find the sum of addition of two numbers.

```
class Add
{
        public static void main(String args[])
        {
                int x, y;
                x = 27;
                y = 35;
                int  z = x + y;
                System.out.print("The Sum is
                "+z);
        }
```

## Variables:

A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

You must declare all variables before they can be used. Following is the basic form of a variable declaration –

```
DataType variable [ = value][, variable [ = value] ...] ;
```

Here *DataType* is one of Java's datatypes and *variable* is the name of the variable. To declare more than one variable of the specified type, you can use a comma-separated list.

Following are valid examples of variable declaration and initialization in Java – Example

```
int a, b,          // Declares three ints, a, b,
int a = 10, b = 10; // Example of initialization
byte B = 22;       // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a
value of PI.
```

There are three kinds of variables in Java –

a) Local variables
b) Instance variables
c) Class/Static variables

### a) Local Variables

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor, or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

### Example 1

Here, *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

```
class Test {
  void pupAge( )
    { int  age = 0;
    age = age +
    7;
    System.out.println("Puppy age is : " + age);
  }

  public static void main(String
    args[]) { Test test = new Test();
    test.pupAge();
  }
```

This will produce the following result –

```
Puppy age is: 7
```

### b) Instance Variables

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally, it is recommended to make these variables private (access level). However, visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However, within static methods (when instance variables are given accessibility), they should be called using the fully qualified name. *ObjectReference.VariableName*.

### Example 1

```
import java.io.*;
public class Employee
  { public String name;
  public Employee (String empName)
    { name = empName;
  }
  public void printEmp()
    { System.out.println("name : " + name
     );
  }
  public static void main(String args[]) {
    Employee empOne = new
    Employee("Shobana"); empOne.printEmp();
  }
```

This will produce the following result – Output

```
name : Shobana
```

### c) Class/Static Variables

- Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final, and static. Constant variables never change from their initial value.

- Static variables are stored in the static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. Static variables can be accessed by calling with the class name *ClassName.VariableName*.

**Example 1**

```
import java.io.*;
public class Employee
  { private static double
  salary;
  public static void main(String
    args[]) { salary = 1000;
    System.out.println("average salary:" + salary);
  }
```

This will produce the following result –

```
average salary:1000
```

**Identifiers:**

*Identifiers* are the names of variables, methods, classes, packages and interfaces. In the Hello program, *Hello, String, args, main* and *print* are identifiers. There are a few rules and conventions related to the naming of variables.

The rules are:

1. Java variable names are case sensitive. The variable name money is not the same as Money or MONEY.
2. Java variable names must start with a letter, or the $ or _ character.
3. After the first character in a Java variable name, the name can also contain numbers (in addition to letters, the $, and the _ character).
4. Variable names cannot be equal to reserved key words in Java. For instance, the words int or forare reserved words in Java. Therefore you cannot name your variables int or for.

Here are a few valid Java variable name examples:

| | | | |
|---|---|---|---|
| myvar | myVar | MYVAR | _myVar |
| $myVar | myVar1 | myVar_1 | |

## Data types:

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java –

a) Primitive Data Types
b) Reference/Object Data Types

**a) Primitive Data Types**

There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

**Integer Data Types:**

| Data Type | Memory Size |
|---|---|
| byte | 1 byte |
| short | 2 bytes |
| int | 4 bytes |
| long | 8 bytes |

**Float Data Types:**

| Data Type | Memory Size |
|---|---|
| float | 4 byte |
| double | 8 bytes |

**Character Data Types:**

The Character Data type is having 2 bytes of memory size.

**String Data Types:**

A String represents a group of characters, like New Delhi, AP123, etc. The simplest way to create a String is by storing a group of characters into a String type variable as:

**String str="New Delhi";**

Now, the String type variable str contains "New Delhi". Note that any string written directly in a program should be enclosed by using double quotes.

There is a class with the name String in Java, where several methods are provided to perform different operations on strings. Any string is considered as an object of String class.

**b) Reference Datatypes**

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy, etc.
- Class objects and various type of array variables come under reference datatype.
- Default value of any reference variable is null.
- A reference variable can be used to refer any object of the declared type or any compatible type.
- Example: Animal an = new Animal("giraffe");

## Java Literals

A literal is a value that is stored into a variable directly in the program. There are 5 types of literals.

a) Integer Literals
b) Float Literals
c) Character Literals
d) String Literals
e) Boolean Literals

a) Integer Literals

```
class Test {
   public static void main(String[] args)
   {
      int dec = 101; // decimal-form
      literal int oct = 0100; // octal-form
      literal
      int hex = 0xFace; // Hexa-decimal form literal

      System.out.println(dec); //101
      System.out.println(oct); //64
      System.out.println(hex); //64206
   }
```

b) Float Literals

```
class Test {
   public static void main(String[] args)
   {
      double d1 = 123.4;
      double d2 = 1.234e2;   //Same value as d1, but in scientific
      notation float f1 = 123.4f;

      System.out.println(d1);
      //123.4
      System.out.println(d2);
      //123.4
      System.out.println(f1);
```

c) Character Literals

```
public class Test {
   public static void main(String[] args)
   {
                  char   ch1   =
                  'a';  char  ch2
                  =    '\'';    char
                  ch3 = '\n';
                  char ch4 = '\u0015';

                 System.out.println(ch1); //
                  a
                  System.out.println(ch2);
                  //
                  " System.out.println(ch3);
```

d) String Literals

String literals represent objects of String class. For example, Hello, Anil Kumar, AP1201, etc. will come under string literals, which can be directly stored into a String object.

e) Boolean Literals

Boolean literals represent only two values-true and false, It means we can store either true or false into a boolean type variable.

## 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 |
|---|---|---|
| == (equal to) | Checks if the values of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != (not equal to) | Checks if the values of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > (greater than) | 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. |
| < (less than) | 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. |
| >= (greater than or equal to) | 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. |
| <= (less than or equal to) | 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

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

| Operator | Description | Example |
|---|---|---|
| & (bitwise and) | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| (bitwise or) | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ (bitwise XOR) | 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 |
| ~ (bitwise complimen t) | 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. |
| << (left shift) | 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 |
| >> (right shift) | 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 |
| >>> (zero fill right shift) | 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 |
|---|---|---|
| && (logical and) | Called Logical AND operator. If both the operands are non-zero, then the condition becomes true. | (A && B) is false |
| \|\| (logical or) | Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true. | (A \|\| B) is true |

| | | |
|---|---|---|
| ! (logical not) | 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

Following are the 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 |

### Miscellaneous 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
```

**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 will produce the following result −

**Output**

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

**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 an example **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 will produce the following result −

**Output**

```
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 −

**Example**

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

```
  }
}
```

This will produce the following result −

**Output**

```
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 toright |
| 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 |

## Primitive Type Conversion and casting:

Java supports two types of castings – primitive data type casting and reference type casting. Reference type casting is nothing but assigning one Java object to another object. It comes with very strict rules and is explained clearly in Object Casting. Now let us go for data type casting.

Java data type casting comes with 3 flavors.

a) Implicit casting
b) Explicit casting
c) Boolean casting.

**a) Implicit casting (widening conversion)**

A data type of lower size (occupying less memory) is assigned to a data type of higher size. This is done implicitly by the JVM. The lower size is widened to higher size. This is also named as automatic type conversion.

Examples:

```
int x = 10;                // occupies 4 bytes
double y = x;              // occupies 8
bytes System.out.println(y);            //
```

In the above code 4 bytes integer value is assigned to 8 bytes double value.

**b) Explicit casting (narrowing conversion)**

A data type of higher size (occupying more memory) cannot be assigned to a data type of lower size. This is not done implicitly by the JVM and requires explicit casting; a casting operation to be performed by the programmer. The higher size is narrowed to lower size.

```
double x = 10.5;          // 8 bytes
int y = x;                // 4 bytes ; raises compilation error
```

In the above code, 8 bytes double value is narrowed to 4 bytes int value. It raises error. Let us explicitly type cast it.

```
double x =
10.5; int y =
(int) x;
```

The double x is explicitly converted to int y. The thumb rule is, on both sides, the same data type should exist.

**c) Boolean casting**

A boolean value cannot be assigned to any other data type. Except boolean, all the remaining 7 data types can be assigned to one another either implicitly or explicitly; but boolean cannot. We say, boolean is incompatible for conversion. Maximum we can assign a boolean value to another boolean.

Following raises

```
error. boolean x
= true;
int y = x;                //
error boolean x = true;
int y = (int) x;          // error
```

**byte –> short –> int –> long –> float –> double**

In the above statement, left to right can be assigned implicitly and right to left requires explicit casting. That is, byte can be assigned to short implicitly but short to byte requires explicit casting.

Following char operations are possible

```
class Demo {
  public static void main(String args[])
    { char ch1 = 'A';
    double d1 = ch1;

    System.out.println(d              // prints
    System.out.println(ch1 * ch1);   // prints 4225 , 65 *

    double d2 = 66.0;
    char ch2 = (char)
    d2;                               // prints
  }
}
```

## Procedure oriented approach:

The Languages like C, Fortran, Pascal, etc., are called procedure oriented programming languages since in these languages, a programmer uses procedures or functions to perform a task. When the programmer wants to write a program, he will first divide the task into several subtasks, each of which expressed as a function. So, C program generally contains several functions which are called and controlled from main() function. This approach is called *procedure oriented approach*.

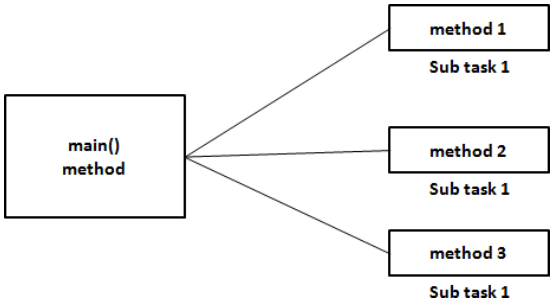



**Fig:** Procedure oriented approach

## Object oriented approach:

On the other hand, languages like C++ and Java use classes and objects in their programs and are called Object Oriented Programming languages. A class is a module which itself contains data and methods (functions) to achieve the task. The main task is divided into several modules, and these are represented as, classes. Each class can perform some tasks for which several methods are written in a class. This approach is called *Object oriented approach.*

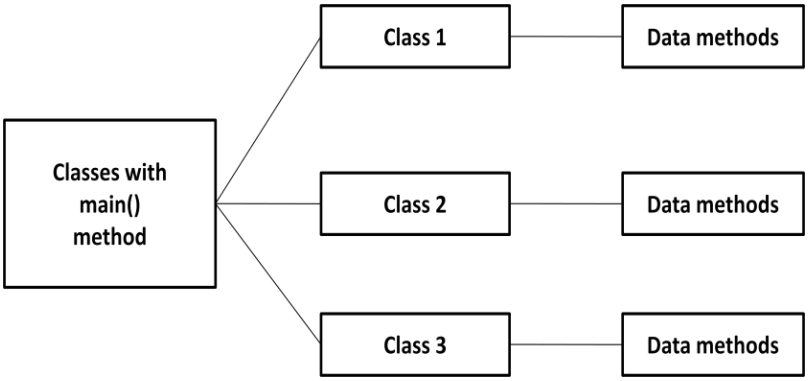



**Fig:** Object oriented approach

## Problems in Procedural oriented approach:

- There is NO Reusability.
- The lines of code are very high.
- Security is very less.
- Don't have any access specifier.

**Difference between POP and OOP:**

| Procedure Oriented Programming | Object Oriented Programming |
|---|---|
| In POP, program is divided into small parts called functions. | In OOP, program is divided into parts called objects. |
| In POP, Importance is not given to data but to functions as well as sequence of actions to be done. | In OOP, Importance is given to the data rather than procedures or functions because it works as a real world. |
| POP follows Top Down approach. | OOP follows Bottom Up approach. |
| POP does not have any access specifier. | OOP has access specifiers named Public, Private, Protected, etc. |
| In POP, Data can move freely from function to function in the system. | In OOP, objects can move and communicate with each other through member functions. |
| To add new data and function in POP is not so easy. | OOP provides an easy way to add new data and function. |
| In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system. | In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data. |
| POP does not have any proper way for hiding data so it is less secure. | OOP provides Data Hiding so provides more security. |
| In POP, Overloading is not possible. | In OOP, overloading is possible in the form of Function Overloading and Operator Overloading. |
| **Examples of POP are:** C, VB, FORTRAN, Pascal. | **Examples of OOP are:** C++, JAVA, VB.NET, C#.NET. |

## Principles of OOP:

The key ideas of the object oriented approach are:

- Class / Objects
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

### *Class / Objects*

Entire OOP methodology has been derived from a single root concept, called 'object'. An object is anything that really exists in the world and can be distinguished from others. This definition specifies that everything in this world is an object. For example, a table, a ball, a car, a dog, a person, etc., everything will come under

objects.

Every object has properties and can perform certain actions. For example, let us take a person whose name is 'Raju'. Raju is an object because he exists physically. He has properties

like name, age, gender, etc. These properties can be represented by variables in our programming.

```
String
name;    int
age;
char gender;
```

Similarly, Raju can perform some actions like talking, walking, eating and sleeping. We may not write code for such actions in programming. But, we can consider calculations and processing of data as actions. These actions are performed by methods (functions), So an object contains variables and methods.

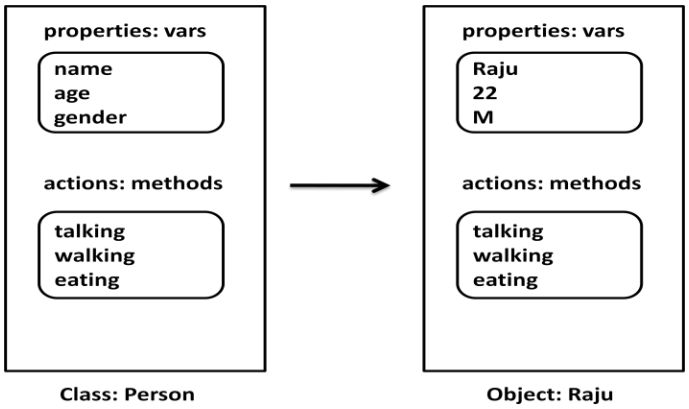



**Fig:** Person class and Raju Object

From the above diagram we can understand that Person is class and Raju is an object i.e., instance of class.

***Note**: A class is a model for creating objects and does not exist physically. An object is anything that exists physically. Both the class and objects contain variables and methods.*

## *Abstraction*

There may be a lot of data, a class contains and the user does not need the entire data. The user requires only some part of the available data. In this case, we can hide the unnecessary data from the user and expose only that data that is of interest to the user. This is called abstraction.

A bank clerk should see the customer details like account number, name and balance amount in the account. He should not be entitled to see the sensitive data like the staff salaries, profit or loss of the bank amount paid by the bank and loans amount to be recovered etc,. So such data can abstract from the clerk's view. Whereas the bank manager is interested to know this data, it will be provided to the manager.

## *Encapsulation*

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java –

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

## *Inheritance*

It creates new classes from existing classes, so that the new classes will acquire all the features of the existing classes is called Inheritance. A good example for Inheritance in nature is parents producing the children and children inheriting the qualities of the parents.

There are three advantages inheritance. First, we can create more useful classes needed by the application (software). Next, the process of creating the new classes is very easy, since they are built upon already existing classes. The last, but very important advantage is managing the code becomes easy, since the programmer creates several classes in a hierarchical manner, and segregates the code into several modules.

## *Polymorphism*

The word polymorphism came from two Greek words 'poly' meaning 'many' and 'morphos' meaning 'forms'. Thus, polymorphism represents the· ability to assume several different forms. In programming, we can use a single variable to refer to objects of different types and thus using that variable we can call the methods of the different objects. Thus method call can perform different tasks depending on the type of object.

Polymorphism provides flexibility in writing programs in such a way that the programmer uses same method call to perform different operations depending on the requirement.

### Application of OOP:

The promising areas includes the followings,

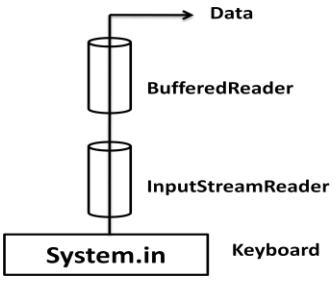
1. *Real Time Systems* Design
2. Simulation and Modeling System
3. Object Oriented Database
4. Object Oriented Distributed Database
5. Client-Server System
6. Hypertext, Hypermedia
7. Neural Networking and Parallel Programming
8. Decision Support and Office Automation Systems
9. CIM/CAD/CAM Systems
10. AI and Expert Systems

## Accessing Input from the keyboard:

- Input represents data given to a program and output represents data displayed as a result of a program.
- A stream is required to accept input from the keyboard. A stream represents flow of data from one place to another place.
- It is like a water-pipe where water flows. Like a water-pipe carries water from one place to another, a stream carries data from one place to another place.
- A stream can carry data from keyboard to memory or from memory to printer or from memory to a file.
- A stream is always required if we want to *move* data from one place to another.
- Basically, there are two types of streams: input streams and output streams.

- Input streams are those streams which receive or read data coming from some other place.
- Output streams are those streams which send or write data to some other place.
- All streams are represented by classes in java.io (input and output) package.

- This package contains a lot of classes, all of which can be classified into two basic categories: input streams and output streams.
- Keyboard is represented by a field, called in System class. When we write System.in, we are representing a standard input device, i.e. keyboard, by default. System class is found in java.lang (language) package and has three fields' as shown below.
- All these fields represent some type of stream:
  - *System.in*: This represents Ii1putStream object, which by default represents standard input device, i.e., keyboard.
  - *System.out*: This represents PrintStream object, which by default represents standard output device, i.e., monitor.
  - *System.err*: This field also represents PrintStream object, which by default represents monitor.
- Note that both *System.out* and *System.err* can be used to represent the monitor and hence any of these two can be used to send data to the monitor.
- To accept data from the keyboard i.e., *System.in*. we need to connect it to an input stream. And input stream is needed to read data from the keyboard.




- Connect the keyboard to an input stream object. Here, we can use

  InputStreamReader that can read data from the keyboard.

  **InputStreamReader isr = new InputStreamReader(System.in);**

- Connect InputStreamReader to BufferedReader, which is another input type of stream.

  **BufferedReader br = new BufferedReader(isr);**

- Now we can read data coming from the keyboard using read( ) and readLine( ) methods available in BufferedReader class.

**Ex:** Write a JAVA Program to read Single character from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter a
                single Character: ");
                char ch = (char)br.read(); System.out.print("\n The
                Character is "+ch);
        }
}
```

**Output:**

```
javac Test.java
java Test
Enter a single Character:
m The Character is m
```

**Ex:** Write a JAVA Program to read String from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter the
                String: ");
                String str = br.readLine(); System.out.print("\n The
                String is "+str);
        }
}
```

**Output:**

```
javac Test.java
java Test
Enter the String:
mothi The String is
mothi
```

**Ex:** Write a JAVA Program to read Integer from the keyboard.

```
import java.io.*; import
java.lang.*; class Test {
        public static void main(String[] args)throws IOException
        {
                InputStreamReader isr = new InputStreamReader(System.in);
                BufferedReader br = new BufferedReader(isr); System.out.print("Enter the
                Number: ");
                int n = Integer.parseInt(br.readLine());
                System.out.print("\n The number is "+n);
        }
}
```

**Output:**

```
javac Test.java
java Test
Enter the Number:
56 The Number is 56
```

- To accept Float value we have to use
  **float f = Float.parseFloat(br.readLine());**
- To accept Double value we have to use
  **double d = Double.parseDouble(br.readLine());**

## Reading Input with *java.util.Scanner* Class:

We can use Scanner class of java.util package to read input from the keyboard or a text file. When the Scanner class receives input, it breaks the input into several pieces, called. *tokens.* These tokens can be retrieved from the Scanner object using the following methods

- **next() - to read a string**
- **nextByte() - to read byte value**
- **nextInt() - to read an integer value**
- **nextFloat() - to read float value**

- nextLong()- to read long value.
- nextDouble () -to read double value

To read input from keyboard, we can use Scanner class as:

```
Scanner sc = new Scanner(System.in);
```

**Ex:** Write a JAVA Program to read different types of data separated by space, from the keyboard using the Scanner class.

```
import java.util.Scanner; import
java.lang.*; class Test {
        public static void main(String[] args)
        {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter id name sal: "); int id =
                sc.nextInt();
                String name = sc.next(); float sal =
                sc.nextFloat();
                System.out.println("Id is: "+id);
                System.out.println("Name is: "+name);
                System.out.println("Sal is: "+sal);

        }
}
```

**Output:**

```
javac Test.java
java Test
Enter id name sal: 09 sudheer
40000.00 Id is: 09
Name is:
sudheer Sal is:
40000.00
```

# Strings:

In java, string is basically an object that represents sequence of char values. An array of characters works same as java string.

String is a class in java.lang package. But in java, all classes are also considered as data types. So, we can take string as a data type also. A class is also called as user-defined data type.

### Creating Strings:

There are three ways to create strings in Java:

- We can create a string just by assigning a group of characters to a string type variable:

```
String s;
s = "Hello";
```

- Preceding two statements can be combined and written as:

```
String s = "Hello";
```

In this, case JVM creates an object and stores the string: "Hello" in that object. This object is referenced by the variable's'. Remember, creating object means allotting memory for storing data.

- We can create an object to String class by allocating memory using new operator. This is just like creating an object to any class, like given here:

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

  Here, we are doing two things. First, we are creating object using new operator. Then, we are storing the string: "Hello" into the object.

- The third way of creating the strings is by converting the character arrays into strings. Let us take a character type array: arr[] With some characters, as:

```
char arr[]={'H','e','l','l','o'};
```

- Now create a string object, by pass.ing the array name to it, as:

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

## String Methods:

| No. | Method | Description |
|---|---|---|
| 1 | char **charAt**(int index) | returns char value for the particular index |
| 2 | **int length**() | returns string length |
| 3 | static String **format**(String format, Object... args) | returns formatted string |
| 4 | String **substring**(int beginIndex, int endIndex) | returns substring for given begin index and end index |
| 5 | boolean **contains**(CharSequence s) | returns true or false after matching the sequence of char value |
| 6 | static String **join**(CharSequence delimiter, CharSequence... elements) | returns a joined string |
| 7 | boolean **equals**(Object another) | checks the equality of string with object |
| 8 | boolean **isEmpty**() | checks if string is empty |
| 9 | String **concat**(String str) | concatenates specified string |
| 10 | String **replace**(char old, char new) | replaces all occurrences of specified char value |

| | | |
|---|---|---|
| 11 | static String equalsIgnoreCase(String another) | Compares another string. It doesn't check case. |

| | | |
|---|---|---|
| 12 | String[] split(String regex, int limit) | returns splitted string matching regex and limit |
| 13 | int indexOf(int ch) | returns specified char value index |
| 14 | String toLowerCase() | Returns string in lowercase. |
| 15 | String toUpperCase() | Returns string in uppercase. |
| 16 | String trim() | Removes beginning and ending spaces of this string. |
| 17 | static String valueOf(int value) | Converts given type into string. It is overloaded. |

**equals method in java:**

**Program:**

```
class StringComapre
{
        public static void main(String args[])
        {
                String s1 = "Hello";
                String s2 = new String("Hello");
                System.out.println(s1==s2); //false
                System.out.println(s1.equals(s2));
                //true
        }
}
```

**Immutability of String:**

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
 public static void main(String
  args[]){ String s="Sachin";
  s.concat(" Tendulkar");
  System.out.println(s);
 }
}
```

**Output:** Sachin

- But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
class Testimmutablestring1{
 public static void main(String
  args[]){ String s="Sachin";
  s=s.concat("
  Tendulkar");
  System.out.println(s);
 }
}
```

**Output:** Sachin Tendulkar

## StringBuffer Class:

- Java *StringBuffer* class is used to create mutable (modifiable) string. The *StringBuffer* class in java is same as String class except it is mutable i.e. it can be changed.
- *String* class objects are immutable and hence their contents cannot be modified. *StringBuffer* class objects are mutable; so they can be modified. Moreover the methods that directly manipulate data of the object are not available in String class. Such methods are available in *StringBuffer* class.

## Creating StringBuffer objects:

We can create *StringBuffer* object by using new operator and pass the string to the object, as:

```
StringBuffer sb = new StringBuffer("Hello");
```

## Methods in StringBuffer class:

| S.No. | Method & Description |
|---|---|
| 1 | **StringBuffer append(String str)**<br>This method appends the specified string to this character sequence. |
| 2 | **char charAt(int index)**<br>This method returns the char value in this sequence at the specified index. |
| 3 | **StringBuffer delete(int start, int end)**<br>This method removes the characters in a substring of this sequence. |
| 4 | **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**<br>This method characters are copied from this sequence into the destination character array dst. |
| 5 | **int indexOf(String str, int fromIndex)**<br>This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index. |
| 6 | **StringBuffer insert(int offset, String str)**<br>This method inserts the string into this character sequence. |
| 7 | **int lastIndexOf(String str)**<br>This method returns the index within this string of the rightmost occurrence of the specified substring. |
| 8 | **int length()**<br>This method returns the length (character count). |
| 9 | **StringBuffer replace(int start, int end, String str)**<br>This method replaces the characters in a substring of this sequence with characters in the specified String. |
| 10 | **StringBuffer reverse()**<br>This method causes this character sequence to be replaced by the reverse of the sequence. |
| 11 | **void setCharAt(int index, char ch)**<br>The character at the specified index is set to ch. |
| 12 | **String substring(int start, int end)**<br>This method returns a new String that contains a subsequence of characters currently contained in this sequence. |
| 13 | **String toString()**<br>This method returns a string representing the data in this sequence. |
| 14 | **void trimToSize()**<br>This method attempts to reduce storage used for the character sequence. |

**Program:** Write a JAVA Program to delete single character. class DeleteCharacter

```
{
        public static void main(String args[])
        {
                StringBuffer sb = new StringBuffer("Hello");
                sb.delete(1,2);
                System.out.println(sb); //Hllo
                sb.delete(3,4);
                System.out.println(sb); //Hll
        }
}
```

## Classes and Objects:

We know a class is a model for creating objects. This means, the properties and actions of the objects are written in the class. Properties are represented by variables and actions of the objects are represented by methods. So, a class contains variables and methods. The same variables and methods are also available in the objects because they are created from the class. These variables are also called 'instance variables' because they are created inside the object (instance).

If we take Person class, we can write code in the class that specifies the properties and actions performed by any person. For example, a person has properties like name, age, etc. Similarly a person can perform actions like talking; walking, etc. So, the class Person contains these properties and actions as shown

```
class Person
{
        String name;
        int age; void
        talk()
        {
                System.out.println("Hello my name is "+name);
                System.out.println("and my age is "+age);
        }
}
```

here:

**Object Creation:**

We know that the class code along with method code is stored in 'method area' of the JVM. When an object is created, the memory is allocated on 'heap'. After creation of an object, JVM produces a unique reference number for the object from the memory address of the object. This reference number is also called hash code number.

To know the hashcode number (or reference) of an object, we can use hashCode() method object class, as shown here:

```
Person p = new Person();
System.out.println(p.hashCode());
```

The object reference, (hash code) internally represents heap memory where instance- variables are stored. There would be a pointer (memory address) from heap memory to a special structure located in method area. In method area, a table

is available which contains pointers to static variables and methods.

```
class Person
{
	String name;
	int age; void
	talk()
	{
		System.out.println("Hello my name is "+name);
		System.out.println("and my age is "+age);
	}
}
class Demo
{
	public static void main(String args[])
	{
		Person p = new Person();
		System.out.println(p.hashCode());	//
		705927765
	}
```

Initializing the instance variables:

It is the duty of the programmer to initialize the instance depending on his requirements. There are various ways to do this.

First way is to initialize the instance variables of Person class in other class, i.e., Demo class. For this purpose, the Demo class can be rewritten like this:

```
class Demo
{
	public static void main(String args[])
	{
		Person p = new Person();
		p.name="Raju";  p.age=22;
		System.out.println(p.hashCode());	// 705927765
	}
}
```

Second way is to initialize the instance variables of Person class in the Same class. For this purpose, the Person class can be rewritten like this:

```
class Person
{
	String name="Raju";
	int age=22;
	void talk()
	{
		System.out.println("My Name is "+name);
		System.out.println("My Age is "+age);
	}
}
```

```
class Demo
{
	public static void main(String args[])
	{
		Person p1 = new Person();
		System.out.println("p1 hashcode "+p1.hashCode());
		p1.talk();
		Person p2 = new Person();
		System.out.println("p2 hashcode "+p2.hashCode());
		p2.talk();
	}
}
```

Output:

```
p1 hashcode
705927765 My Name
is Raju
My Age is 22
p2 hashcode
705934457 My Name
is Raju
My Age is 22
```

But, the problem in this way of initialization is that all the objects are initializing with same data.

## Constructors:

The third possibility of initialization is using constructors. A constructor is similar to a method is used to initialize the instance variables. The sole purpose of a constructor is to initialize the instance variables. A constructor has the following characteristics:

- The constructor's name and class name should be same. And the constructor's name should end with a pair of simple braces.

```
Person()
{
}
```

- A constructor may have or may not have parameters: Parameters are variables to receive data from outside into the constructor. If a constructor does not have any parameters, it is called 'Default constructor'. If a constructor has 1 or more parameters, it is called 'parameterized constructor'; For example, we can write a default constructor as:

```
Person()
{
}
```

And a Parameterized constructor with two parameters, as:

```
Person(String s, int i)
{
}
```

- A constructor does not return any value, not even 'void'. Recollect, if a method does not return any value, we write 'void' before the method name. That means, the method is returning ‘void’ which means 'nothing'. But in case of a constructor, we should not even write 'void' before the constructor.
- A constructor is automatically called and executed at the time of creating an object. While creating an object, if nothing is passed to, the object, the default constructor is called and executed. If-some values are passed to the object, then the parameterized constructor is called. For example, if we create the

```
Person p = new Person(); // Default Constructor
Person p = new Person(Raju, 22); // Parameterized Constructor
```

object as:

- A constructor is called and executed only once per object. This means .when we create an object, the constructor is called. When we create second object, again the constructor is called second time.

```
class Person
{
 String name;
 int age;
 Person()
 {
       name="Raju"; age=22;
 }
 void talk()
 {
       System.out.println("Hello my name is "+name);
       System.out.println("and my age is "+age);
 }
}
class Demo
{
 public static void main(String args[])
 {
       Person p = new Person(); // Default Constructor
       System.out.println("p  hashcode  "+p.hashCode());
       p.talk();
       Person p1 = new Person();
       System.out.println(p1.hashCode());
       p1.talk();
 }
}
```

Output:

```
p hashcode
705927765 My
Name is Raju
My Age is 22
p1 hashcode
705934457 My Name
```

is Raju
My Age is 22

From the output, we can understand that the same data "Raju" and 22 are store in both objects p1 and p2. p2 object should get p2's data, not p1's data. Isn't it? To mitigate the problem, let us try parameterized constructor, which accepts data from outside and initializes instance variables with that data.

```
class Person
{
	String name;
	int age;
	Person()
	{
		name="Raju";
		age=22;
	}
	Person(String s, int i)
	{
		name=s;
		age=i;
	}
	void talk()
	{
		System.out.println("My Name is "+name);
		System.out.println("My Age is "+age);
	}
}
class Demo
{
	public static void main(String args[])
	{
		Person p1 = new Person();
		System.out.println("p1 hashcode "+p1.hashCode()); p1.talk();
		Person p2 = new Person("Sita",23);
		System.out.println(p2.hashCode());
		p2.talk();
	}
}
```

Output:

```
p1 hashcode
705927765 My Name
is Raju
My Age is 22
p2 hashcode
705934457 My Name
is Sita
My Age is 23
```

*Q: When is a constructor called, before or after creating the object?*

*Ans: A constructor is called concurrently when the object creation is going on. JVM first allocates memory for the object and then executes the constructor to initialize the instance variables. By the time, object creation is completed; the constructor execution is also completed.*

**Difference between Default Constructor and Parameterized Constructor:**

| Default Constructor | Parameterized Constructor |
|---|---|
| Default constructor is useful to initialize all the objects with same data. | Parameterized constructor is useful to initialize each object with different data. |
| Default constructor does not have any parameters. | Parameterized constructor will have 1 or more parameters. |
| When the data is not passed at the time of creating object, default constructor is called. | When the data is passed at the time of creating object, default constructor is called. |

**Difference between Constructor and Method:**

| Java Constructor | Java Method |
|---|---|
| Constructor is used to initialize the state of an object. | Method is used to expose behaviour of an object. |
| Constructor must not have return type. | Method must have return type. |
| Constructor is invoked implicitly. | Method is invoked explicitly. |
| The java compiler provides a default constructor if you don't have any constructor. | Method is not provided by compiler in any case. |
| Constructor name must be same as the class name. | Method name may or may not be same as class name. |

***Q:*** *What is constructor overloading?*

***Ans:*** *Writing two or more constructors with the same name but with difference in the parameters is called constructor overloading. Such constructors are useful to perform different tasks.*

**Write a Java Program for Constructor Overloading?**

```
class Box
{
	double width, height, depth; Box()
	{
		width = height = depth = 0;
	}
	Box(double len)
	{
		width = height = depth = len;
	}
	Box(double w, double h, double d)
	{
		width = w;
		height = h;
		depth = d;
	}
```

```
        double volume()
        {
                return width * height * depth;
        }
}
public class Test
{
        public static void main(String args[])
        {
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mybox3 = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println(" Volume of mybox1 is " + vol); vol =
                mybox2.volume();
                System.out.println(" Volume of mybox2 is " + vol); vol =
                mybox3.volume();
                System.out.println(" Volume of mybox3 is " + vol);
        }
}
```

Output:

Volume of mybox1 is
3000.0 Volume of mybox2
is 0.0 Volume of mybox3 is
343.0

## Java Garbage Collection

- In java, garbage means unreferenced objects. Garbage Collection is process of reclaiming the runtime unused memory automatically.
- In other words, it is a way to destroy the unused objects. To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

## 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 (a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

- Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable (thus eligible for GC) if it is no longer required.
- There are many ways:
  - By nulling the reference
  - By assigning a reference to another

- By annonymous object etc.

1) By nulling a reference:

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

2) By assigning a reference to another:

```
Employee e1=new Employee();
Employee e2=new Employee();
e1=e2;//now the first object referred by e1 is available for garbage collec
tion
```

3) By annonymous object:

```
new Employee();
```

## 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 Object class as: `protected void finalize(){}`

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public static void gc(){}
```

**Program:**

```
public class TestGarbage1
{
   public void finalize()
   {
     System.out.println("object is garbage collected");
   }
    public static void main(String args[])
   {
      TestGarbage1 s1=new TestGarbage1();
      TestGarbage1 s2=new TestGarbage1();
      s1=null;
      s2=null;
      System.gc();
   }
}
```

**Output:**

```
object is garbage
collected object is
garbage collected
```

# Methods in Java:

- A method represents a group of statements that performs a task. Here 'task' represents a calculation or processing of data or generating a report etc.
- A Method has two types:
  - Method Header
  - Method Body

- **Method Header:** It contains method name, method parameters and method return data type.

```
returntype methodName(parameters)
```

- **Method Body:** Method body consists of group of statements which contains logic to perform the task.

```
{
    Statements;
}
```

- If a method returns some value, then a return statement should be written within the body of method, as:

```
return somevalue;
```

- There are two types of methods in java:
  a) Instance methods
  b) Static methods

**a) Instance Methods:**

Instance methods are the methods which ***act*** on the instance variables of the class. To call the instance methods, we should use the form:

```
objectname. methodname ();
```

**Example-1:** Write a program to perform addition of two numbers using instance method without return statement.

```
class Addition
{
    void add(double a,double b)
    {
        double c=a+b;
        System.out.println("Addition is
        "+c);
    }
}
class MethodDemo
{
    public static void main(String args[])
    {
        Addition a1=new
        Addition(); a1.add(27,35);
    }
```

**Output:**

```
Addition is 62
```

**Example-2:** Write a program to perform addition of two numbers using instance method with return statement.

```
class Addition
{
    double add(double a,double b)
    {
        double c=a+b;
        return c;
    }
```

```
}
class MethodDemo
{
      public static void main(String args[])
      {
            Addition a1=new
            Addition(); double
            c=a1.add(27,35);
           System.out.println("Addition is "+c);

      }
```

**Output:**

```
Addition is 62
```

b) **Static Methods:**

Static methods are the methods which ***do not act*** on the instance variables of the class. static methods are declared as 'static'. To call the static methods, we need not create the object. we call a static method, as:

```
ClassName. methodname ();
```

**Example-1:** Write a program to perform addition of two numbers using static method without return statement.

```
class Addition
{
      static void add(double a,double b)
      {
            double c=a+b;
            System.out.println("Addition is
            "+c);
      }
}
class MethodDemo
{
      public static void main(String args[])
      {
            Addition.add(27,35);
      }
```

**Output:**

```
Addition is 62
```

**Example-2:** Write a program to perform addition of two numbers using static method with return statement.

```
class Addition
{
      static double add(double a,double b)
      {
            double c=a+b;
            return c;
      }
}
class MethodDemo
{
```

```
        public static void main(String args[])
        {
             double c= Addition.add(27,35);
            System.out.println("Addition is
            "+c);

        }
```

**Output:**

```
Addition is 62
```

*Q: why instance variables are not available to static methods?*
***Ans:*** *After executing static methods, JVM creates the objects. So, the instance variables are not available to static methods.*

**Example:** Write a JAVA program to implement method overloading.

## Static Block:

- A static block is a block of statements declared as 'static'.

```
static {
                statements;
}
```

- JVM executes a static block on highest priority basis. This means JVM first goes to static block even before it looks for the `main()` method in the program.

**Example-1:** Write a program to test which one is-executed first by JVM, the static block or the static method.

```
class Demo
{
   static
   {
      System.out.println("Static block");
   }
   public static void main(String args[])
   {
      System.out.println("Static Method");
   }
}
```

**Output:**

```
Static block
Static
```

**Example-2:** Write a java program without `main()` method.

```
class Demo
{
   static
   {
      System.out.println("Static block");
   }
}
```

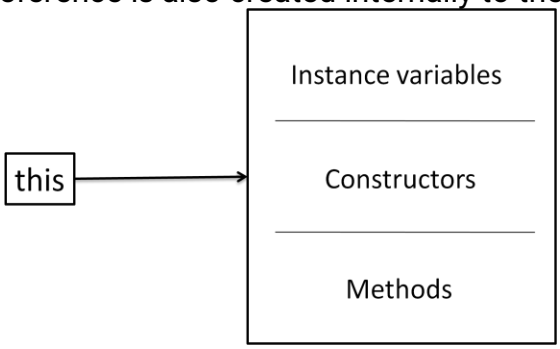
**Output:**

```
Static block
```

*Q: Is it possible to compile and run a Java program without writing* `main()` *method? Ans: Yes, it is possible by using a static block in theJava program.*

## 'this' keyword:

- 'this' is a keyword that refers to the current object of the class where it is used. In other words, 'this' refers the object of the present class.
- Generally, we write instance variables, constructors and methods in a class. All these members are referenced by 'this'. When an object is created to a class, a default reference is also created internally to the object.




Instance variables are not initialized then default value

int -0
float -0.0
Char- emptyspace
String-null
Boolean-false

**Example:** Write a program for 'this' keyword.

```
class Demo
{
      int x;
      Demo(int
      x)
      {
          this.x=x+3;
       System.out.println("x="+x);
      System.out.println("this.x="+this.x);
      }
      public static void main(String args[])
      {
            Demo d=new Demo(5);
      }
```

**Output:**

```
x=5
this.x=8
```

## Recursion:

A method calling itself is known as recursive method', and that process is called 'recusrion'. It is possible to write recursive methods in Java. Let us, take an example to find the factorial value of the given number. Factorial value of number num is defined as: num * (num-1) * (num-2) * .....

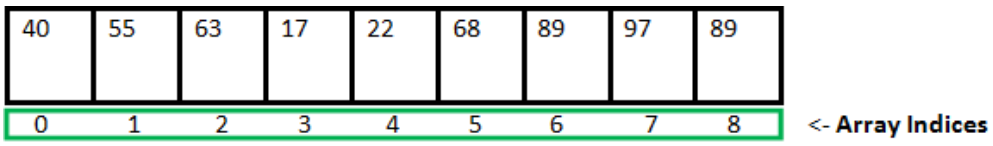
**Example:** Write a java Program to find factorial of a given number using recursion.

```
import java.util.Scanner;
class Factorial
{
     static long fact(long num)
     {
          if(num==1 || num==0)
               return 1;
          else
```

```
        {
            return num*fact(num-1);
        }
    }
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.print("Enter number: ");
        long num=sc.nextInt();
        long f=Factorial.fact(num);
        System.out.println("Factorial is
        "+f);
    }
```

## Arrays:

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.




**Array Length = 9**
**First Index = 0**
**Last Index = 8**

### Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

### Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

### Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

### Declaring Single Dimensional Array:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

**Syntax:**

```
dataType[] varName;
      or
dataType varName[];
```

**Example:**

```
int a[];        or        int[] a;
```

### Instantiating an Array in Java

When an array us declared, only a reference of array is created. To actually create or give memory to array, you create an array like this: The general form of *new* as it applies to one-dimensional arrays appears as follows:

```
varName = new type [size];
```

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *varName* is the name of array variable that is linked to the array.

That is, to use *new* to allocate an array, you must specify the type and number of elements to allocate.

### Single -Dimensional Array:

The declaration and instantiating of an 1-D array is, as follows

**Example:**

```
int[] a=new int[5];
```

**Example:** Write a JAVA program to find sum of all numbers in an array.

```
import java.util.Scanner;
class SumArray
{
	public static void main(String[] args)
	{
		Scanner sc=new Scanner(System.in);
		System.out.print("Enter the no. of elements: ");
		int n=sc.nextInt();
		int[] a=new int[n];
		System.out.print("Enter the "+n+" elements ");
		for(int i=0;i<n;i++)
			a[i]=sc.nextInt();
		int sum=0;
		for(int i=0;i<n;i++)
		{
			sum=sum+a[i];
		}
		System.out.print("The Sum of elements is "+sum);
	}
}
```

**Output:**

```
Enter the no. of elements:
5 Enter the 5 elements 5 4
2 1 3 The Sum of elements
is 15
```

### Two-Dimensional Array:

The declaration and instantiating of an 1-D array is, as follows

**Example:**

```
int[][] a=new int[2][3];
```

**Example:** Write a java Program to perform Matrix addition.

```
import java.util.Scanner;
class MatrixAddition
{
	public static void main(String[] args)
	{
		Scanner sc=new Scanner(System.in);
		int[][] a=new int[2][2];
		int[][] b=new int[2][2];
		int[][] c=new int[2][2];
		System.out.println("Enter the A Matrix elements ");
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				a[i][j]=sc.nextInt();
		System.out.println("Enter the B Matrix elements ");
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				b[i][j]=sc.nextInt();
		for(int i=0;i<2;i++)
			for(int j=0;j<2;j++)
				c[i][j]=a[i][j]+b[i][j];
		System.out.println("The Addition of Matrix is ");
		for(int i=0;i<2;i++)
		{
			for(int j=0;j<2;j++)
			{
				System.out.print(c[i][j]+" ");
			}
			System.out.print("\n");
		}
	}
}
```

**Output:**

```
Enter the A Matrix
elements 1 2
3 4
Enter the B Matrix
elements 5 6
7 8
The Addition of Matrix
is 6 8
10 12
```

**Example:** Write a java Program to perform Matrix multiplication.

```
import java.util.Scanner;
class
MatrixMultiplication
{
 public static void main(String[] args)
 {
      Scanner sc=new Scanner(System.in);
      int[][] a=new int[2][2];
      int[][] b=new int[2][2];
      int[][] c=new int[2][2];
      System.out.println("Enter the A Matrix elements ");
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                a[i][j]=sc.nextInt();
      System.out.println("Enter the B Matrix elements ");
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                b[i][j]=sc.nextInt();
      for(int i=0;i<2;i++)
           for(int j=0;j<2;j++)
                for(int k=0;k<2;k++)
                c[i][j]+=a[i][k]*b[k][j];
      System.out.println("The Multiplication of Matrix is ");
      for(int i=0;i<2;i++)
      {
           for(int j=0;j<2;j++)
           {
                System.out.print(c[i][j]+" ");
           }
           System.out.print("\n");
      }
 }
```

**Output:**

```
Enter the A Matrix
elements 1 2
3 4
Enter the B Matrix
elements 5 6
7 8
The Multiplication of Matrix
is 19 22
43 50
```

## Command line arguments:

- The java command-line argument is an argument i.e. passed at the time of running the java program.
- The arguments passed from the console can be received in the java program and it can be used as an input.
- So, it provides a convenient way to check the behaviour of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

**Example:** Write a JAVA Program to read input from command line.

```
class CommandLine
{
  public static void main(String[] args)
  {
       System.out.println("Command line argument values are ");
       for(int i=0;i<args.length;i++)
       {
            System.out.println(args[i]);
       }
  }
}
```

**Output:**

```
Command line argument values
are 27
2.5
krishna
```

## Nested Classes:

- Nested classes are the classes that contain other classes like inner classes. Inner class is basically a safety mechanism since it is hidden from other classes in its outer class.
- To make instance variables not available outside the class, we use 'private' access specifier before the variables. This is how we provide the security mechanism to variables. Similarly, in some cases we want to provide security for the entire class.
- In this case, can we use 'private' access specifier before the class? The problem is, if we use private access specifier before a class, the class is not available to the Java compiler or JVM. SO it is illegal to use 'private' before a class name in Java.
- But, private specifier is allowed before an inner class and thus it is useful to provide security for the entire inner class.
- An inner class is a class that is defined inside another class.
- Inner class is a safety mechanism.
- Inner class is hidden from other classes in its outer class.
- An object to inner class cannot be created in other classes.
- An object to inner class can be created only in its outer class.

- Inner class _can access the members of outer class directly.
- Inner class object and outer class objects are created in separate memory locations.

**Example:** Write a JAVA program to create the outer class BankAccount and the inner class Interest in it.

```
import java.util.Scanner;
class BankAccount
{
	double bal;
	BankAccount(double
	b)
	{
		bal=b;
	}
	void contact(double r)
	{
		Scanner sc=new Scanner(System.in);
		System.out.print("Enter Password: ");
		String password=sc.next();
		if(password.equals("cse556"))
		{
			Interest in=new
			Interest(r);
			in.calculateInterest();
		}
	}
	private class Interest
	{
		double rate;
		Interest(double
		r)
		{
			rate=r;
		}
		void calculateInterest()
		{
			double
			inter=bal*rate/100;
			bal=bal+inter;
			System.out.println("Updated Balance= "+bal);
		}
	}
	public static void main(String[] args)
	{
```

**Output:**

Enter Password: cse556
Updated Balance=
10950.0