Need of OOP:

OOP stands for **Object-Oriented Programming**.

Procedural programming is about writing procedures or methods that perform operations on the data, while object-oriented programming is about creating objects that contain both data and methods.

Object-oriented programming has several advantages over procedural programming:

- OOP is faster and easier to execute
- OOP provides a clear structure for the programs
- OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
- OOP makes it possible to create full reusable applications with less code and shorter development time

Benefits of OOP

- We can build the programs from standard working modules that communicate with one another, rather than having to start writing the code from scratch which leads to saving of development time and higher productivity,
- OOP language allows breaking the program into the bit-sized problems that can be solved easily (one object at a time).
- The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.
- OOP systems can be easily upgraded from small to large systems.
- It is possible that multiple instances of objects co-exist without any interference,
- It is very easy to partition the work in a project based on objects.
- It is possible to map the objects in problem domain to those in the program.
- The principle of data hiding helps the programmer to build secure programs which cannot be invaded by the code in other parts of the program.
- By using inheritance, we can eliminate redundant code and extend the use of existing classes.
- Message passing techniques is used for communication between objects which makes the interface descriptions with external systems much simpler.
- The data-centered design approach enables us to capture more details of model in an implementable form.

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.

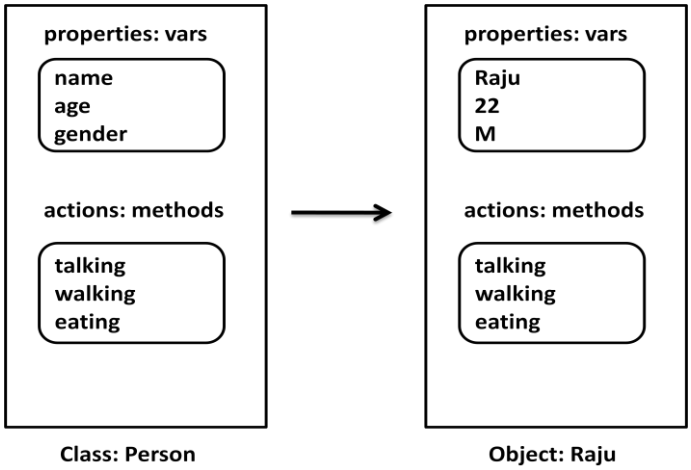


Fig: Person class and Raju Object

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

Procedural Languages vs OOP:

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriented programming, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach***.	Object-oriented programming follows a ***bottom-up approach***.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.

Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.
Procedural programming is based on the ***unreal world***.	Object-oriented programming is based on the ***real world***.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

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:

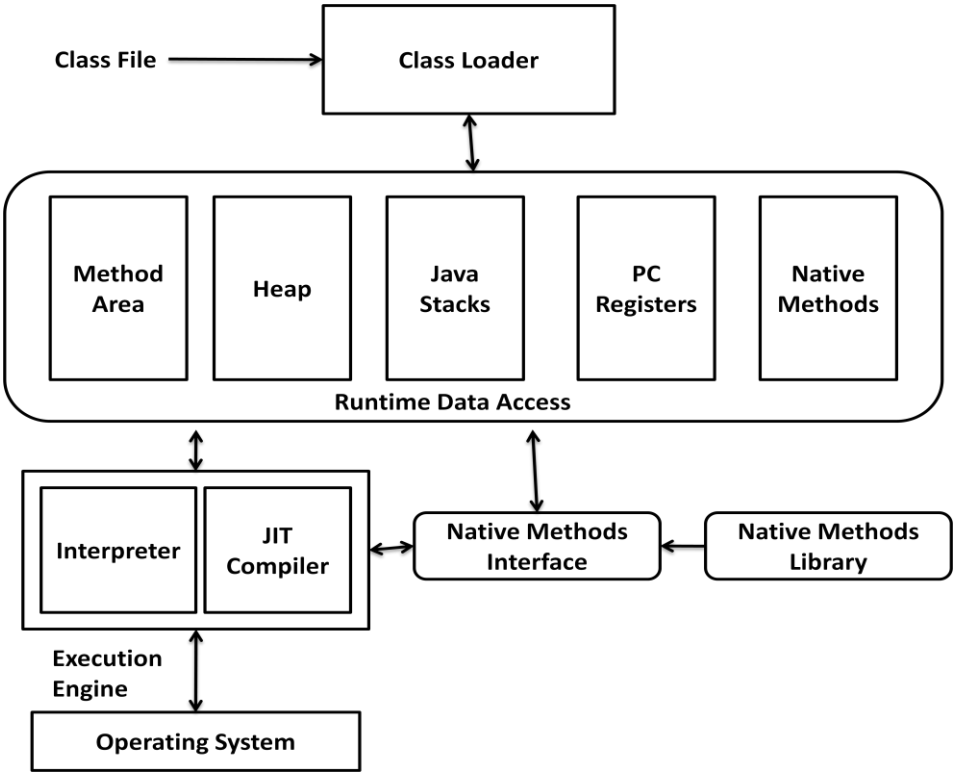


Fig: Components in JVM Architecture

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
    - 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 #include statement in C++. Example: `import java.lang.System;`
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
|
Methods

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 statement as:

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

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{
   statements;
}
```

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];
```

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, c;         // Declares three ints, a, b, and c.
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.
char a = 'a';        // the char variable a is initialized with value 'a'
```

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("Sudheer");
      empOne.printEmp();
   }
}
```

This will produce the following result –

Output

```
name  : Sudheer
```

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); //123.4
    }
}
```

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); //
      System.out.println(ch4); // §
    }
}
```

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 compliment)	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 : 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 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 Car;
      System.out.println( result );
   }
}
```

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);         // prints 10.0
```

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(d1);              // prints 65.0
    System.out.println(ch1 * ch1);       // prints 4225 , 65 * 65

    double d2 = 66.0;
    char ch2 = (char) d2;
    System.out.println(ch2);             // prints B
  }
}
```

Wrapper Classes:

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.

Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

- **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- **Synchronization:** Java synchronization works with objects in Multithreading.
- **java.util package:** The java.util package provides the utility classes to deal with objects.
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

The eight classes of the java.lang package are known as wrapper classes in Java. The list of eight wrapper classes is given below:

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
class WrapperExample1{
	public static void main(String args[]){
		//Converting int into Integer
		int a=20;
		Integer i=Integer.valueOf(a);
		Integer j=a;
		System.out.println(a+" "+i+" "+j);
	}
}
```

Output:

```
20 20 20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
class WrapperExample2{
	public static void main(String args[]){
		//Converting Integer to int
		Integer a=new Integer(3);
		int i=a.intValue();
		int j=a;
		System.out.println(a+" "+i+" "+j);
	}
}
```

Output:

```
3 3 3
```

Java Wrapper classes Example

```
//wrapper objects and vice-versa
class WrapperExample3{
	public static void main(String args[]){
		byte b=10;
		short s=20;
		int i=30;
		long l=40;
		float f=50.0F;
		double d=60.0D;
		char c='a';
		boolean b2=true;

		//Autoboxing: Converting primitives into objects
		Byte byteobj=b;
		Short shortobj=s;
		Integer intobj=i;
		Long longobj=l;
		Float floatobj=f;
		Double doubleobj=d;
		Character charobj=c;
		Boolean boolobj=b2;

		//Printing objects
		System.out.println("---Printing object values---");
		System.out.println("Byte object: "+byteobj);
		System.out.println("Short object: "+shortobj);
		System.out.println("Integer object: "+intobj);
		System.out.println("Long object: "+longobj);
		System.out.println("Float object: "+floatobj);
		System.out.println("Double object: "+doubleobj);
		System.out.println("Character object: "+charobj);
		System.out.println("Boolean object: "+boolobj);

		//Unboxing: Converting Objects to Primitives
		byte bytevalue=byteobj;
		short shortvalue=shortobj;
		int intvalue=intobj;
		long longvalue=longobj;
		float floatvalue=floatobj;
		double doublevalue=doubleobj;
		char charvalue=charobj;
		boolean boolvalue=boolobj;

		//Printing primitives
		System.out.println("---Printing primitive values---");
		System.out.println("byte value: "+bytevalue);
```

```
                System.out.println("short value: "+shortvalue);
                System.out.println("int value: "+intvalue);
                System.out.println("long value: "+longvalue);
                System.out.println("float value: "+floatvalue);
                System.out.println("double value: "+doublevalue);
                System.out.println("char value: "+charvalue);
                System.out.println("boolean value: "+boolvalue);
        }
    }
```

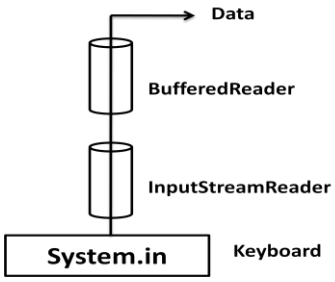
Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

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**
- ✓ **nextlnt() - 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
```

Java Control Statements:

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. **Decision Making statements**
   - if statements
   - switch statement
2. **Loop statements**
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. **Jump statements**
   - break statement
   - continue statement

1. Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax:

```
if(condition) {
      statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the if statement in the java code.

Example:

```
class Student {
      public static void main(String[] args) {
            int x = 10;
            int y = 12;
            if(x+y > 20) {
                  System.out.println("x + y is greater than 20");
            }
      }
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {
      statement 1; //executes when condition is true
}
else{
      statement 2; //executes when condition is false
}
```

Consider the following example.

Example:

```
class Student {
	public static void main(String[] args) {
		int x = 10;
		int y = 12;
		if(x+y < 10) {
			System.out.println("x + y is less than 10");
		}   else {
			System.out.println("x + y is greater than 20");
		}
	}
}
```

Output:

```
	x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax:

```
	if(condition 1) {
		statement 1; //executes when condition 1 is true
	}
	else if(condition 2) {
		statement 2; //executes when condition 2 is true
	}
	else {
		statement 2; //executes when all the conditions are false
	}
```

Consider the following example.

Example:

```
// Java program to illustrate if-else-if ladder
class ifelseifDemo {
	public static void main(String args[])
	{
		int i = 20;

		if (i == 10)
			System.out.println("i is 10");
		else if (i == 15)
			System.out.println("i is 15");
		else if (i == 20)
			System.out.println("i is 20");
		else
			System.out.println("i is not present");
	}
}
```

Output:

```
	i is 20
```

4. Nested if-statement

In nested if-statements, the if statement can contain a if or if-else statement inside another if or else-if statement.

Syntax:

```
if(condition 1) {
      statement 1; //executes when condition 1 is true
      if(condition 2) {
            statement 2; //executes when condition 2 is true
      }
      else{
            statement 2; //executes when condition 2 is false
      }
}
```

Consider the following example.

Example:

```
// Java program to illustrate nested-if statement

class NestedIfDemo {
      public static void main(String args[])
      {
            int i = 10;
            if (i == 10 || i<15) {
                  if (i < 15)
                        System.out.println("i is smaller than 15");
                  if (i < 12)
                        System.out.println("i is smaller than 12 too");
            } else{
                        System.out.println("i is greater than 15");
            }
      }
}
```

Output:

```
      i is smaller than 15
      i is smaller than 12 too
```

5. Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Syntax:

```
switch (expression){
    case value1:
      statement1;
      break;
    .
    .
    .
    case valueN:
      statementN;
```

```
      break;
    default:
      default statement;
}
```

Example:

```
class Student implements Cloneable {
      public static void main(String[] args) {
            int num = 2;
            switch (num){
                  case 0:
                        System.out.println("number is 0");
                        break;
                  case 1:
                        System.out.println("number is 1");
                        break;
                  default:
                        System.out.println(num);
            }
      }
}
```

Output:

```
      2
```

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

- for loop
- while loop
- do-while loop

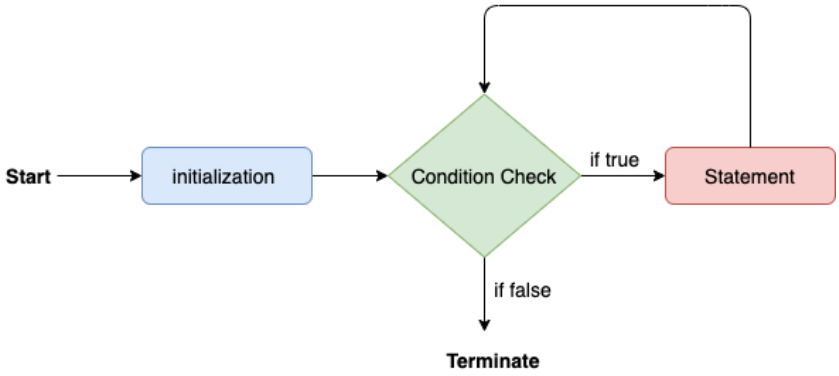
Let's understand the loop statements one by one.

Java for loop

In Java, for loop is similar to C and C++. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
      for(initialization, condition, increment/decrement) {
            //block of statements
      }
```

The flow chart for the for-loop is given below.



Example:

```
class Calculattion {
	public static void main(String[] args) {
	int sum = 0;
	for(int j = 1; j<=10; j++) {
		sum = sum + j;
	}
	System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
	The sum of first 10 natural numbers is 55
```

Java for-each loop

Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
	for(data_type var : array_name/collection_name){
		//statements
	}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Example:

```
class Calculation {
public static void main(String[] args) {
	String[] names = {"Java","C","C++","Python"};
	for(String name:names) {
		System.out.println(name);
	}
}
}
```

Output:

```
	Printing the content of the array names:
	Java
	C
	C++
	Python
```

Java while loop

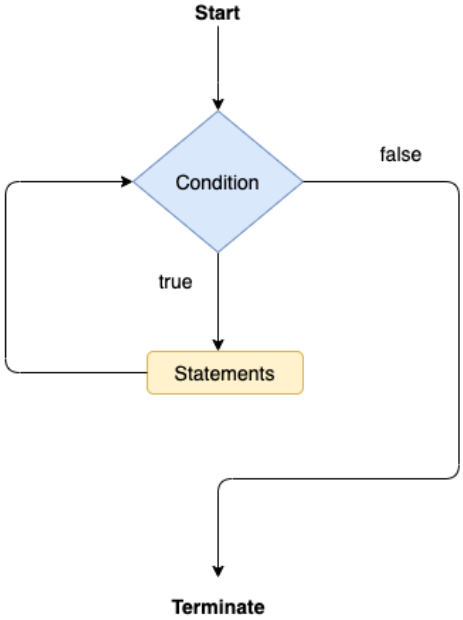
The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

Syntax:

```
while(condition){
        //looping statements
}
```

The flow chart for the while loop is given in the following image.



Example:

```
class Calculation {
        public static void main(String[] args) {
                int i = 0;
                while(i<=10) {
                        System.out.print(i+" ");
                        i = i + 2;
                }
        }
}
```

Output:

```
0 2 4 6 8 10
```

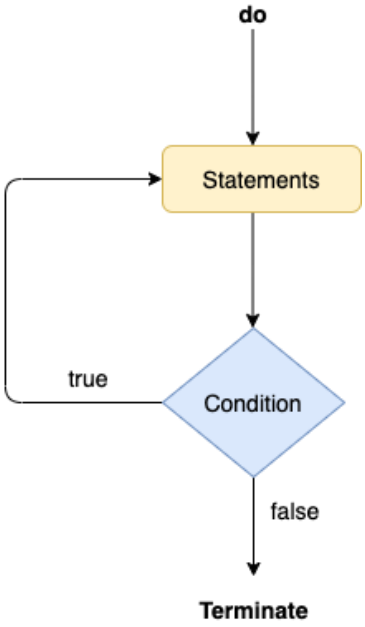
Java do-while loop

The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.

It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

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

The flow chart of the do-while loop is given in the following image.



Example:

```
class Calculation {
      public static void main(String[] args) {
            int i = 0;
            do {
                  System.out.print(i+" ");
                  i = i + 2;
            }while(i<=10);
      }
}
```

Output:

0 2 4 6 8 10

Jump Statements

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java, i.e., break and continue.

Java break statement

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop.

The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for loop.

Example:

```
class BreakExample {
	public static void main(String[] args) {
		for(int i = 0; i<= 10; i++) {
			System.out.print (i+" ");
			if(i==6) {
				break;
			}
		}
	}
}
```

Output:

0 1 2 3 4 5 6

Java continue statement

Unlike break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately. Consider the following example to understand the functioning of the continue statement in Java.

Example:

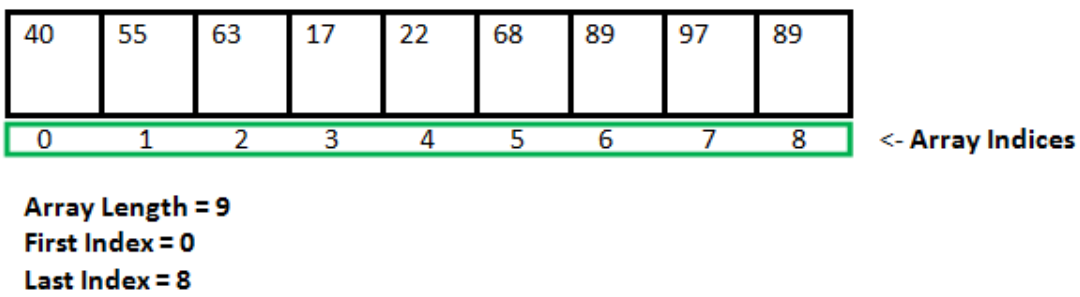
```
class ContinueExample {
	public static void main(String[] args) {
		for(int i = 0; i<= 2; i++) {
			for (int j = i; j<=5; j++) {
				if(j == 4) {
					continue;
				}
				System.out.print (j+" ");
			}
		}
	}
}
```

Output:

0 1 2 3 5 1 2 3 5 2 3 5

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.



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
```