

LEARNING MATERIAL

ON

OBJECT ORIENTED METHODOLOGY

(FOR 3RD SEMESTER CSE)

PREPARED BY:

PRIYANKA MAHARANA

GOVT.POLYTECHNIC,DHENKANAL

UNIT -I

OBJECT ORIENTED PROGRAMMING (OOPS) CONCEPTS

PROGRAMMING LANGUAGES

PROGRAM

A computer program is a collection of instructions that can be executed by a computer to perform a specific task. A computer program is usually written by a computer programmer in a programming language.

PROGRAMMING LANGUAGES

A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms.

Characteristics of a programming Language

- A programming language must be simple, easy to learn and use.
- A programming language must be consistent in terms of syntax and semantics.
- A programming language should be well structured and documented.
- Programming language's efficiency must be high.
- A portable programming language is always preferred.
- Necessary tools for development, debugging, testing, and maintenance of a program must be provided by a programming language.

Types of Computer Programming Languages

There are basically two types of computer programming languages given below:

- Low level language
- High level language
- **Low Level Languages**

The programming languages that are very close to machine code (0s and 1s) are called low-level programming languages.

The program instructions written in these languages are in binary form.

The examples of low-level languages are:

Machine language

Assembly language

MACHINE LANGUAGE

- **Machine language**, the numeric codes for the operations that a particular computer can execute directly.
- The codes are strings of 0s and 1s, or binary digits ("bits"), which are frequently converted both from and to hexadecimal (base 16) for human viewing and modification.
- This language is different for different computers.
- It is not easy to learn the machine language.
- **Advantage of Machine Language**
- It is very faster because no translation program is required for the CPU.
- **Disadvantage of Machine Language**
- Machine Dependent
- Difficult to Modify
- Difficult to Program

ASSEMBLY LANGUAGE

- An assembly language is a low-level programming languages designed for a specific type of processors.
- Assembly language is also known as second generation of programming language.
- With assembly language, a programmer writes instructions using symbolic instruction code.

Example:

MOV - move data from one location to another

ADD - add two values

SUB - subtract a value from another value

- **Advantage of Assembly Language**
- Easy to understand and use
- Easier to locate and correct errors
- Easy to modify
- Efficiency of machine language
- **Disadvantage of Assembly Languages**
- Machine dependent
- Knowledge of hardware required
- Machine level coding

HIGH LEVEL LANGUAGE

The programming languages that are close to human languages called the high-level languages.

Each high level language has its own rule and grammar for writing program instructions. These rules are called syntax of the language.

The program written in high level language must be translated to machine code before to run it.

The examples of high-level languages are:

FORTRAN, COBOL, Basic, Pascal, C, C++, Java

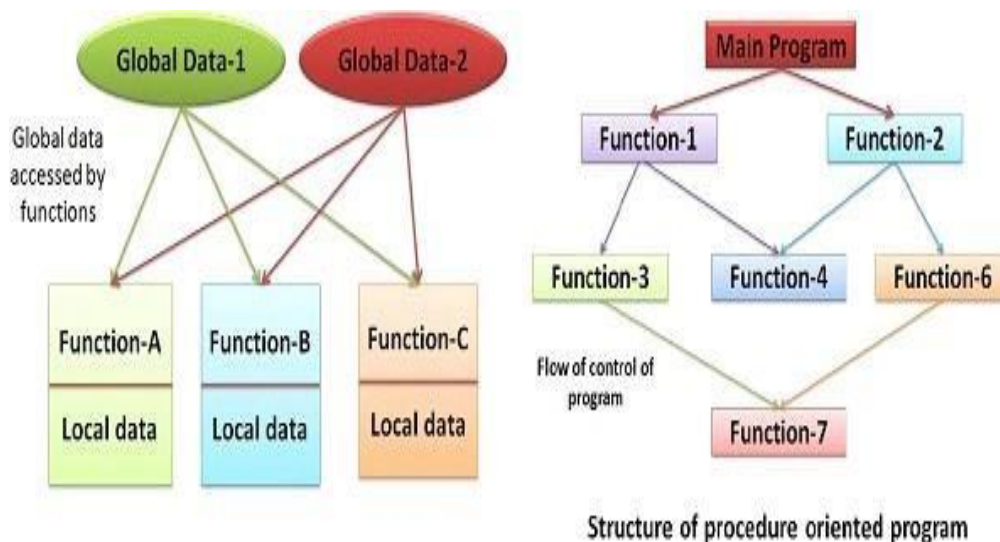
The high level programming languages are further divided into:

- Procedural languages
- Object oriented programming languages
- Non procedural languages

PROCEDURAL LANGUAGES

- It is derived from structured programming, based upon the concept of calling procedure.
- In procedural languages, the program code is written as a sequence of instructions.
- User has to specify “what to do” and also “how to do” (step by step procedure).
- These instructions are executed in the sequential order.
- These instructions are written to solve specific problems.
- Procedural programming follows top down approach.
- In procedural programming, program is divided into small parts called functions.
- In procedural programming, function is more important than data.
- Most of the function share global data.
- Adding new data and function is not easy.

Ex. FORTRAN, COBOL, Pascal, ADA and C.



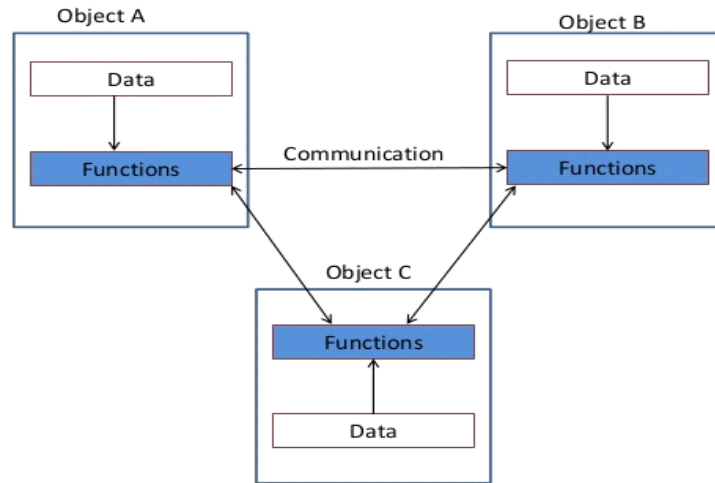
- **Advantages**
- Procedural Programming is excellent for general-purpose programming
- The coded simplicity along with ease of implementation of compilers and interpreters

- A large variety of books and online course material available on tested algorithms, making it easier to learn along the way
- The source code is portable, therefore, it can be used to target a different CPU as well
- The code can be reused in different parts of the program, without the need to copy it
- Through Procedural Programming technique, the memory requirement also slashes
- The program flow can be tracked easily
- **Disadvantages**
- The program code is harder to write when Procedural Programming is employed
- The Procedural code is often not reusable, which may pose the need to recreate the code if it is needed to use in another application
- Difficult to relate with real-world objects
- The importance is given to the operation rather than the data, which might pose issues in some data-sensitive cases
- The data is exposed to the whole program, making it not so much security friendly

OBJECT ORIENTED PROGRAMMING

- Object oriented programming can be defined as a programming model which is based upon the concept of objects.
- Objects contain data in the form of attributes and code in the form of methods.
- In object oriented programming, computer programs are designed using the concept of objects that interact with real world.
- Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.
- In object oriented programming, program is divided into small parts called objects.
- Object oriented programming follows bottom up approach.
- EX: Java, C++, C#, Python, PHP, JavaScript, Ruby, Perl, Objective-C, Dart, Swift, Scala.

Object Oriented Programming



Organization of data and functions in OOP

Non-Procedural Language

- In the non-procedural languages, the user has to specify only “what to do” and not “how to do”.
 - It is also known as an applicative or functional language. It involves the development of the functions from other functions to construct more complex functions.
 - It involves the development of the functions from other functions to construct more complex functions.
 - **Examples of Non-Procedural languages:**
SQL, PROLOG, LISP.
- Advantages of OOP**

- Due to modularity and encapsulation, OOP offers ease of management
- OOP mimics the real world, making it easier to understand
- Since objects are whole within themselves, they are reusable in other programs

Disadvantages of OOP

- Object-Oriented programs tend to be slower and use up a high amount of memory
- Over-generalization
- Programs built using this paradigm may take longer to be created

1.2 OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software.

The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.

• 1.3 BASIC CONCEPT OF OOP(OBJECT-ORIENTED PROGRAMMING):

There are some basic concepts of object-oriented programming as follows:

- Object
- Class
- Data abstraction
- Data encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- *Object*

Objects are important runtime entities in object oriented method.

They may characterize a location, a bank account, and a table of data or any entry that the program must handle.

Each object holds data and code to operate the data. Object can interact without having to identify the details of each other's data or code. It is sufficient to identify the type of message received and the type of reply returned by the objects.

example of object is CAR

Object: CAR
DATA
Colour Cost
METHODS LOCKIT() DRIVEIT()

Fig. 1.2 Representation of object CAR

- **CLASSES**

A class is a set of objects with similar properties (attributes), common behaviour (operations), and common link to other objects. The complete set of data and code of an object can be made a user-defined data type with the help of class.

The objects are variable of type class. A class is a collection of objects of similar type. Classes are user-defined data types and work like the built-in type of the programming language. Once the class has been defined, we can make any number of

objects belonging to that class. Each object is related with the data of type class with which they are formed.

As we learned that, the classification of objects into various classes is based on its properties (States) and behaviour (methods). Classes are used to distinguish a type of object from another. The important thing about the class is to identify the properties and procedures and applicability to its instances.

In example, we will create an object MH-01 1234 belonging to the class car. The object develops its distinctiveness from the difference in their attribute value and relationships to other objects.

- **DATA ABSTRACTION**

Data abstraction refers to the act of representing important description without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, cost and functions operate on these attributes. They summarize all the important properties of the objects that are to be created.

Classes use the concept of data abstraction and it is called as Abstract Data Type (ADT).

- **DATA ENCAPSULATION**

Data Encapsulation means wrapping of data and functions into a single unit (i.e. class). It is most useful feature of class. The data is not easy to get to the outside world and only those functions which are enclosed in the class can access it.

These functions provide the boundary between Object's data and program. This insulation of data from direct access by the program is called as **Data hiding**.

- **INHERITANCE**

Inheritance is the process by which objects of one class can get the properties of objects of another class. Inheritance means one class of objects inherits the data and behaviours from another class. Inheritance maintains the hierarchical classification in which a class inherits from its parents.

Inheritance provides the important feature of OOP that is reusability. That means we can include additional characteristics to an existing class without modification. This is possible by deriving a new class from an existing one.

In other words, it is a property of object-oriented systems that allow objects to be built from other objects. Inheritance allows open use of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The derived class holds the properties and behaviour of the base class in addition to the properties and behaviour of the derived class.

In Fig. 1.5, the Santro is a part of the class Hyundai which is again part of the class car and car is the part of the class vehicle. That means vehicle class is the parent class.

- **POLYMORPHISM**

(Poly means "many" and morph means "form"). Polymorphism means the ability to take more than one form. Polymorphism plays a main role in allocating objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific activities associated with each operation may differ. Polymorphism is broadly used in implementing inheritance.

It means objects that can take on or assume many different forms. Polymorphism means that the same operations may behave differently on different classes. Booch defines polymorphism as the relationship of objects of many different classes by some common super class. Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation detail to the objects involved.

For Example:

In a payroll system, manager, office staff and production worker objects all will respond to the compute payroll message, but the real operations performed are object particular.

- *DYNAMIC BINDING*

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code related with a given procedure call is not known until the time of the call at runtime.

Dynamic binding is associated with polymorphism and inheritance.

8. MESSAGE PASSING

In OOP, objects communicate with each other through member methods of the class. To establish communication between two objects, the following steps take place:

- a. Creating classes that declare variables and methods.
- b. Creating objects of classes that already declared.
- c. Calling methods through suitable data to establish communication between objects.

Message passing involves three elements: name of object, name of method, and information to be sent. For example, consider the below statement.

Employee.salary(name);

Here, Employee is the name of object, salary is the name of method and name is parameter that contains information.

1.4. BENEFIT OF OOPS

- Through inheritance, we can eliminate redundant code and extend the use of existing classes which is not possible in procedure oriented approach.
- We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch which happens in procedure oriented approach. This leads to saving of development time and higher productivity.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

- It is possible to have multiple instances of object to co-exist without any interference.
- It is possible to map objects in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects .
- The data-centered design approach enables us to capture more details of a model in implementable form.
- Object oriented systems can be easily upgraded from small to large systems.
- Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

UNIT-II

INTRODUCTION TO JAVA

WHAT IS JAVA

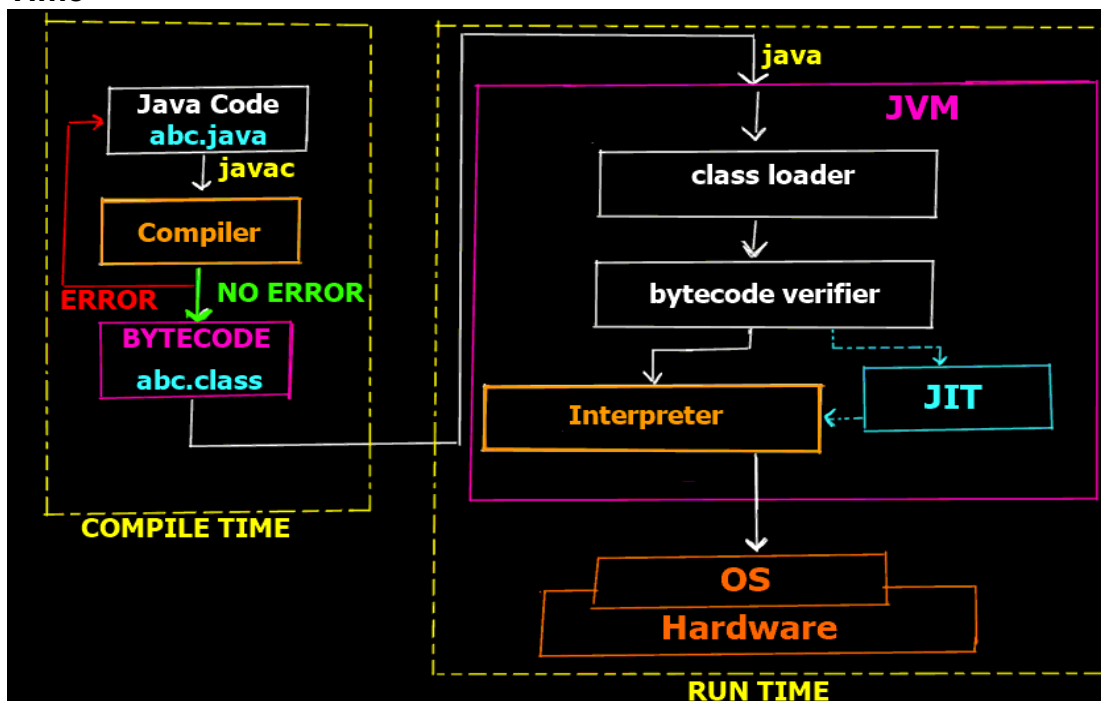
Java is a **programming language** and a **platform**. Java is a high level, robust, object-oriented and secure programming language.

Java was developed by *Sun Microsystems* (which is now the subsidiary of Oracle) in the year 1995. *James Gosling* is known as the father of Java. Before Java, its name was *Oak*. Since Oak was already a registered company, so James Gosling and his team changed the name from Oak to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

EXECUTION MODEL OF JAVA

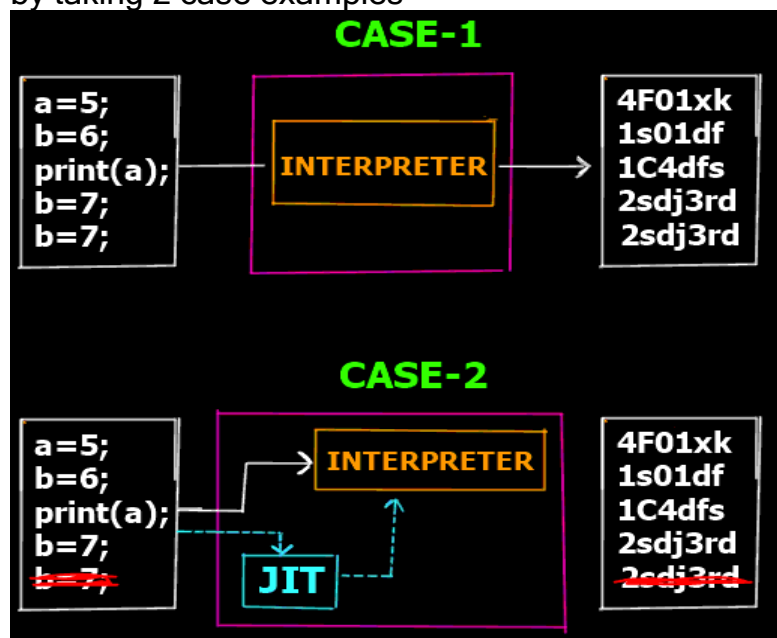
Lets first see a complete flow diagram and then we will understand the execution process step by step. Note that there basically are 2 phases - **Compile time** and **Run Time** –



- **Step 1:** Writing the code in NetBeans IDE. As we can see in the diagram step 1 is to actually type the code. In this image above you can see we have our code file as **abc.java**
- **Step 2:** Once we have written the code save it and click Run (if you are using Netbeans IDE otherwise you compile the code in the command prompt by using the command `javac abc.java`).
This invokes the Java Compiler. The compiler checks the code for syntax errors and any other compile time errors and if no error is found the compiler converts the java code into an intermediate code(**abc.class** file) known as **bytecode**.
This **intermediate code is platform independent** (you can take this bytecode from a machine running windows and use it in any other machine running Linux or MacOS etc). Also this bytecode is an intermediate code, hence it is only understandable by the **JVM** and not the user or even the hardware /OS layer.
- **Step 3:** This is the start of the Run Time phase, where the bytecode is loaded into the JVM by the **class loader**(another inbuilt program inside the JVM).
- **Step 4:** Now the **bytecode verifier**(an inbuilt program inside the JVM) checks the bytecode for its integrity and if not issues are found passes it to the interpreter.
- **Step 5:** Since java is both compiled and interpreted language, now the interpreter inside the JVM converts each line of the bytecode into executable machine code and passed it to the OS/Hardware i.e. the CPU to execute.
These are the standard steps involved in a typical java program execution scenario.

Working of JIT (Just-In-Time) Compiler

In the above steps we didn't mention the working of **JIT compiler**. Lets understand this by taking 2 case examples –



- **Case 1:**
- In case 1 you can see that we are at the interpretation phase (Step 5 of the overall program execution). Let's assume we have 5 lines which are supposed to be interpreted to their corresponding machine code lines. So as you can see in the Case 1 there is no JIT involved. Thus the interpreter converts each line into its corresponding machine code line. However if you notice the last 2 lines are the same (consider it a redundant line inserted by mistake). Clearly that line is redundant and does not have any effect on the actual output but yet since the interpreter works line by line it still creates 5 lines of machine code for 5 lines of the bytecode.
- Now this is inefficient right? lets see case 2
- **Case 2:**
- In case 2 we have the JIT compiler. Now before the bytecode is passed onto the interpreter for conversion to machine code, the **JIT compiler scans** the full code to see if it can be **optimized**. As it finds the last line is redundant it removes it from the bytecode and passes only 4 lines to the interpreter thus making it more **efficient** and **faster** as the interpreter now has 1 line less to interpret.
- So this is how JIT compiler speeds up the overall execution process.
This was just one scenario where JIT compiler can help in making the execution process fast and efficient. There are other cases like *inclusion of only those packages needed in the code, code optimizations, redundant code removal* etc which overall makes the process very fast and efficient. Also different JITs developed by different companies work differently and JIT compilers are an **optional** step and not invoked everytime.
So this was the complete Execution Process of Java Program in Detail with the Working of JIT Compiler.

THE JAVA VIRTUAL MACHINE

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java byte code can be executed.

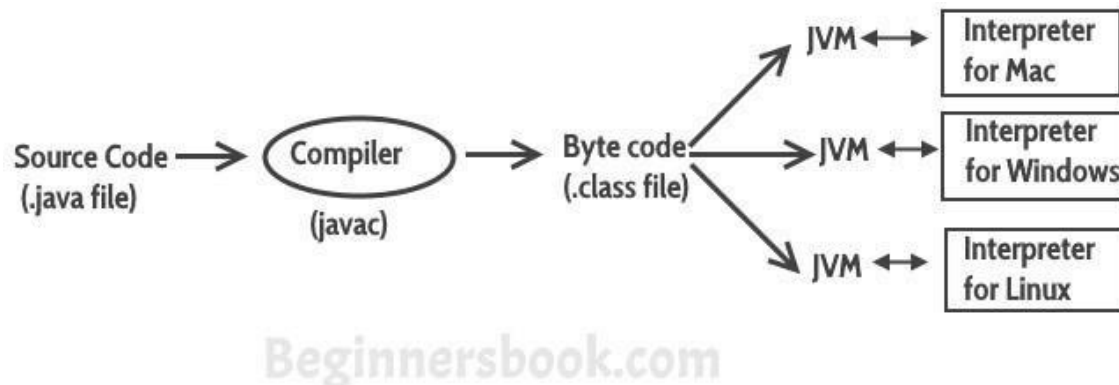
java is a high level programming language. A program written in high level language cannot be run on any machine directly. First, it needs to be translated into that particular machine language. The **javac compiler** does this thing, it takes java program (.java file containing source code) and translates it into machine code (referred as byte code or .class file).

Java Virtual Machine (JVM) is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code**. This makes it easier for compiler as it has to generate byte code for JVM rather than different machine code for each type of machine. JVM executes the byte code generated by compiler and produce output. **JVM is the one that makes java platform independent.**

So, now we understood that the primary function of JVM is to execute the byte code produced by compiler. **Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.** Which means that the byte code generated on Windows can be run on Mac

OS and vice versa. That is why we call java as platform independent language. The same thing

an be seen in the diagram below:



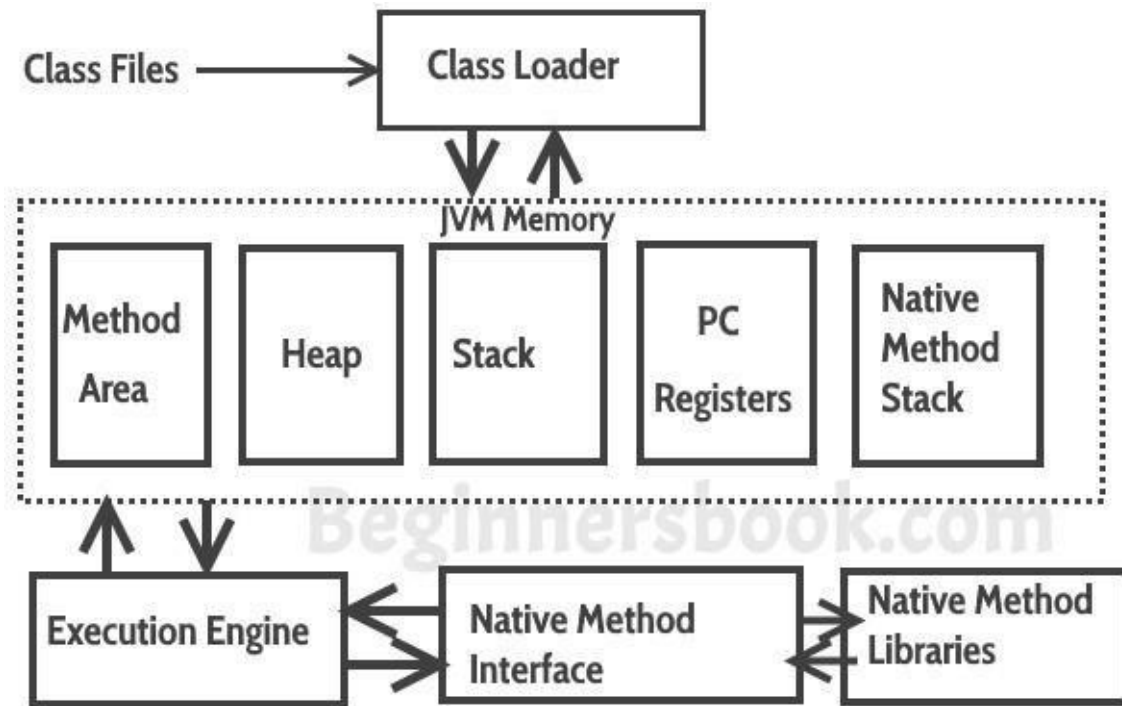
So to summarise everything: The Java Virtual machine (JVM) is the virtual machine that runs on actual machine (your computer) and executes Java byte code. The JVM doesn't understand Java source code, that's why we need to have javac compiler that compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM. JVM makes java portable (write once, run anywhere). Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.

WHAT IT DOES

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM Architecture



Lets see how JVM works:

Class Loader: The class loader reads the .class file and save the byte code in the **method area**.

Method Area: There is only one method area in a JVM which is shared among all the classes. This holds the class level information of each .class file.

Heap: Heap is a part of JVM memory where objects are allocated. JVM creates a Class object for each .class file.

Stack: Stack is a also a part of JVM memory but unlike Heap, it is used for storing temporary variables.

PC Registers: This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.

Native Method stack: A native method can access the runtime data areas of the virtual machine.

Native Method interface: It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.

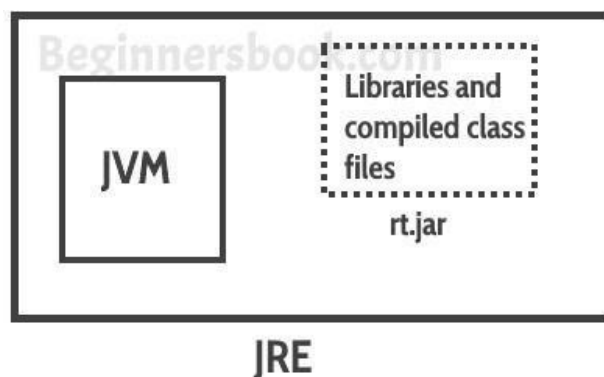
Garbage collection: A class instance is explicitly created by the java code and after use it is automatically destroyed by garbage collection for memory management.

JVM Vs JRE Vs JDK

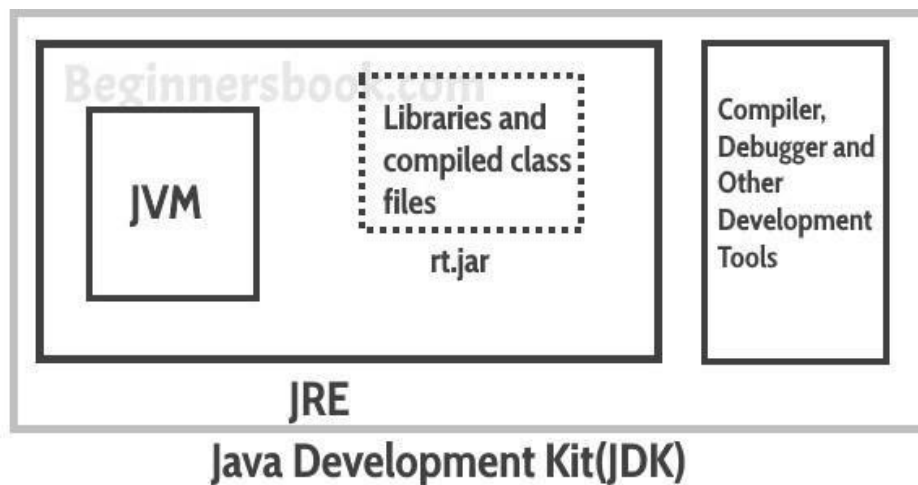
JRE: JRE is the environment within which the java virtual machine runs. JRE contains Java virtual Machine(JVM), class libraries, and other files excluding development tools such as compiler and debugger.

Which means you can run the code in JRE but you can't develop and compile the code in JRE.

JVM: As we discussed above, JVM runs the program by using class, libraries and files provided by JRE.



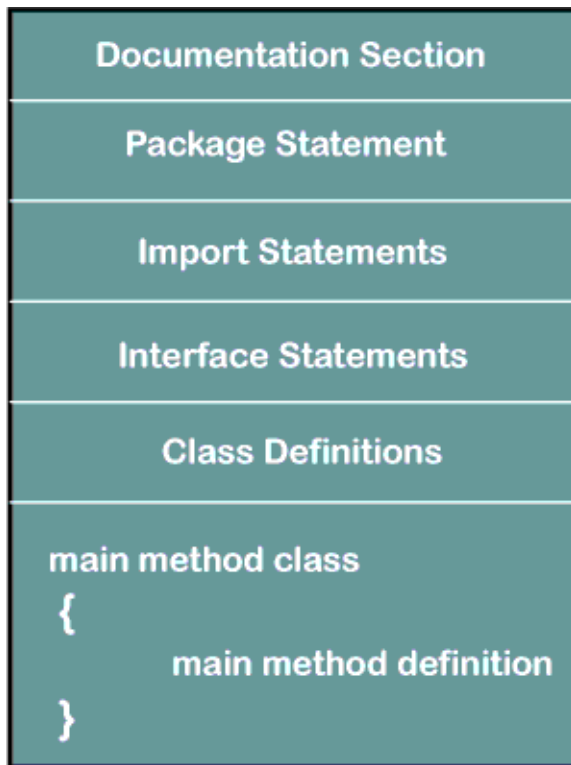
JDK: JDK is a superset of JRE, it contains everything that JRE has along with development tools such as compiler, debugger etc.



STRUCTURE OF JAVA PROGRAM

A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviors



Structure of Java Program

Documentation Section

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. To write the statements in the

documentation section, we use **comments**. The comments may be **single-line**, **multi-line**, and **documentation** comments.

- **Single-line Comment:** It starts with a pair of forwarding slash (**//**). For example:

- `//First Java Program`

- **Multi-line Comment:** It starts with a **/*** and ends with ***/**. We write between these two symbols. For example:

- `/*It is an example of`
 - `multiline comment*/`

- **Documentation Comment:** It starts with the delimiter (**/****) and ends with ***/**. For example:

- `/**It is an example of documentation comment*/`

Package Declaration

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. example:

- **package** javatpoint; //where javatpoint is the package name
- **package** com.javatpoint; //where com is the root directory and javatpoint is the subdirectory

Import Statements

The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. . For example:

- **import** java.util.Scanner; //it imports the Scanner class only
- **import** java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the **implements** keyword. An interface can also be used with other interfaces by using the **extends** keyword. For example:

interface car

{

```
void start();  
void stop();  
}
```

Class Definition

In this section, we define the class. It is **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may contain more than one class definition. We use the **class** keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the `main()` method. For example:

```
class Student //class definition  
  
{  
  
}
```

Main Method Class

In this section, we define the **main() method**. It is essential for all Java programs. Because the execution of all Java programs starts from the `main()` method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the `main()` method:

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

For example:

```
public class Student //class definition  
  
{  
  
public static void main(String args[])  
  
{  
//statements  
}  
}
```

A FIRST JAVA PROGRAM

First Java Program

```
class Simple{  
    public static void main(String args[]){  
        System.out.println("Hello Java");  
    }  
}
```

Save the above file as Simple.java.

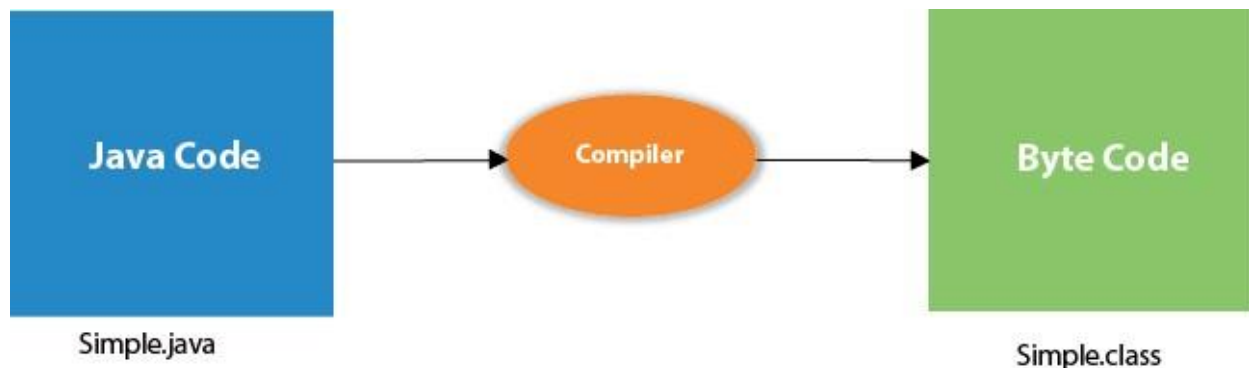
To compile:	javac Simple.java
To execute:	java Simple

Output:

Hello Java

Compilation Flow:

When we compile Java program using javac tool, the Java compiler converts the source code into byte code.



PARAMETERS USED IN FIRST JAVA PROGRAM

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method.

The core advantage of the static method is that there is no need to create an object to

invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.

- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args** or **String args[]** is used for command line argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. We will discuss the internal working of System.out.println() statement in the coming section.

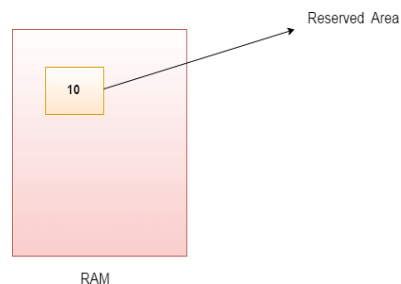
VARIABLES AND DATA TYPES

JAVA VARIABLES

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type.

Variable is a name of memory location. There are three types of variables in java: local, instance and static.

A variable is the name of a reserved area allocated in memory. In other words, it is a name of the memory location. It is a combination of "vary + able" which means its value can be changed Java



- **int data=50;**//Here data is variable

TYPES OF VARIABLES

There are three types of variables in Java:

- local variable

- instance variable
- static variable

1) LOCAL VARIABLE

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

2) INSTANCE VARIABLE

A variable declared inside the class but outside the body of the method, is called an instance variable. It is not declared as **static**.

It is called an instance variable because its value is instance-specific and is not shared among instances.

3) STATIC VARIABLE

A variable that is declared as static is called a static variable. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

PRIMITIVE DATATYPES & DECLARATIONS

EXAMPLE TO UNDERSTAND THE TYPES OF VARIABLES IN JAVA

public class A

```
{  
    static int m=100;//static variable  
    void method()  
    {  
        int n=90;//local variable  
    }  
    public static void main(String args[])  
    {  
        int data=50;//instance variable  
    }  
}
```

```
}//end of class
```

JAVA VARIABLE EXAMPLE: ADD TWO NUMBERS

```
public class Simple{  
    public static void main(String[] args){  
        int a=10;  
        int b=10;  
        int c=a+b;  
        System.out.println(c);  
    }  
}
```

Output:

20

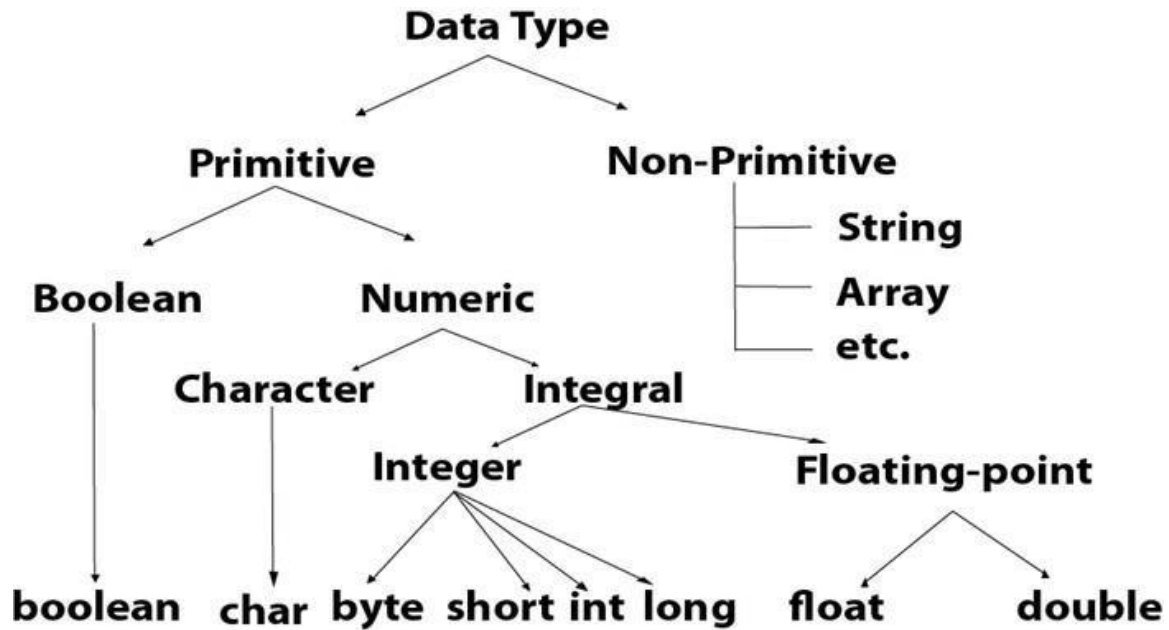
DATA TYPES IN JAVA

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

- **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
- **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

JAVA PRIMITIVE DATA TYPES

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.



here are eight primitive data types in Java:

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

		9,223,372,036,854,775,807
float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
char	2 bytes	Stores a single character/letter or ASCII values

byte

- Byte data type is an 8-bit signed two's complement integer
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive) ($2^7 - 1$)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an integer.
- Example: byte a = 100, byte b = -50

short

- Short data type is a 16-bit signed two's complement integer
- Minimum value is -32,768 (-2^{15})
- Maximum value is 32,767 (inclusive) ($2^{15} - 1$)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an integer
- Default value is 0.
- Example: short s = 10000, short r = -20000

int

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is -2,147,483,648 (-2^{31})
- Maximum value is 2,147,483,647 (inclusive) ($2^{31} - 1$)
- Integer is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0
- Example: `int a = 100000, int b = -200000`

long

- Long data type is a 64-bit signed two's complement integer
- Minimum value is -9,223,372,036,854,775,808 (-2^{63})
- Maximum value is 9,223,372,036,854,775,807 (inclusive) ($2^{63} - 1$)
- This type is used when a wider range than int is needed
- Default value is 0L
- Example: `long a = 100000L, long b = -200000L`

float

- Float data type is a single-precision 32-bit IEEE 754 floating point
- Float is mainly used to save memory in large arrays of floating point numbers
- Default value is 0.0f
- Float data type is never used for precise values such as currency
- Example: `float f1 = 234.5f`

double

- double data type is a double-precision 64-bit IEEE 754 floating point
- This data type is generally used as the default data type for decimal values, generally the default choice
- Double data type should never be used for precise values such as currency
- Default value is 0.0d
- Example: `double d1 = 123.4`

boolean

- boolean data type represents one bit of information
- There are only two possible values: true and false
- This data type is used for simple flags that track true/false conditions
- Default value is false
- Example: `boolean one = true`

char

- char data type is a single 16-bit Unicode character
- Minimum value is '\u0000' (or 0)
- Maximum value is '\uffff' (or 65,535 inclusive)
- Char data type is used to store any character
- Example: char letterA = 'A'

NON-PRIMITIVE DATA TYPES

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for **String**).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be **null**.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Examples of non-primitive types are Strings, Arrays,

NUMERIC AND CHARACTER LITERALS

LITERALS IN JAVA

Literals are number, text, or anything that represent a value. In other words, Literals in Java are the constant values assigned to the variable. It is also called a constant.

For example,

```
int x = 100;
```

So, 100 is literal.

Literals in Java can be classified into four types-

- Integer Literals
- Floating-point Literals
- Character and String Literals
- Boolean literals

INTEGER LITERALS IN JAVA

Literal assigned to a type byte, short, int or long is called an integer literal in Java. Make sure when you assign a literal value to a byte or short it is within the range of that particular type.

An integer literal is of type long if it ends with the letter L or l; otherwise it is of type int. It is recommended that you use the upper case letter L because the lower case letter l is hard to distinguish from the digit 1.

You will generally use a base 10 number i.e. decimal as an integer literal. But integer literals can be expressed by Binary (base two) and hexadecimal (base 16) number system also. Integer literals can be expressed by these number systems:

- **Decimal:** Base 10, whose digits consists of the numbers 0 through 9; this is the number system you use every day.
- **Hexadecimal:** Base 16, whose digits consist of the numbers 0 through 9 and the letters A through F.
- **Binary:** Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later).

As example if you want to write 50 using hexadecimal -

```
int hNum = 0x32;  
System.out.println("" + hNum);
```

Same way, if you want to use binary system to assign literal 50 -

```
int bnum = 0b110010;  
System.out.println("" + bnum);
```

Note - The prefix 0x indicates hexadecimal and 0b indicates binary.

FLOATING-POINT LITERALS IN JAVA

Literal assigned to a type float or double is called floating-point literal in Java. A floating-point literal is of type float if it ends with the letter F or f; otherwise its type is double in that case it can end with the letter D or d but that is optional.

Floating-point literal can also be expressed using E or e (for scientific notation).

As example

```
double dNum1 = 156.7;  
// same value as d1, but in scientific notation  
double dNum2 = 1.567e2;  
float fNum = 34.8f;
```

CHARACTER AND STRING LITERALS IN JAVA

Literals of types `char` and `String` may contain any Unicode (UTF-16) characters. Always use 'single quotes' for `char` literals and "double quotes" for `String` literals. The Java programming language also supports a few special escape sequences for `char` and `String` literals: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `'` (single quote), and `\\` (backslash).

As example if you want to assign single quote (') itself to a `char` variable you need escape sequence in that case.

```
char c = "\";
```

```
System.out.println(c);
```

That will give ' as output.

Refer [How to add double quotes to a String](#) for a Java program showing how to add double quotes to a `String`.

Example of String literals

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

```
String lines = "This is the first line.\n"+
```

```
    "This is the second line.\n"+
```

```
    "This is the third line.\n";
```

One thing to note here is that `String` is not a primitive type when you create a `String` literal a reference to an instance of `String` class is created.

Boolean literal in Java

Boolean literal can have only one of two values `true` and `false`. A Boolean literal can only be assigned to a variable of type `boolean`. It can also be used with conditions which evaluate to a boolean value.

As example

```
int num1 = 9;
```

```
int num2 = 7;
```

```
if(num1 > num2 == true){
```

```
    System.out.println(" num1 is greater than num2");
```

```
}
```

Here if condition `num1 > num2` evaluates to either `true` or `false`. Though it can be expressed implicitly also (preferred)

```
int num1 = 9;
```

```
int num2 = 7;
```

```
if(num1 > num2){
```

```
    System.out.println(" num1 is greater than num2");
```

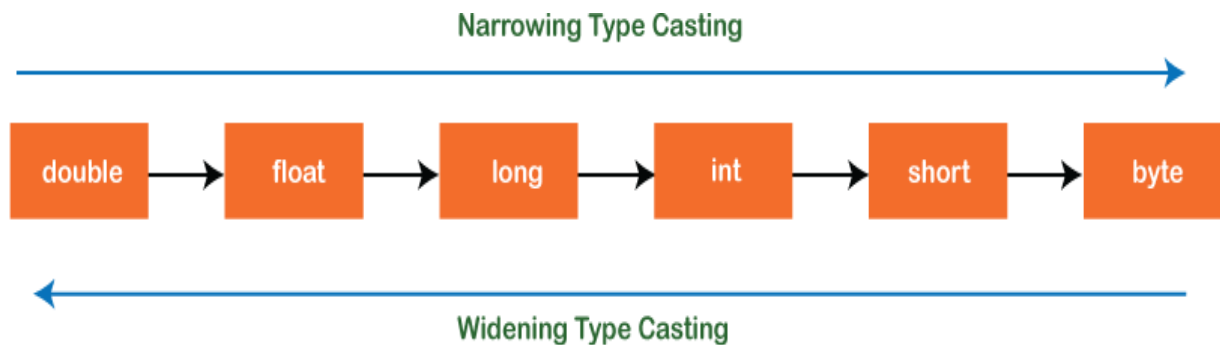
```
}
```

Here you can see with if only expression is used (`num1 > num2`) as it will evaluate to `true` or `false` anyway.

CASTING AND TYPE CASTING

TYPE CASTING IN JAVA

In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer. In this section, we will discuss **type casting** and **its types** with proper examples.



Type Casting in Java

TYPE CASTING

Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting

There are two types of type casting:

- Widening Type Casting
- Narrowing Type Casting

WIDENING AND NARROWING CONVERSIONS

WIDENING TYPE CASTING

Converting a lower data type into a higher one is called **widening** type casting. It is also known as **implicit conversion** or **casting down**. It is done automatically. It is safe because there is no chance to lose data. It takes place when:

- Both data types must be compatible with each other.
- The target type must be larger than the source type.
- **byte -> short -> char -> int -> long -> float -> double**

For example, the conversion between numeric data type to char or Boolean is not done automatically. Also, the char and Boolean data types are not compatible with each other. Let's see an example.

WideningTypeCastingExample.java

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
}
}
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

In the above example, we have taken a variable x and converted it into a long type. After that, the long type is converted into the float type.

NARROWING TYPE CASTING

Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

- **double -> float -> long -> int -> char -> short -> byte**

Let's see an example of narrowing type casting.

In the following example, we have performed the narrowing type casting two times. First, we have converted the double type into long data type after that long data type is converted into int type.

NarrowingTypeCastingExample.java

```
public class NarrowingTypeCastingExample
{
public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
}
}
```

Output

```
Before conversion: 166.66
After conversion into long type: 166
After conversion into int type: 166
```

OPERATORS AND EXPRESSIONS

OPERATORS IN JAVA

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 –

Show Examples

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 –

Show Examples

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

The following table lists the bitwise operators –

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

Show Examples

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 –

Show Examples

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 –

Show Examples

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 $C = C >> 2$

		>> 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

Following is an example –

Example

[Live Demo](#)

```
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

Live Demo

```
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

CONTROL FLOW STATEMENT

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.

- Decision Making statements
- if statements

- switch statement
- Loop statements
- do while loop
- while loop
- for loop
- for-each loop
- Jump statements
- break statement
- continue statement

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.

- Simple if statement
- if-else statement
- if-else-if ladder
- 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 of if statement is given below.

```
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.

Student.java

Student.java

```
public 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.

Student.java

```
public 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 of if-else-if statement is given below.

```
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.

Student.java

```
public class Student {  
    public static void main(String[] args) {  
        String city = "Delhi";  
        if(city == "Meerut") {  
            System.out.println("city is meerut");  
        }else if (city == "Noida") {  
            System.out.println("city is noida");  
        }else if(city == "Agra") {  
            System.out.println("city is agra");  
        }else {  
            System.out.println(city);  
        }  
    }  
}
```

Output:

Delhi

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 of Nested if-statement is given below.

- **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
- }

- }

Output:

Delhi

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.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.
- 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 syntax to use the switch statement is given below.

switch (expression){

case value1:

 statement1;

break;

 .

 .

 .

case valueN:

 statementN;

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

Consider the following example to understand the flow of the switch statement.

Student.java

```
public 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

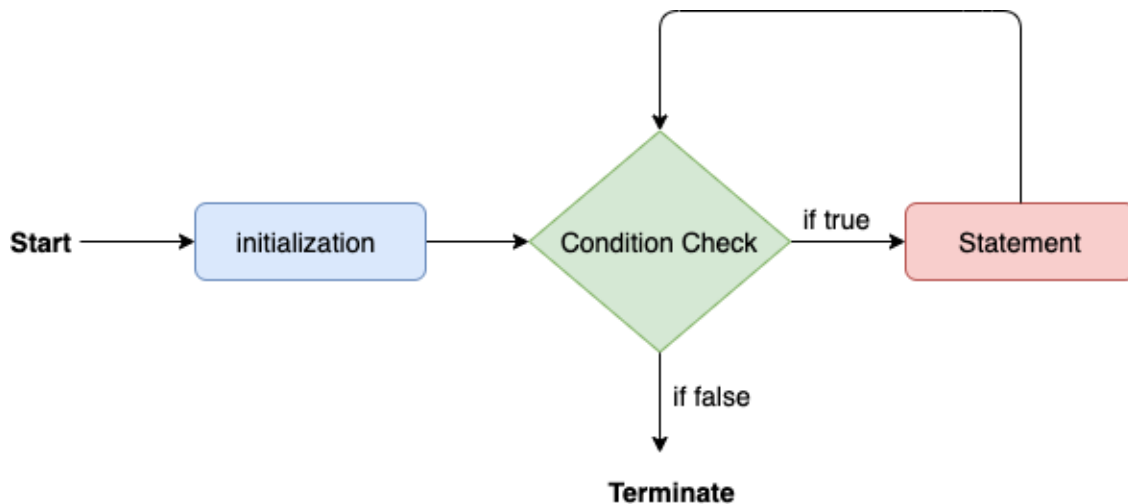
Let's understand the loop statements one by one.

Java for loop

. 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.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculation {  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        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.

Calculation.java

```

public class Calculation {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String[] names = {"Java","C","C++","Python","JavaScript"};
        System.out.println("Printing the content of the array names:\n");
        for(String name:names) {
            System.out.println(name);
        }
    }
}

```

Output:

```
Printing the content of the array names:
```

Java
C
C++
Python
JavaScript

JAVA WHILE LOOP

The [while loop](#)

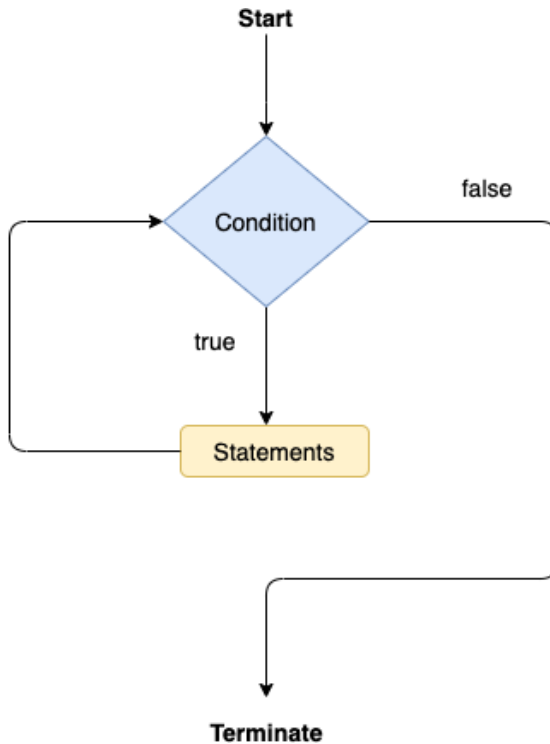
is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

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.

The syntax of the while loop is given below.

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

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



Consider the following example.

Calculation .java

```
public class Calculation {  
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    int i = 0;  
    System.out.println("Printing the list of first 10 even numbers \n");  
    while(i<=10) {  
        System.out.println(i);  
        i = i + 2;  
    }  
}
```

Output:

Printing the list of first 10 even numbers

0
2
4
6
8
10

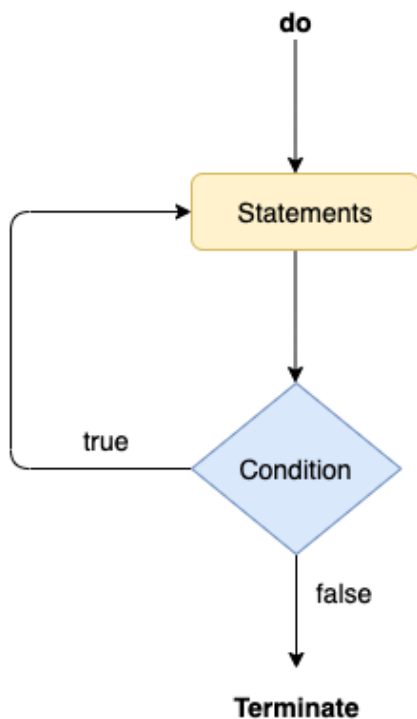
JAVA DO-WHILE LOOP

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.



Consider the following example to understand the functioning of the do-while loop in Java.

Calculation.java

```
public class Calculation {  
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
int i = 0;  
    System.out.println("Printing the list of first 10 even numbers \n");  
    do {  
        System.out.println(i);  
        i = i + 2;  
    }while(i<=10);  
}  
}
```

Output:

Printing the list of first 10 even numbers

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.

BreakExample.java

```
public class BreakExample {  
  
    public static void main(String[] args) {  
        // TODO Auto-generated method stub  
        for(int i = 0; i<= 10; i++) {  
            System.out.println(i);  
            if(i==6) {  
                break;  
            }  
        }  
    }  
}
```

Output:

```
0  
1  
2  
3  
4  
5  
6
```

break statement example with labeled for loop

Calculation.java

```
public class Calculation {  
  
    public static void main(String[] args) {
```

```
// TODO Auto-generated method stub
```

```
a:
```

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

```
b:
```

```
for(int j = 0; j<=15;j++) {
```

```
c:
```

```
for (int k = 0; k<=20; k++) {
```

```
System.out.println(k);
```

```
if(k==5) {
```

```
break a;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Output:

```
0
1
2
3
4
5
```

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.

```
public class ContinueExample {
```



```
public static void main(String[] args) {  
    // TODO Auto-generated method stub
```

```
for(int i = 0; i<= 2; i++) {
```

```
    for (int j = i; j<=5; j++) {
```

```
        if(j == 4) {
```

```
            continue;
```

```
        }
```

```
        System.out.println(j);
```

```
    }
```

```
}
```

```
}
```

```
}
```

Output:

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

UNIT-III

OBJECT AND CLASSES

A class is a user defined blueprint or prototype from which objects are created.

It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

- **Fields**
- **Methods**
- **Constructors**
- **Blocks**
- **Nested class and interface**

SYNTAX TO DECLARE A CLASS:

```
class <class_name>{  
    field;  
    method;  
}
```

INSTANCE VARIABLE IN JAVA

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

METHOD IN JAVA

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

NEW KEYWORD IN JAVA

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area

ADDING VARIABLE & METHODS

We can add variable and data field to a class. A class with only data fields has no life. The object created by such a class can't respond to any message. We must therefore add methods that are necessary for manipulating the data contain in the class.

Syntax of method declaration:

type method name (parameter list)

{

method body;

}

* The type specify the type of value method would return.

* The parameter list contains the variable name and types of all the values we want to give to the method as input.

Example:

Class rectangle

{

int l;

int w;

void get data(int x,int y)

{

l=x;

```
w=y;
```

```
}
```

```
}
```

OBJECT AND CLASS EXAMPLE: MAIN WITHIN THE CLASS

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

1. *//Java Program to illustrate how to define a class and fields*
2. **class** Student{
3. **int** id;*//field or data member or instance variable*
4. String name;
5. **public static void** main(String args[])
6. {
7. *//Creating an object or instance*
8. Student s1=**new** Student();*//creating an object of Student*
9. *//Printing values of the object*
10. System.out.println(s1.id);*//accessing member through reference variable*
11. System.out.println(s1.name);
12. }
13. }

it Now

Output:

0

null

OBJECT AND CLASS EXAMPLE: MAIN OUTSIDE THE CLASS

In real time development, we create classes and use it from another class. It is a better

approach than previous one. Let's see a simple example, where we are having main () method in another class.

We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main() method.

File: TestStudent1.java

1. *//Java Program to demonstrate having the main method in*
2. *//another class*
3. *//Creating Student class.*
4. **class** Student{
5. **int** id;
6. String name;
7. }
8. *//Creating another class TestStudent1 which contains the main method*
9. **class** TestStudent1{
10. **public static void** main(String args[]){
11. Student s1=**new** Student();
12. System.out.println(s1.id);
13. System.out.println(s1.name);
14. }
15. }

Test it Now

Output:

0

Null

OBJECT

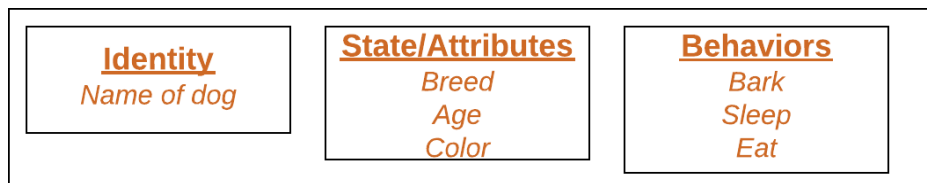
It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

State: It is represented by attributes of an object. It also reflects the properties of an object.

Behavior: It is represented by methods of an object. It also reflects the response of an object with other objects.

Identity: It gives a unique name to an object and enables one object to interact with other objects.

Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

ASSESSING CLASS NUMBER

It is important to understand that each object has its own copy of instance variable of its class. This means that any changes to the variable of an object have no effect on the variable of another.

All variable must be assigned value before they are used. But we cannot access the instance variable and method directly to do this we must use the concern object and dot (.) operator.

3 WAYS TO INITIALIZE OBJECT

There are 3 ways to initialize object in Java.

1. By reference variable
2. By method
3. By constructor

1) OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH REFERENCE

Initializing an object means storing data into the object. Let's see a simple example

where we are going to initialize the object through a reference variable.

File: TestStudent2.java

```
1. class Student{
2.     int id;
3.     String name;
4. }
5. class TestStudent2{
6.     public static void main(String args[]){
7.         Student s1=new Student();
8.         s1.id=101;
9.         s1.name="Sonoo";
10.        System.out.println(s1.id+" "+s1.name);//printing members with a white space
11.    }
12. }
```

Output:

101 Sonoo

We can also create multiple objects and store information in it through reference variable.

2) OBJECT AND CLASS EXAMPLE: INITIALIZATION THROUGH METHOD

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

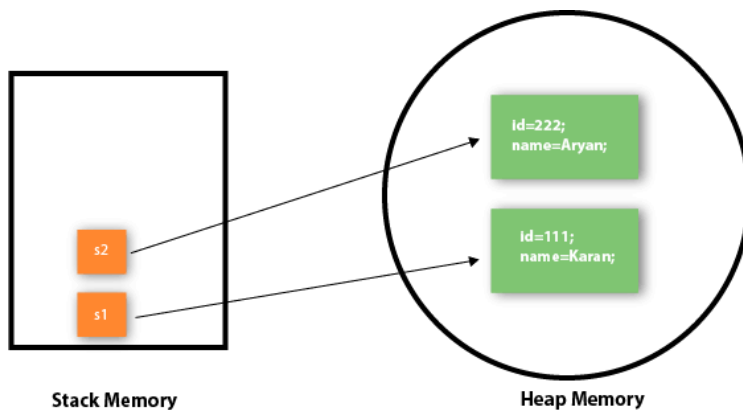
```
1. class Student{
2.     int rollno;
3.     String name;
4.     void insertRecord(int r, String n){
5.         rollno=r;
6.         name=n;
7.     }
8.     void displayInformation(){System.out.println(rollno+" "+name);}
9. }
10. class TestStudent4{
11.     public static void main(String args[]){
12.         Student s1=new Student();
```

```

13. Student s2=new Student();
14. s1.insertRecord(111,"Karan");
15. s2.insertRecord(222,"Aryan");
16. s1.displayInformation();
17. s2.displayInformation();
18. }
19. }

```

Output:
111 Karan
222 Aryan



As you can see in the above figure, object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

File: TestEmployee.java

```

1. class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {

```



```

6.     id=i;
7.     name=n;
8.     salary=s; 9.
        }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.     Employee e1=new Employee();
15.     Employee e2=new Employee();
16.     Employee e3=new Employee();
17.     e1.insert(101,"ajeet",45000);
18.     e2.insert(102,"irfan",25000);
19.     e3.insert(103,"nakul",55000);
20.     e1.display();
21.     e2.display();
22.     e3.display();
23. }
24. }

```

Test it Now

Output:

```

101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0

```

Object and Class Example: Rectangle

There is given another example that maintains the records of Rectangle class.

File: TestRectangle1.java

```

1. class Rectangle{
2.     int length;
3.     int width;
4.     void insert(int l, int w){
5.         length=l;
6.         width=w;

```

```
7.  }
8.  void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11. public static void main(String args[]){
12.     Rectangle r1=new Rectangle();
13.     Rectangle r2=new Rectangle();
14.     r1.insert(11,5);
15.     r2.insert(3,15);
16.     r1.calculateArea();
17.     r2.calculateArea();
18. }
19. }
```

Test it Now

Output:

55

45

CONSTRUCTORS IN JAVA

In Java , a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called. It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

1. `<class_name>(){}`

Example of default constructor that displays the default values

1. `//which displays the default values`
2. `class Student3{`
3. `int id;`
4. `String name;`
5. `//method to display the value of id and name`
6. `void display(){System.out.println(id+" "+name);}`
7.
8. `public static void main(String args[]){`
9. `//creating objects`
10. `Student3 s1=new Student3();`
11. `Student3 s2=new Student3();`
12. `//displaying values of the object`
13. `s1.display();`
14. `s2.display();`
15. `}`

16. }

Test it Now

Output:

0 null

0 null

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

1. *//Java Program to demonstrate the use of the parameterized constructor.*
2. **class** Student4{
3. **int** id;
4. String name;
5. *//creating a parameterized constructor*
6. Student4(**int** i,String n){
7. id = i;
8. name = n;
9. }
10. *//method to display the values*
11. **void** display(){System.out.println(id+" "+name);}
- 12.

```
13. public static void main(String args[]){
14.     //creating objects and passing values
15.     Student4 s1 = new Student4(111,"Karan");
16.     Student4 s2 = new Student4(222,"Aryan");
17.     //calling method to display the values of object
18.     s1.display();
19.     s2.display();
20. }
21. }
```

Test it Now

Output:

111 Karan

222 Aryan

ACCESS MODIFIERS IN JAVA

Access modifiers are used to specify the accessibility or access levels of a type (class, interface) and its members (methods, variables and even constructors). There are three access modifiers and four access levels in Java. The three access modifiers are private, protected and public. Four access levels (from most restricted to least restricted) are private, default (no modifier), protected and public. If no access modifier is specified, it is called a default access level.

Access levels and their accessibility can be summarized as:

Private: Same class.

Default: Same class, Same package.

Protected: Same class, Same package, Subclasses.

Public: Same class, Same package, Subclasses, Everyone.

The top level classes (classes not within another class) have only public and default access; but for inner classes (classes within classes) have all four access levels.

Private

Private members are accessible only within the same class and also are not inherited.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and

private method. We are accessing these private members from outside the class, so there is a compile-time

error.

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5.
6. public class Simple{
7. public static void main(String args[]){
8. A obj=new A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }Default
```

Default (no modifier) members can be accessed by members of the same class and members of any class in the same package.

Default members are inherited by another class only if both parent and child are in same package.

If you don't use any modifier, it is treated as default by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1. //save by A.java
2. package pack;
3. class A{
4. void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B{
5. public static void main(String args[]){
6. A obj = new A();//Compile Time Error
7. obj.msg();//Compile Time Error
8. }
9. }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Protected

Protected members can be accessed by

- ❑ members of the same class
- ❑ members of any class in the same package (same as default)
- ❑ subclasses (any level of subclass hierarchy) in other packages (only through inheritance).

Protected members (static or instance) cannot be accessed from a non-subclass in another package.

Inherited protected members in the child class cannot be accessed using a parent reference variable (irrespective of the object it point to at runtime).

Static protected members can be accessed in subclasses through:

- ❑ object reference (Parent or Child) in subclasses
- ❑ class name

The protected access modifier is accessible within package and outside the package but through

inheritance only.

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

It provides more accessibility than the default modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public,

so can be accessed from outside the package. But msg method of this package is declared as protected, so

it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. package pack;
3. public class A{

4. protected void msg(){System.out.println("Hello");}

5. }

1. //save by B.java

2. package mypack;

3. import pack.*;

4.

5. class B extends A{

6. public static void main(String args[]){

7. B obj = new B();

8. obj.msg();

9. }

10. }

Output:Hello

Public

Public members can be accessed from everywhere within your application either through inheritance or through object reference.

Note that the class or packages should also be accessible to access its members. You will not be able to access a public member from a default class from another package.

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

Example of public access modifier

1. //save by A.java

2.

3. package pack;

```
4. public class A{
5. public void msg(){System.out.println("Hello");}
6. }

1. //save by B.java
2.
3. package mypack;
4. import pack.*;
5.
6. class B{
7. public static void main(String args[]){
8. A obj = new A();
9. obj.msg();
10. }
11. }
```

Output:Hello

UNIT-IV

USING JAVA OBJECT

String,StringBufferandStringBuilder

1. String:

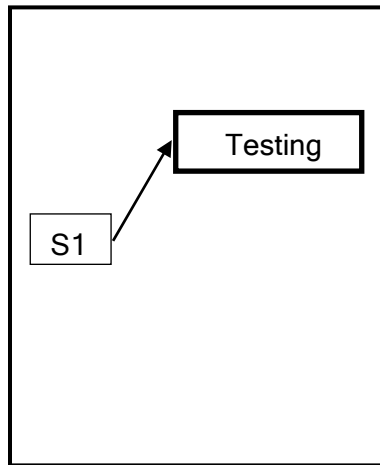
- String is a non-primitive data type in Java
- String is a class in Java
- String class objects are **immutable**.
- Once they are initialized, we cannot change them.
- We can create String objects using two methods.
- **Method#1:** String s = new String("Testing Shastra");// Created in Heap as well as SCP
- **Method#2:** String s = "Testing Shastra";// Created in String Constant Pool

What is meant by immutable?

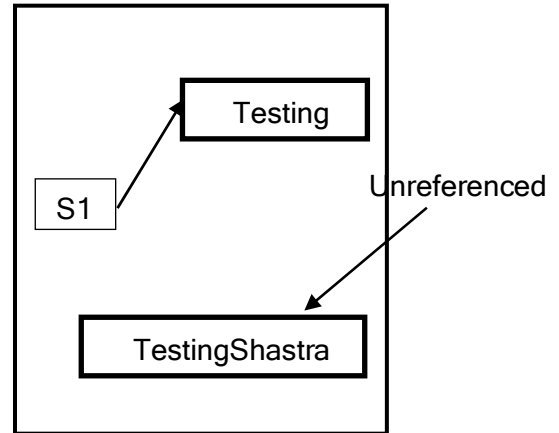
- Immutable objects cannot be deleted or changed once they are created. If we try to change them, a new object with changed value will be generated in memory. This feature is called immutability.

```
public class StringTest{  
  
    Public static void main(String[]  
    args){String s1="Testing";  
  
    s1.concat("Shastra");System  
    em.out.println(s1);  
  
    }  
}
```

- In above example, String "Shastra" will not be concatenated to "Testing" directly. Instead it will allocate new memory location to the concatenated String as shown in diagram.



BeforeConcatenation



(b)AfterConcatenation

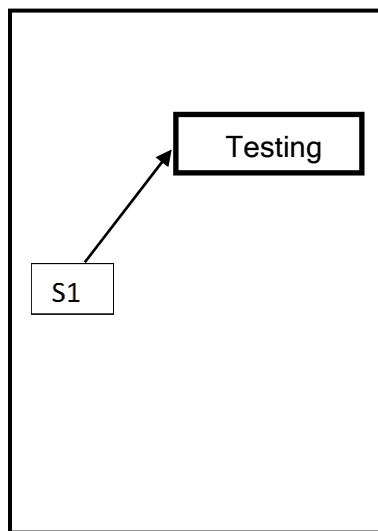
- We can observe that newly created string object (Testing Shastra) is not printed as output because we have not referenced to the new value yet. It will remain unreferenced till we provide reference to it.
- To get the concatenated string we have to make reference to it. Observe below program.

```

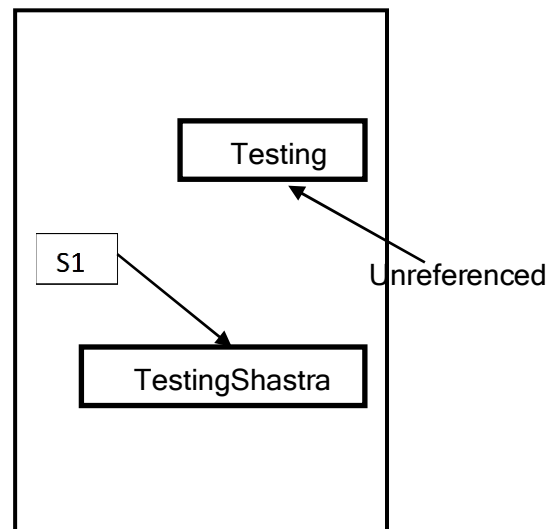
public class StringTest{
    Public static void main(String[]
    args){String s1="Testing";
    s1=s1.concat("Shastra");
    System.out.println(s1);
    }
  
```

Reference →

- Now we have created a reference to newly created string. Old string value will have no reference. But it is still present in memory. This is called **Immutability**.



(a) Before Concatenation



(b) After Concatenation

- All string objects are stored in either **String Constant Pool** or in Heap memory.
- String constant pool is a special type of memory dedicated to store String objects only.
- String Constant pool objects are NOT eligible for garbage collection. They will remain in memory until the life of program.
- Object creation in SCP is optional. If value is already present in SCP then new value will not

be created. Instead reference of old value will be given to new object.

Eg.

```
public class StringTest{  
    public static void main(String[]  
        args){String s1="Testing";  
        String s2="Testing";  
        if(s1==s2){ //Here == operator checks reference  
            System.out.println("s1 and s2 have same address");  
        }  
    }  
}
```

- Hence in above example s1 and s2 have same address. SCP doesn't allow duplicate values.
- Whenever a request for new object creation is generated, JVM check whether same value is present in SCP or not. If same value is present then address of that value will be assigned to newly created object, else new object will have new value.

Important Methods of String Class:

1. **public char charAt(int index):**

This method will return character at specified index.

E.g. String s="Testing
Shastra"; S.o.pl(s.charAt(5))// 'n' will be printed.

2. **public String concat(String s):**

This method will concat two strings.

E.g. String
s="Testing"; s=s.concat("Shast
ra");

3. **public int length():**

This method will return number of characters present in String. Eg. String s="TestingShastra";

s.o.p(s.length()) // This will print 15. Space will be considered as character

4. **public Boolean equals(Strings):**

This method will return true if content of two strings are exact match including cases.

E.g. `Strings="Testing";`

```
s.o.p(s.equals("Testing"))           //This  
will print true.s.o.p(s.equals("testing")) //This will  
print false
```

5. **Public Boolean equalsIgnoreCase(Strings):**

This method will compare two strings ignoring case.

E.g. `Strings="Testing";`

```
s.o.p(s.equalsIgnoreCase("Testing")) //This will print true.s.o.p(s.  
equalsIgnoreCase("testing")) //This will print true
```

6. **public String trim():**

This method trims starting and ending spaces if string has any.

E.g. `Strings="Testing Shastra ";`

```
s.o.p(s.trim());           //This will print "Testing Shastra". It will not trim intermediate spaces.
```

Disadvantage of String Class:

- String class is immutable hence it will create new value in memory after each and every operation.
- This will lead to memory wastage at some extent.
- To overcome this advantage Java have introduced `StringBuffer` and `StringBuilder` classes.

2. **StringBuffer:**

- If you are modifying your string frequently then it is not recommended to use `String` objects as they are immutable.
- When we have frequent operation as modification of string then we should compulsorily go for `StringBuffer`.
- Objects of `StringBuffer` class are mutable. They can be changed once they are initialized.
- Modifications are performed in existing object only. No new object is created.

Constructor of StringBuffer:

1. **StringBuffer sb=new StringBuffer();**

- This will create empty `StringBuffer` object with default initial capacity as 16.

- Once `StringBuffer` object reaches its initial capacity, a new `StringBuffer` object with new capacity is created using formula:

$\text{NewCapacity} = (\text{CurrentCapacity} + 1) * 2$

2. `StringBuffer sb = new StringBuffer(int initialCapacity);`

- This will create an empty `StringBuffer` object with specified initial capacity.

3. `StringBuffer sb = new StringBuffer(String s);`

- This will create a `StringBuffer` object for the string with capacity = $16 + s.length()$;

Important methods of `StringBuffer`:

- `public int length()` // It gives number of characters in `StringBuffer`
- `public int capacity()` // It gives current capacity of `StringBuffer`
- `public void setCharAt(int index, char c);` // It replaces character at specified index with specified character.
- `public StringBuffer append()`
// This method is overloaded and it appends two strings or values.

E.g `public StringBuffer append(String s);`
`public StringBuffer append(int i);`
`public StringBuffer append(doubled);`
`public StringBuffer append(Object o);` etc.

3. `StringBuilder`:

- `StringBuilder` and `StringBuffer` are same except, all methods of `StringBuffer` are synchronized.
- That means only one thread can operate on `StringBuffer` object at a time. This will lead to slow performance.
- If you open the `StringBuffer` class by pressing `Ctrl+click`, you can see that 'synchronized' keyword is attached in the declaration of each and every method, like:
`public synchronized void trimToSize()`
- For example if a single thread takes 1 sec to operate on `StringBuffer` object and there are 10 threads in system which are requesting access to same `StringBuffer` object, then total execution time of program will be raised by 10 seconds. Because each and every object has to wait for completion of previously operating thread.
- To overcome this issue, Java has introduced `StringBuilder`.
- None of the methods of `StringBuilder` are synchronized.

Difference between String, StringBuffer and StringBuilder:

Sr.No	String	StringBuffer	StringBuilder
1	String class is immutable in Java	StringBuffer is mutable	StringBuilder is mutable
2	None of the methods of String class is synchronized	Each method of StringBuffer is synchronized	None of the methods of StringBuilder class is synchronized
3	String is not thread safe	StringBuffer is thread safe	StringBuilder is not thread safe
4	It may lead to memory wastage	It may not lead to memory wastage	It will not lead to memory wastage
5	Not recommended where frequent operations are modification	Recommended where frequent operations are modifications	Recommended where frequent operations are modifications

THE THIS KEYWORD

this is a keyword in Java which is used as a reference to the object of the current class, within an instance method or a constructor. Using this you can refer the members of a class such as constructors, variables and methods.

Note – The keyword this is used only within instance methods or constructors

This

In general, the keyword this is used to –

Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student {  
    int age;  
  
    Student(int age) {  
        this.age = age;  
    }  
}
```

Call one type of constructor (parametrized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
class Student {  
    int age
```

```
Student() {  
    this(20);  
}
```

```
Student(int age) {  
    this.age = age;  
}  
}
```

PARAMETER PASSING TECHNIQUES IN JAVA WITH EXAMPLES

There are different ways in which parameter data can be passed into and out of methods and functions. Let us assume that a function B() is called from another function A(). In this case A is called the “caller function” and B is called the “called function or callee function”. Also, the arguments which A sends to B are called actual arguments and the parameters of B are called formal arguments.

TYPES OF PARAMETERS:

Formal Parameter : A variable and its type as they appear in the prototype of the function or method.

Syntax:

```
function_name(datatype variable_name)
```

Actual Parameter : The variable or expression corresponding to a formal parameter that appears in the function or method call in the calling environment.

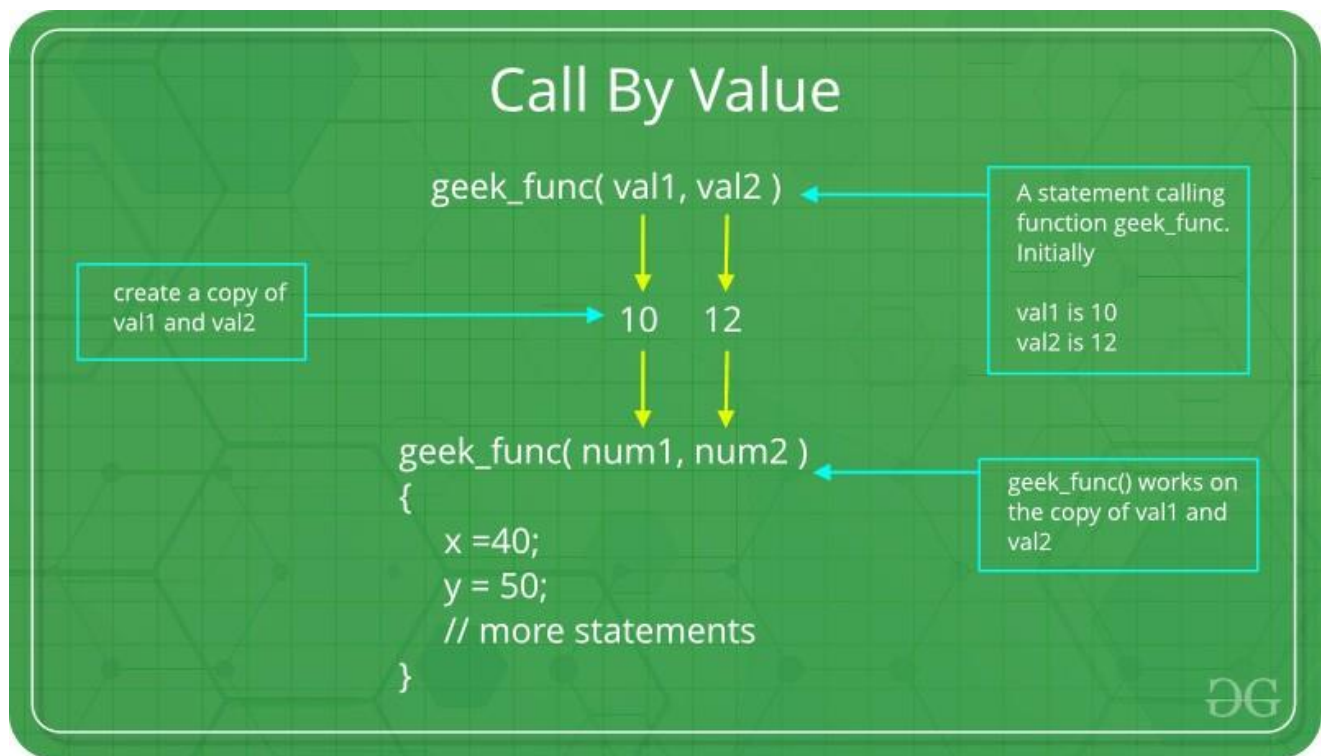
Syntax:

```
func_name(variable name(s));
```

. PASS BY VALUE:

Changes made to formal parameter do not get transmitted back to the caller. Any modifications to the formal parameter variable inside the called function or method affect only the separate storage location and will not be reflected in the actual parameter in the calling environment. This method is also called as call by value.

Java in fact is strictly call by value.



```
class CallByValue {  
  
    // Function to change the value  
    // of the parameters  
    public static void Example(int x, int y)  
    {
```

```
        x++;  
        y++;  
    }  
}
```

// Caller

```
public class Main {  
    public static void main(String[] args)  
    {  
  
        int a = 10;  
        int b = 20;  
  
        // Instance of class is created  
        CallByValue object = new CallByValue();  
  
        System.out.println("Value of a: " + a  
                            + " & b: " + b);  
  
        // Passing variables in the class function  
        object.Example(a, b);  
  
        // Displaying values after  
        // calling the function  
        System.out.println("Value of a: "
```

```

        + a + " & b: " + b);
    }
}

```

Output:

Value of a: 10 & b: 20

Value of a: 10 & b: 20

2. **CALL BY REFERENCE:** Changes made to formal parameter do get transmitted back to the caller through parameter passing. Any changes to the formal parameter are reflected in the actual parameter in the calling environment as formal parameter receives a reference (or pointer) to the actual data. This method is also called as call by reference. This method is efficient in both time and space.

```

class CallByReference {

    int a, b;

    // Function to assign the value
    // to the class variables
    CallByReference(int x, int y)
    {
        a = x;
        b = y;
    }

    // Changing the values of class variables
    void ChangeValue(CallByReference obj)
    {

```

```
    obj.a += 10;
    obj.b += 20;
}
}
```

// Caller

```
public class Main {
```

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

```
{
```

```
    // Instance of class is created
```

```
    // and value is assigned using constructor
```

```
    CallByReference object
```

```
        = new CallByReference(10, 20);
```

```
    System.out.println("Value of a: "
```

```
        + object.a
```

```
        + " & b: "
```

```
        + object.b);
```

```
    // Changing values in class function
```

```
    object.ChangeValue(object);
```

```
    // Displaying values
```

```
// after calling the function
```

```
System.out.println("Value of a: "
```

```
    + object.a
```

```
    + " & b: "
```

```
    + object.b);
```

```
}
```

```
}
```

Output:

Value of a: 10 & b: 20

Value of a: 20 & b: 40

UNIT-V

INHERITANCE IN JAVA

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Super class is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

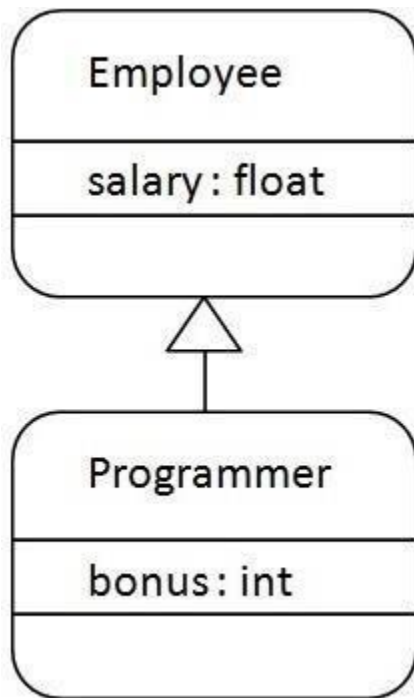
The syntax of Java Inheritance

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends** keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
1. class Employee{
2.     float salary=40000;
3. }
4. class Programmer extends Employee{
5.     int bonus=10000;
6.     public static void main(String args[]){
7.         Programmer p=new Programmer();
8.         System.out.println("Programmer salary is:"+p.salary);
9.         System.out.println("Bonus of Programmer is:"+p.bonus);
10.    }
11.    }
```

Test it Now

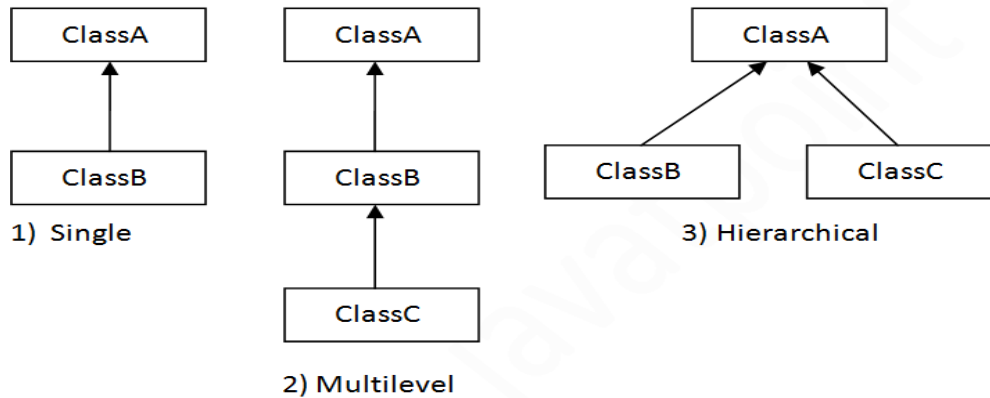
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

Types of inheritance in java

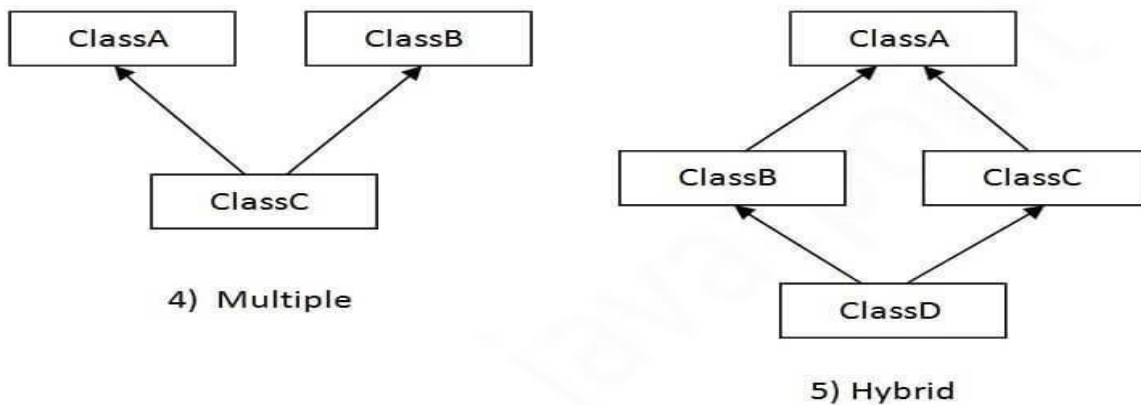
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

File: TestInheritance.java

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}

```
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class TestInheritance{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Output:

barking...

eating...

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
8. void weep(){System.out.println("weeping...");}
9. }
```

```
10.  class TestInheritance2{
11.  public static void main(String args[]){
12.  BabyDog d=new BabyDog();
13.  d.weep();
14.  d.bark();
15.  d.eat();
16.  }}
```

Output:

weeping...

barking...

eating...

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void bark(){System.out.println("barking...");}
6.  }
7.  class Cat extends Animal{
8.  void meow(){System.out.println("meowing...");}
9.  }
10. class TestInheritance3{
```

```
11.  public static void main(String args[]){
12.    Cat c=new Cat();
13.    c.meow();
14.    c.eat();
15.    //c.bark();//C.T.Error
16.  }
```

Output:

meowing...

eating...

Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1.  class A{
2.    void msg(){System.out.println("Hello");}
3.  }
4.  class B{
5.    void  msg(){System.out.println("Welcome");}
```

```

6. }
7. class C extends A,B{//suppose if it were
8.
9. public static void main(String args[]){
10.     C obj=new C();
11.     obj.msg();//Now which msg() method would be invoked?
12. }
13. }

```

Test it Now

Compile Time Error

The final Keyword

If you don't want other classes to inherit from a class, use the **final** keyword:

If you try to access a **final** class, Java will generate an error:

```

final class Vehicle{
...
}

class Car extends Vehicle{
...
}

```

The output will be something like this:

```

Main.java:9: error: cannot inherit from final Vehicle
class Main extends Vehicle {
      ^
1 error)

```


UNIT-VI

POLYMORPHISM IN JAVA

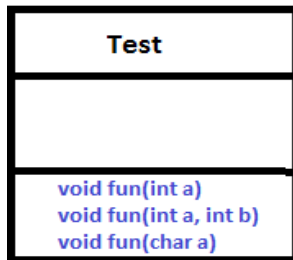
The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

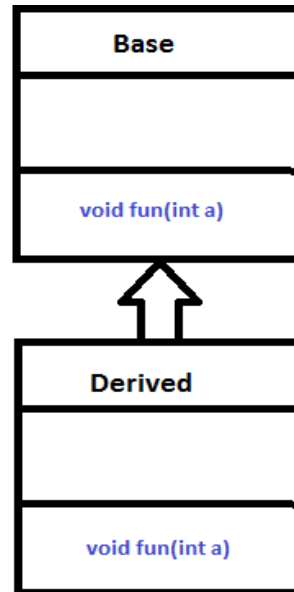
Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
 - Runtime Polymorphism
1. **Compile-time polymorphism:** It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java doesn't support the Operator Overloading.**



Overloading



Overriding

Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Example: By using different types of arguments

```
// Java program for Method overloading
class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 2 double parameter
    static double Multiply(double a, double b)
    {
        return a * b;
    }

}

class Main {
    public static void main(String[] args)
    {
        System.out.println(MultiplyFun.Multiply(2, 4));
    }
}
```

```
        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
    }
}
```

Output:

8

34.65

Example 2: By using different numbers of arguments

```
class MultiplyFun {
    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c)
    {
        return a * b * c;
    }
}

class Main {
    public static void main(String[] args)
    {
        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
```

Output:

8

42

2. Runtime polymorphism: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example:

// Java program for Method overriding

```
class Parent {

    void Print()
    {
```

```
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {

    void Print()
    {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {

    void Print()
    {
        System.out.println("subclass2");
    }
}

class TestPolymorphism3 {
    public static void main(String[] args)
    {
        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output:
subclass1
subclass2

UNIT-VII

PACKAGES IN JAVA

Package in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff..Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that, we can simply write an import class from existing packages and use it in our program. A package is a container of a group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.

We can reuse existing classes from the packages as many time as we need it in our program.

Attention reader! Don't stop learning now. Get hold of all the important **Java Foundation** and Collections concepts with the **Fundamentals of Java and Java Collections Course** at a student-friendly price and become industry ready. To complete your preparation from learning a language to DS Algo and many more, please refer **Complete Interview Preparation Course**.

How packages work?

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.

Package naming conventions : Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.

Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to be imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Example :

```
import java.util.*;
```

util is a subpackage created inside **java** package.

Accessing classes inside a package

Consider following two statements :

```
// import the Vector class from util package.
```

```
import java.util.Vector;
```

```
// import all the classes from util package
```

```
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.

- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package
```

```
// will be accessible but not subpackages.
```

```
import package.*;
```

```
// Only mentioned class of this package will be accessible.
```

```
import package.classname;
```

```
// Class name is generally used when two packages have the same
```

```
// class name. For example in below code both packages have
```

```
// date class so using a fully qualified name to avoid conflict
```

```
import java.util.Date;
```

```
import my.packag.Date;
```

```
// Java program to demonstrate accessing of members when
```

```
// corresponding classes are imported and not imported.
```

```
import java.util.Vector;
```

```
public class ImportDemo
```

```
{
```

```
    public ImportDemo()
```

```
    {
```

```
        // java.util.Vector is imported, hence we are
```

```
        // able to access directly in our code.
```

```
        Vector newVector = new Vector();
```

```
        // java.util.ArrayList is not imported, hence
```

```
        // we were referring to it using the complete
```

```
        // package.
```

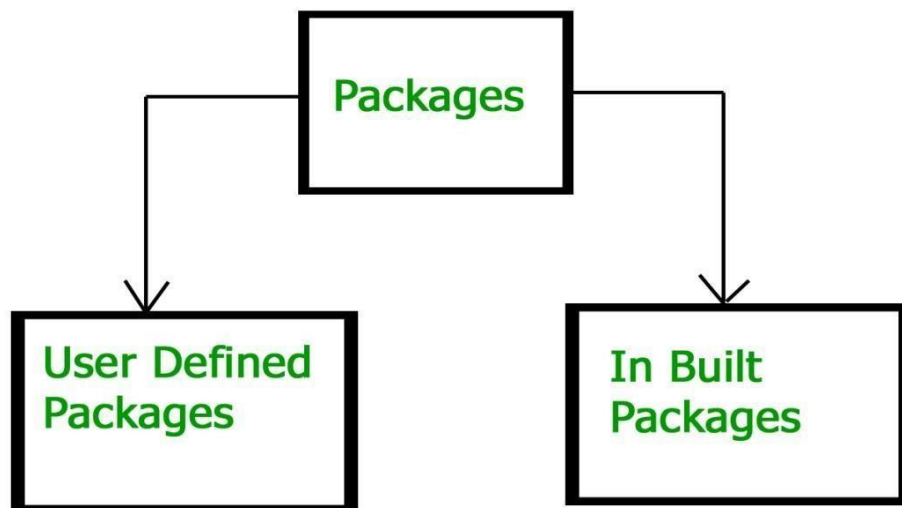
```
        java.util.ArrayList newList = new java.util.ArrayList();
```

```
    }
```

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

```
{  
  
    newImportDemo();  
  
}  
  
}
```

Types of packages:



Built-in Packages

These packages consist of a large number of classes which are a part of Java **API**. Some of the commonly used built-in packages are:

- 1) **java.lang**: Contains language support classes (e.g. classes which define primitive data types, math operations). This package is automatically imported.
- 2) **java.io**: Contains classes for supporting input / output operations.
- 3) **java.util**: Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
- 4) **java.applet**: Contains classes for creating Applets.
- 5) **java.awt**: Contains classes for implementing the components for graphical user interfaces (like button , ; menus etc).
- 6) **java.net**: Contains classes for supporting networking operations.

User-defined packages

These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

```
// Name of the package must be same as the directory
```

```
// under which this file is saved
```

```
package myPackage;
```

```
public class MyClass
```

```
{
```

```
    public void getNames(String s)
```

```
    {
```

```
        System.out.println(s);
```

```
    }
```

```
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
```

```
import myPackage.MyClass;
```

```
public class PrintName
```

```
{
```

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

```
    {
```

```
        // Initializing the String variable
```

```
        // with a value
```

```
        String name = "GeeksforGeeks";
```

```
        // Creating an instance of class MyClass in
```

```
        // the package.
```

```
        MyClass obj = new MyClass();
```

```
        obj.getNames(name);
    }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

Using Static Import

Static import is a feature introduced in **Java** programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.

Following program demonstrates **static import** :

```
// Note static keyword after import.
```

```
import static java.lang.System.*;
```

```
class StaticImportDemo
```

```
{
```

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

```
    {
```

```
        // We don't need to use 'System.out'
```

```
        // as imported using static.
```

```
        out.println("GeeksforGeeks");
```

```
    }
```

```
}
```

Output:

GeeksforGeeks

Handling name conflicts

The only time we need to pay attention to packages is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;
```

```
import java.sql.*;
```

//And then use Date class, then we will get a compile-time error :

```
Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using a specific import statement:

```
import java.util.Date;
```

```
import java.sql.*;
```

If we need both Date classes then, we need to use a full package name every time we declare a new object of that class.

For Example:

```
java.util.Date deadLine = new java.util.Date();
```

```
java.sql.Date today = new java.sql.Date();
```

Directory structure

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, the class Circle of package com.zzz.project1.subproject2 is stored as

“\$BASE_DIR\com\zzz\project1\subproject2\Circle.class”, where \$BASE_DIR denotes the base directory of the package. Clearly, the “dot” in the package name corresponds to a sub-directory of the file system.

The base directory (\$BASE_DIR) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the \$BASE_DIR so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

Setting CLASSPATH:

CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows, choose control panel ? System ? Advanced ? Environment Variables ? choose “System Variables” (for all the users) or “User Variables” (only the currently login user) ? choose “Edit” (if CLASSPATH already exists) or “New” ? Enter “CLASSPATH” as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., “.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar”). Take note that you need to include the current working directory (denoted by ‘.’) in the CLASSPATH.

To check the current setting of the CLASSPATH, issue the following command:

- > SET CLASSPATH
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
- > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,
- > java -classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3

Illustration of user-defined packages:

Creating our first package:

File name - ClassOne.java

```
packagepackage_name;
```

```
publicclassClassOne {
```

```
    publicvoidmethodClassOne() {
```

```
        System.out.println("Hello there its ClassOne");
```

```
}  
  
}
```

Creating our second package:
File name - ClassTwo.java

```
packagepackage_one;  
  
publicclassClassTwo {  
  
    publicvoidmethodClassTwo(){  
  
        System.out.println("Hello there i am ClassTwo");  
  
    }  
  
}
```

Making use of both the created packages:
File name - Testing.java

```
importpackage_one.ClassTwo;  
  
importpackage_name.ClassOne;  
  
publicclassTesting {  
  
    publicstaticvoidmain(String[] args){  
  
        ClassTwo a = newClassTwo();
```

```

        ClassOne b = new ClassOne();

        a.methodClassTwo();

        b.methodClassOne();

    }

}

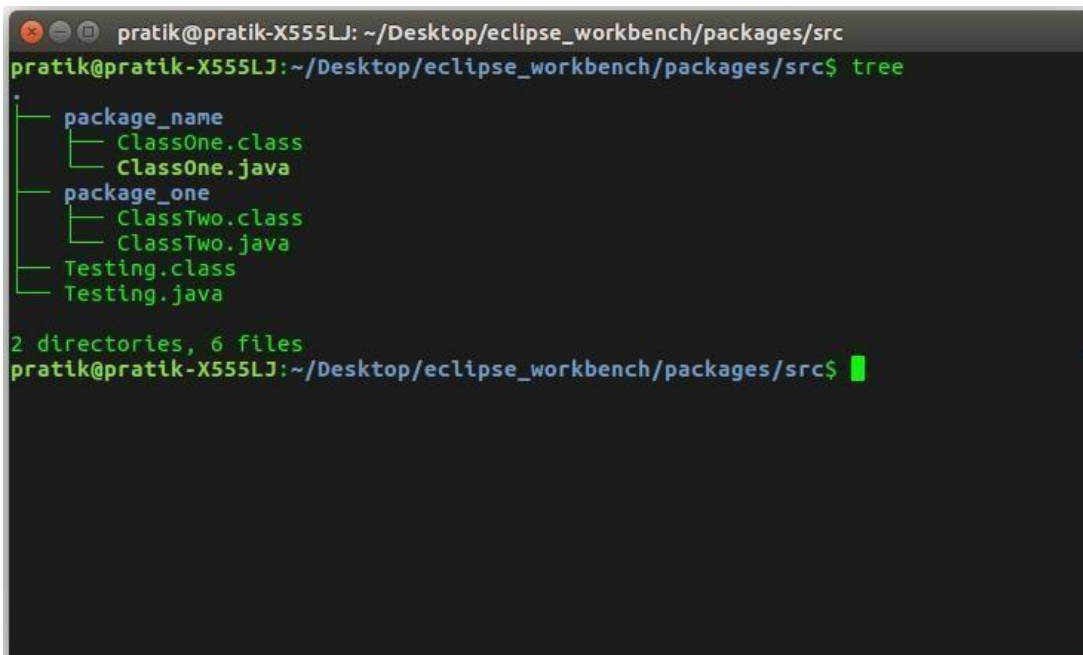
```

Output:

Hello there i am ClassTwo

Hello there its ClassOne

Now having a look at the directory structure of both the packages and the testing class file:



```

pratik@pratik-X555LJ: ~/Desktop/eclipse_workbench/packages/src
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$ tree
.
├── package_name
│   ├── ClassOne.class
│   └── ClassOne.java
├── package_one
│   ├── ClassTwo.class
│   └── ClassTwo.java
├── Testing.class
└── Testing.java

2 directories, 6 files
pratik@pratik-X555LJ:~/Desktop/eclipse_workbench/packages/src$

```

Important points:

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).

5. We can access public classes in another (named) package using: **package-name.class-name**

UNIT-VIII

JAVA FILE AND I/O

Java I/O (Input and Output) is used *to process the input and produce the output*.

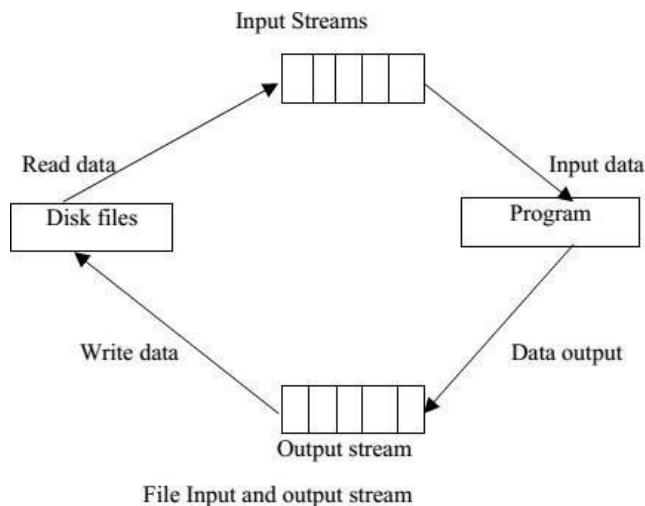
Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

STREAM

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

The stream is the only representation of input or output that is maybe a source or destination of the data. We can also write the data in the stream or read the particular data from the stream. We can also visualize the stream of data as a sequence of bytes that flow out of the program.



In Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) **System.out**: standard output stream

2) **System.in**: standard input stream

3) **System.err**: standard error stream

Let's see the code to print **output and an error** message to the console.

1. `System.out.println("simple message");`
2. `System.err.println("error message");`

Let's see the code to get **input** from console.

1. `int i=System.in.read();//returns ASCII code of 1st character`
2. `System.out.println((char)i);//will print the character`

TYPES OF STREAMS

Depending upon the data a stream holds, it can be classified into:

- Byte Stream
- Character Stream

BYTE STREAM

Byte stream is used to read and write a single byte (8 bits) of data.

All byte stream classes are derived from base abstract classes called `InputStream` and `OutputStream`.

To learn more, visit

- Java `InputStream` Class
- Java `OutputStream` Class

CHARACTER STREAM

Character stream is used to read and write a single character of data.

All the character stream classes are derived from base abstract classes `Reader` and `Writer`.

To learn more, visit

- Java Reader Class
- Java Writer Class

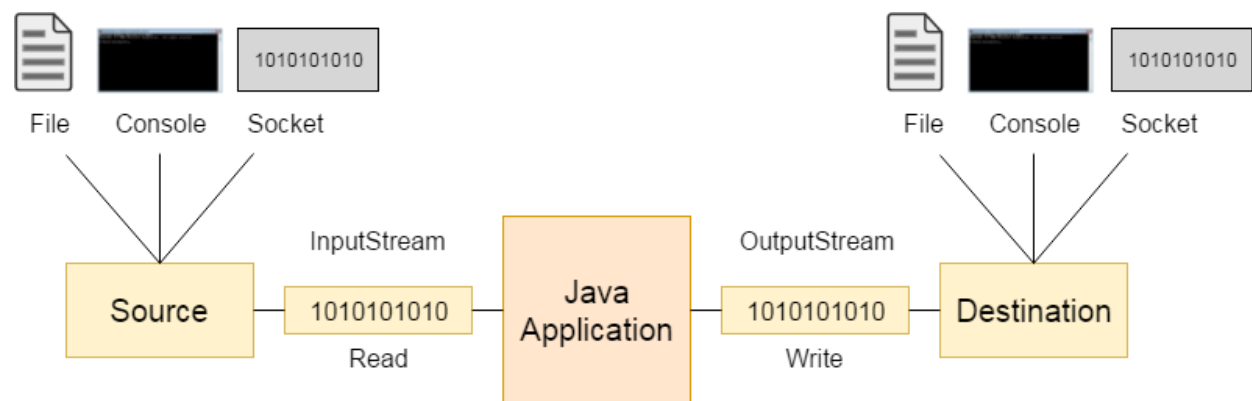
OUTPUT STREAM

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

INPUT STREAM

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.



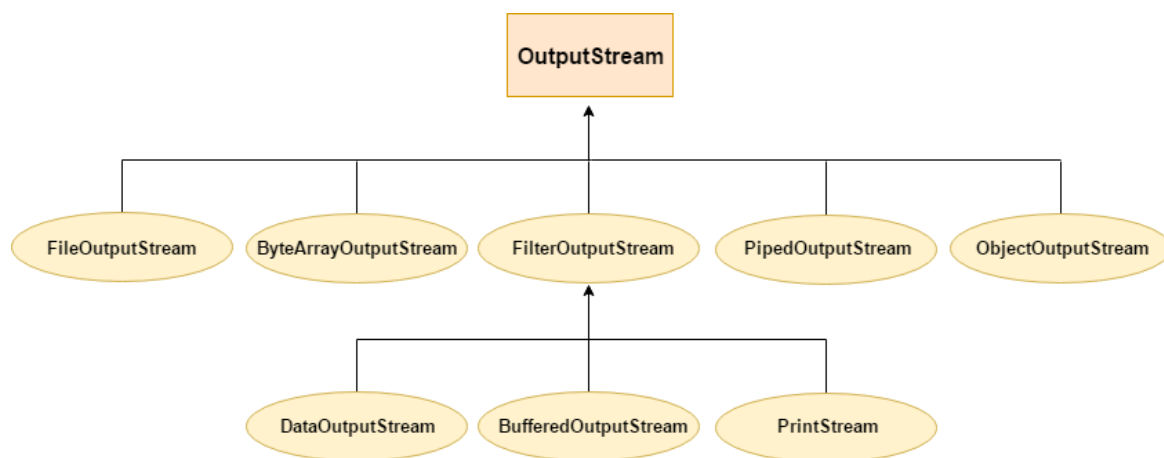
OUTPUTSTREAM CLASS

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

USEFUL METHODS OF OUTPUTSTREAM

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

OutputStream Hierarchy



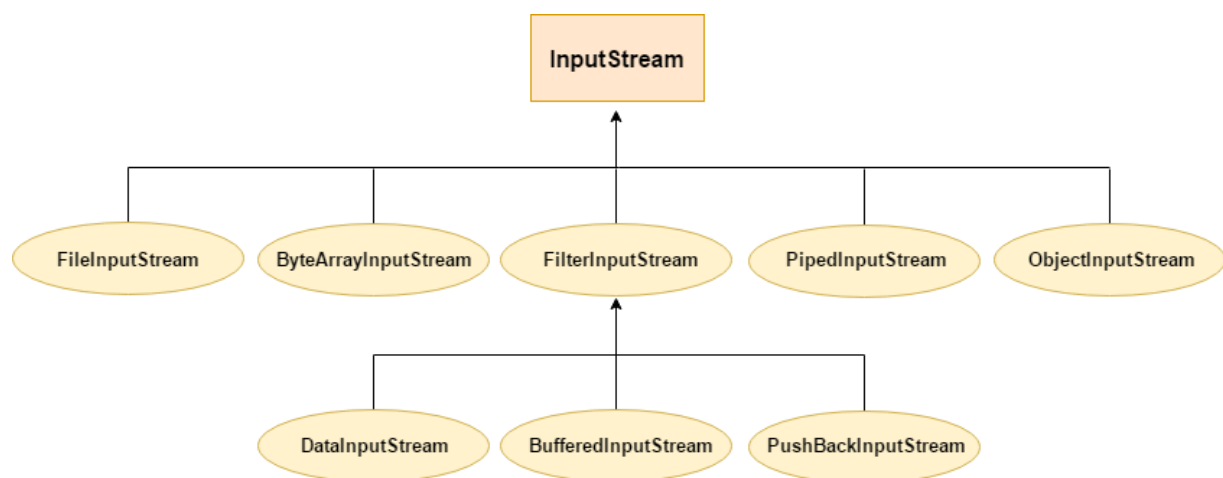
INPUTSTREAM CLASS

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

USEFUL METHODS OF INPUTSTREAM

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

INPUTSTREAM HIERARCHY



JAVA FILEOUTPUTSTREAM CLASS

Java FileOutputStream is an output stream used for writing data to a file

.

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter

than FileOutputStream.

FILEOUTPUTSTREAM CLASS DECLARATION

Let's see the declaration for Java.io.FileOutputStream class:

1. **public class** FileOutputStream **extends** OutputStream

FileOutputStream class methods

Method	Description
protected void finalize()	It is used to clean up the connection with the file output stream.
void write(byte[] ary)	It is used to write ary.length bytes from the byte array to the file output stream.
void write(byte[] ary, int off, int len)	It is used to write len bytes from the byte array starting at offset off to the file output stream.
void write(int b)	It is used to write the specified byte to the file output stream.
FileChannel getChannel()	It is used to return the file channel object associated with the file output stream.
FileDescriptor getFD()	It is used to return the file descriptor associated with the

	stream.
void close()	It is used to closes the file output stream.

JAVA FILEOUTPUTSTREAM EXAMPLE 1: WRITE BYTE

```

1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.         try{
5.             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.             fout.write(65);
7.             fout.close();
8.             System.out.println("success...");
9.         }catch(Exception e){System.out.println(e);}
10.    }
11.}

```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **A**.

testout.txt

```
A
```

JAVA FILEOUTPUTSTREAM EXAMPLE 2: WRITE STRING

```

1. import java.io.FileOutputStream;
2. public class FileOutputStreamExample {
3.     public static void main(String args[]){
4.         try{

```

```
5.      FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
6.      String s="Welcome to javaTpoint.";
7.      byte b[]=s.getBytes();//converting string into byte array
8.      fout.write(b);
9.      fout.close();
10.     System.out.println("success...");
11.     }catch(Exception e){System.out.println(e);}
12. }
13. }
```

Output:

```
Success...
```

The content of a text file **testout.txt** is set with the data **Welcome to javaTpoint.**

testout.txt

```
Welcome to javaTpoint.
```

JAVA FILEINPUTSTREAM CLASS

Java FileInputStream class obtains input bytes from a file

. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader

class.

JAVA FILEINPUTSTREAM CLASS DECLARATION

Let's see the declaration for java.io.FileInputStream class:

```
1. public class FileInputStream extends InputStream
```

JAVA FILEINPUTSTREAM CLASS METHODS

Method	Description
int available()	It is used to return the estimated number of bytes that can be read from the input stream.
int read()	It is used to read the byte of data from the input stream.
int read(byte[] b)	It is used to read up to b.length bytes of data from the input stream.
int read(byte[] b, int off, int len)	It is used to read up to len bytes of data from the input stream.
long skip(long x)	It is used to skip over and discards x bytes of data from the input stream.
FileChannel getChannel()	It is used to return the unique FileChannel object associated with the file input stream.
FileDescriptor getFD()	It is used to return the FileDescriptor object.
protected void finalize()	It is used to ensure that the close method is call when there is no more reference to the file input stream.
void close()	It is used to closes the stream

JAVA FILEINPUTSTREAM EXAMPLE 1: READ SINGLE CHARACTER

```
import java.io.FileInputStream;
```

1. **public class** DataStreamExample {
2. **public static void** main(String args[]){
3. **try**{


```

4.      FileInputStream fin=new FileInputStream("D:\\testout.txt");
5.      int i=fin.read();
6.      System.out.print((char)i);
7.
8.      fin.close();
9.  }catch(Exception e){System.out.println(e);}
10. }
11. }

```

Note: Before running the code, a text file named as "**testout.txt**" is required to be created. In this file, we are having following content:

```
Welcome to javatpoint.
```

After executing the above program, you will get a single character from the file which is 87 (in byte form). To see the text, you need to convert it into character.

Output:

```
W
```

JAVA FILEINPUTSTREAM

EXAMPLE 2: READ ALL CHARACTERS

```

1. package com.javatpoint;
2.
3. import java.io.FileInputStream;
4. public class DataStreamExample {
5.     public static void main(String args[]){
6.         try{
7.             FileInputStream fin=new FileInputStream("D:\\testout.txt");
8.             int i=0;
9.             while((i=fin.read())!=-1){

```

```
10.      System.out.print((char)i);
11.      }
12.      fin.close();
13.      }catch(Exception e){System.out.println(e);}
14.      }
15.      }
```

Output:

```
Welcome to javaTpoint
```

UNIT-IX

EXCEPTION HANDLING IN JAVA

WHAT IS AN EXCEPTION?

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

ERROR VS EXCEPTION

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Basically, an **Error** is used by the Java run-time system (JVM) to indicate errors that are associated with the run-time environment (JRE).

StackOverflowError is an example of such an error.

There are two types of errors:

1. Compile time errors
2. Runtime errors

Compile time errors can be again classified again into two types:

- Syntax Errors
- Semantic Errors

Syntax Errors Example:

Instead of declaring `int a;` you mistakenly declared it as `in a;` for which compiler will throw an error.

Example: You have declared a variable `int a;` and after some lines of code you again declare an integer as `int a;`. All these errors are highlighted when you compile the code.

Runtime Errors Example

A Runtime error is called an **Exceptions** error. It is any event that interrupts the normal flow of program execution.

Exception: Exception indicates conditions that a reasonable application might try to catch.

DIFFERENCE BETWEEN ERRORS AND EXCEPTIONS

Errors	Exceptions
1. Impossible to recover from an error	1. Possible to recover from exceptions
2. Errors are of type 'unchecked'	2. Exceptions can be either 'checked' or 'unchecked'
3. Occur at runtime	3. Can occur at compile time or run time
4. Caused by the application running environment	4. Caused by the application itself

An exception is a problem that arises during the execution of a program. It can occur for various reasons say-

- A user has entered an invalid data
- File not found
- A network connection has been lost in the middle of communications
- The JVM has run out of a memory

What happens if exceptions are not handled?

When an exception occurs, and if you don't handle it, the program will terminate abruptly (the piece of code after the line causing the exception will not get executed)/ system failure. That is why java introduced exception handling to handle the runtime errors so that the normal flow of execution will not be disrupted .

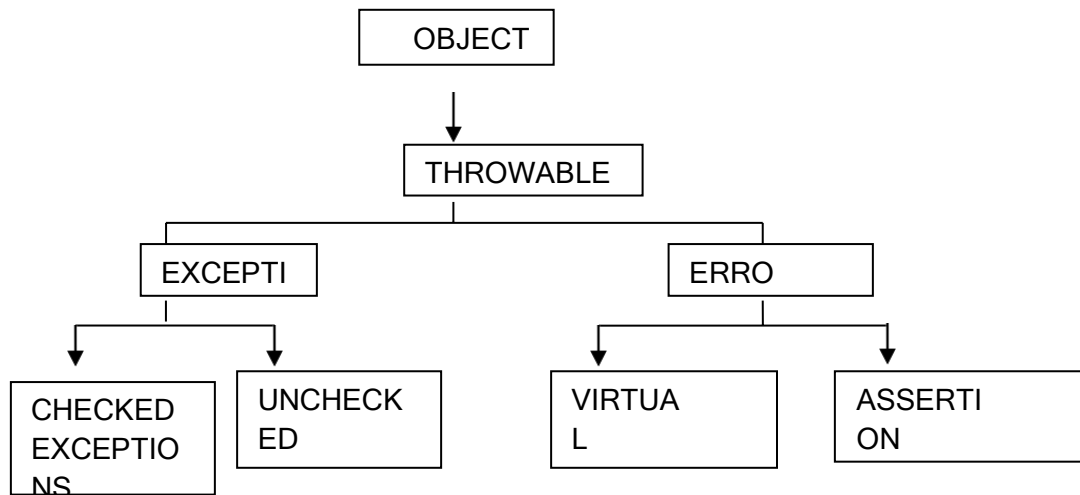
Ex. i/o exception

Class not found

EXCEPTIONS HIERARCHY

All exception and error types are subclasses of class Throwable, which is the base class of hierarchy. One branch is headed by Error which occurs at run-time and other by *Exception* that can happen either at compile time or run-time.

This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, Error are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



TYPES OF JAVA EXCEPTIONS

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception

It is an exception that occurs at compile time, also called compile time exceptions. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using throws keyword.

The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

Unchecked Exception

It is an exception that occurs at the time of execution. These are also called Runtime Exceptions. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to specify or catch the exceptions.

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

JAVA EXCEPTION KEYWORDS

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java try and catch

The `try` statement allows you to define a block of code to be tested for errors while it is being executed.

The `catch` statement allows you to define a block of code to be executed, if an error occurs in the try block.

The `try` and `catch` keywords come in pairs:

Syntax:

```
try{  
    // Block of code to try  
}  
  
catch(Exceptione){  
    // Block of code to handle errors  
}
```

Multiple catch block:

```
try {  
    // Protected code  
} catch (ExceptionType1 e1) {  
    // Catch block  
} catch (ExceptionType2 e2) {  
    // Catch block  
} catch (ExceptionType3 e3) {  
    // Catch block  
}
```

Example:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        int[] myNumbers = {1, 2, 3};
```

```
System.out.println(myNumbers[10]);  
  
}  
  
} //after running the program it will show you:
```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 10 out of bounds for length 3 at Main.main(Main.java:4)

Using try and catch to catch the error and execute some code to handle it:

```
public class Main {  
  
    public static void main(String[ ] args) {  
  
        try {  
  
            int[] myNumbers = {1, 2, 3};  
  
            System.out.println(myNumbers[10]);  
  
        } catch (Exception e) {  
  
            System.out.println("Something went wrong.");  
  
        }  
  
    }  
  
}
```

The output will be:

Something went wrong.

Ex:

```
class JavaException {  
    public static void main(String args[]) {  
        int d = 0;  
        int n = 20;  
        try {  
            int fraction = n / d;  
            System.out.println("This line will not be Executed");  
        } catch (ArithmeticException e) {  
            System.out.println("In the catch Block due to Exception = " + e);  
        }  
        System.out.println("End Of Main");  
    }  
}
```



```
}
```

Using try, catch and throw:

```
public class Main{  
    static void checkAge(int age){  
        if(age < 18){  
            throw new ArithmeticException("you are not eligible for voting");  
        }  
        else{  
            System.out.println("you are eligible for voting ");  
        }  
    }  
}
```

```
public static void main(String[] args){  
    checkAge(15); // Set age to 15 (which is below 18...)  
}  
}
```

Ex:

```
class Main{  
    public static void divideByZero(){  
  
        // throw an exception  
        throw new ArithmeticException("Trying to divide by 0");  
    }  
  
    public static void main(String[] args){  
        divideByZero();  
    }  
}
```

Output

```
Exception in thread "main" java.lang.ArithmeticException: Trying to divide by 0
    at Main.divideByZero(Main.java:5)
    at Main.main(Main.java:9)
```

Types of Exceptions

1. Built-in Exceptions

Built-in Exceptions	Description
ArithmeticException	It is thrown when an exceptional condition has occurred in an arithmetic operation.
ArrayIndexOutOfBoundsException	It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
ClassNotFoundException	This exception is raised when we try to access a class whose definition is not found.
FileNotFoundException	An exception that is raised when a file is not accessible or does not open.
IOException	It is thrown when an input-output operation is failed or interrupted.
InterruptedException	It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
NoSuchFieldException	It is thrown when a class does not contain the field (or variable) specified.

2. User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, a user can also create exceptions which are called 'User-Defined Exceptions'.

Key points to note:

- a. A user-defined exception must extend Exception class.
- b. The exception is thrown using *throw* keyword.