OBJECT ORIENTED METHODOLOGY

3rd SEM

COMPUTER SCIENCE & ENGINEERING

ASHIS BEHERA

SR. LECTURE , CSE

PADMASHREE KRUTARTHA ACHARYA INSTITUTE OF ENGINEERING & TECHNOLOGY, BARGARH, ODISHA

OBJECT ORIENTED METHODOLOGY (SYLLABUS)

Chapter-1

OOP CONCEPT

Evolution of Programming Language:

A programming language is a series of instructions, or algorithms, written within a specific language environment like Python or C, with a primary goal of performing a wide range of specific tasks.

Types of Programming Languages

Generally, all computer programming languages, can be divided into two main categories:

Low Level Languages

Low level languages are used for writing computer instructions in binary code – that is machine code made up of the numbers 0 and 1. Examples of low level languages include machine language and assembly language. Machine language is the first generation of computer programming – using instructions in binary form that can be directly interpreted by a CPU without a need for translation. Assembly language is the second generation of low level computer programming. This type of language allows programmers to write computer instructions through the use of symbolic code instead of binary code made up of just 0s and 1s.

High Level Languages

High level languages are programming languages that enable software developers to write computer instructions by using commands that are written in human languages like English. Every high level language has its own set of rules and grammar for writing instructions to program any digital device. These unique sets of rules are generally referred to as the 'syntax' of a particular programming language. In contrast to low level languages, before running a program written in a high level language, the coding instructions must first be translated into machine code. Every high level programming language uses its own built-in translation program.

1883: Ada

In the early days, Charles Babbage had made the device, but he was confused about how to give instructions to the machine, and then Ada Lovelace wrote the instructions for the analytical engine.

The device was made by Charles Babbage and the code was written by Ada Lovelace for computing Bernoulli's number.

First time in history that the capability of computer devices was judged.

1949: Assembly Language

- It is a type of low-level language.
- It mainly consists of instructions (kind of symbols) that only machines could understand.
- In today's time also assembly language is used in real-time programs such as simulation flight navigation systems and medical equipment eg – Fly-by-wire (FBW) systems.
- It is also used to create computer viruses.

1952: Autocode

- Developed by AlickGlennie.
- The first compiled computer programming language.
- COBOL and FORTRAN are the languages referred to as Autocode.

1957: FORTRAN

- Developers are John Backus and IBM.
- It was designed for numeric computation and scientific computing.
- Software for NASA probes voyager-1 (space probe) and voyager-2 (space probe) was originally written in FORTRAN 5.

1958: ALGOL

- ALGOL stands for ALGOrithmic Language.
- The initial phase of the most popular programming languages of C, C++, and JAVA.
- It was also the first language implementing the nested function and has a simple syntax than FORTRAN.
- The first programming language to have a code block like "begin" that indicates that your program has started and "end" means you have ended your code.

1959: COBOL

- It stands for COmmon Business-Oriented Language.
- In 1997, 80% of the world's business ran on Cobol.
- The US internal revenue service scrambled its path to COBOL-based IMF (individual master file) in order to pay the tens of millions of payments mandated by the coronavirus aid, relief, and economic security.

1964: BASIC

- It stands for beginners All-purpose symbolic instruction code.
- In 1991 Microsoft released Visual Basic, an updated version of Basic
- The first microcomputer version of Basic was co-written by Bill Gates, Paul Allen, and Monte Davidoff for their newly-formed company, Microsoft.

1972: C

- It is a general-purpose, procedural programming language and the most popular programming language till now.
- All the code that was previously written in assembly language gets replaced by the C language like operating system, kernel, and many other applications.
- It can be used in implementing an operating system, embedded system, and also on the website using the Common Gateway Interface (CGI).
- C is the mother of almost all higher-level programming languages like C#, D, Go, Java, JavaScript, Limbo, LPC, Perl, PHP, Python, and Unix's C shell.

1979: C++

- C++ was developed by BjarneStroustrup in 1979.
- C++ supports polymorphism, encapsulation, and inheritance because it is an object oriented programming language.
- C++ is (mostly) a superset of C.
- C++ is known as hybrid language because C++ supports both procedural and object oriented programming paradigms.
- Data and functions are encapsulated together in form of an object in C++.
- Built-in & user-defined data types is supported in C++.
- C++ focuses on data instead of focusing on method or procedure.
- C++ follows the Bottom-up approach.

1995: JAVA

Java is an object-oriented, class-based, concurrent, secured, general-purpose, platformindependent, high level, robust, object-oriented and secure programming language. The program written in java language can run in any platform or any environment. It is widely used in windows based, web-based, enterprise, and mobile applications. Since java uses both interpreter and compiler, the java source code is converted into bytecode at compilation time and the interpreter executes this bytecode at runtime and produces output. Due to this interpreted, java is also known as platform independent.

Java Bytecode:

Bytecode is essentially the machine level language which runs on the Java Virtual Machine. As soon as a java program is compiled, java bytecode is generated. Java bytecode is the machine code in the form of a .class file.Whenever a class is loaded, it gets a stream of bytecode per method of the class. Whenever that method is called during the execution of a program, the bytecode for that method gets invoked.Javac not only compiles the program but also generates the bytecode for the program. Thus, we have realized that the bytecode implementation makes Java a platform-independent language. This helps to add portability to Java which is lacking in languages like C or C++. Portability ensures that Java can be implemented on a wide array of platforms like desktops, mobile devices, severs and many more. Supporting this, Sun Microsystems captioned JAVA as *"write once, read anywhere" or "WORA"* in resonance to the bytecode interpretation.

Definition of JAVA:

Java is a Simple, Object-Oriented, Portable, Platform independent, Secured, Robust, Architecture neutral, Interpreted, High Performance, Multithreaded, Distributed, and Dynamic programming language.

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:Java syntax is based on C++. Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc. and There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.

Platform Independent

Java is platform independent because Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- No explicit pointer
- Java Programs run inside a virtual machine sandbox.
- Classloader: Classloader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- Bytecode Verifier: It checks the code fragments for illegal code that can violate access rights to objects.
- Security Manager: It determines what resources a class can access such as reading and writing to the local disk.

Robust

Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.

- Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

OOP Principles:

Java support following OOP principles.

Object: Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Class: Collection of objects is called class. It is a logical entity.A class can also be defined as a blueprint from which we can create an individual object. Class doesn't consume any space.

Encapsulation: Binding (or wrapping) code and data together into a single unit are known as encapsulation. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Inheritance: Inheritance is an important concept of OOP (Object Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features (fields and methods) of another class. We are achieving inheritance by using extends keyword. Inheritance is also known as “is-a” relationship.

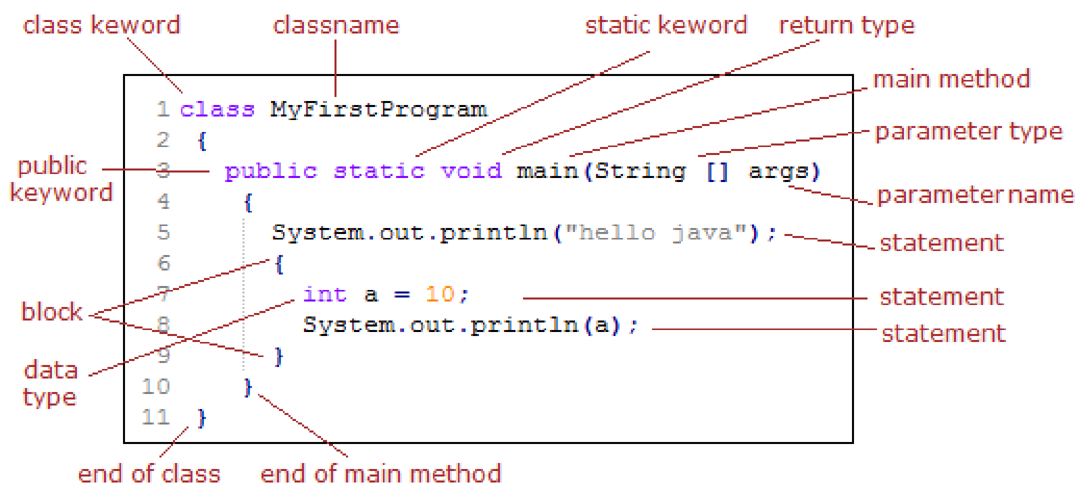
Polymorphism: It refers to the ability of object-oriented programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities. The ability to appear in many forms is called polymorphism.

Data Abstraction: It is the property by virtue of which only the essential details are displayed to the user. The trivial or non-essential units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.Data Abstraction may also be defined as the process of identifying only the required characteristics of an object, ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the object.

Chapter-2

ITRODUCTION TO JAVA

Brief description of each term of a simple JAVA Program:



```
class MyFirstProgram
 {
   public static void main(String [] args)
     {
       System.out.println("hello java");
       {
         int a = 10;
         System.out.println(a);
       }
     }
 }
```

class - The class is a keyword in java which is used to define a class. In java every program must have a class. A class contains set of methods and variables. After class keyword programmers need to write the name of the class which is used to refer that class within or outside the class. In above program **MyFirstProgram** is the name of class, everything that is inside balanced {} after the class name are the part of class.

Statement - A statement is similar to a sentence in english language. As sentences makes a complete idea, similarly java statement makes a complete unit of execution.

Method - A method is set of statements that performs specific task or call other methods. A method has a name and a return type. The name of the method is used to refer that method within or outside the class. A class can have multiple methods. In above program **main** is the method name. Everything that comes in balanced { } after method name are the part of method. Every java program must have a **main** method if it needs to be run independently. The **main** method is the starting point of execution of a program in java.

block - A block is a group of zero or more statements. it starts with curly braces { and ends with balanced }. All statements inside balanced { } is the part of block. A block is generally used to group several statements as a single unit. A block can have another block inside it. Blocks does not have a name, they are just logical grouping of statements inside { }.

public - The keyword public is an access modifier that decides the visibility or accessibility of a member. Variables or methods declared with public keyword can be accessed outside the class. Since main method is called by JVM at the time of program execution that is why it must be declared as public, otherwise JVM won't be able to find the main method in your program and your program will not execute.

static - The static is a keyword in java. A method or variable declared with static keyword can be called without creating the object of that class. Since JVM calls the main method without creating the object of class, that is why it must be declared as static, otherwise JVM won't be able to call the **main** method.

return - The return is also a keyword in java. It is used to return value from the method to the caller of the method. Every method must have a return type, if it's not returning any value then the return type of that method must be void. Since **main** method doesn't return any value, that's why it's return type is void.

Variable and data type - Variable in java is very similar to variable in mathematics, used to store value. Data type of a variable defines what type of data that variable can store. In above program **a** is a variable and int is it's data type, which means **a** can contain only integer type of value.

Parameter - A parameter is special kind of variable which receives the value from the caller of the method. A parameter can be used within the method in which it is declared. In above example args is a parameter of type String array. Any argument passed to program while running the program is stored in args parameter.

System.out.println() - It is used to print the output(any string or variable) of a program on console. Anything passed to println method will be printed on console. A console is basically a window or terminal where you can pass input to program or print the output of a program. Command prompt(cmd) in windows is an example of console.

Data types in java:

A data is simply an instruction, it could be like 123, -125, 3.14, "hello" etc. A **data type** is basically classification of these data. These data can be divided in different data types like integer number, float number, character etc. In java there are basically two types of data types :

- Primitive Data Type
- Non Primitive Data Type

Primitive Data Type

Primitive data type deals on basic data like 123, -125, 3.14 etc. Primitive data types are predefined by the java language itself. Considering size and type of data, java has defined eight types of primitive data type.

Type	**Size**	**Range**	**Default Value**	**Example**
byte	1 byte	-128 to 127	0	byte b = 123;
short	2 byte	-32768 to 32767	0	short s = 1234;
int	4 byte	-2^{31} to $2^{31}-1$	0	int i = 123456;
long	8 byte	-2^{63} to $2^{63}-1$	0L	long l = 312456L; long ll = 312456l;
float	4 byte	1.4E-45f to 3.4028235E38f	0.0f	float f = 123.45f; float ff = 123.45F;
double	8 byte	4.9E-324 to 1.7976931348623157E308	0.0d	double d = 1234.67d; double dd = 1234.67D;
char	2 byte	'\u0000' (or 0) to '\uffff' (or 65,535 inclusive)	'\u0000'	char c = 'C';
boolean	represents 1 bit of information	Not Applicable	false	boolean b = true;

Data types byte, short, int, long is also known as **Integer** data type because they can contain only integer type values while data types float, double is also known as **Floating** point data type because they are used to store float type values.

Primitive types are also called as **value types** in java because variables of primitive data types directly holds the values rather than the reference(address) of that value in memory. That is why when you access a variable of primitive type, you get the value directly rather than the reference of that value in memory.

Primitive data type program in Java

```
 class PrimitiveDataTypeExample {
   public static void main(String[] args) {
      byte byteVar = 123;
      short shortVar = 1234;
      int intVar = 123456;
      long longVar = 3124567891L; // Must end with l or L
      float floatVar = 123.45f; // Must end with f or F
```

```
        double doubleVar = 12345.6789d; // d or D is optional
        double doubleVar2 = 125.67;
        boolean booleanVar = true;
        char charVar = 65; // code for A
        char charVar2 = 'C';

        System.out.println("byteVar = "+ byteVar);
        System.out.println("shortVar = "+ shortVar);
        System.out.println("intVar = "+ intVar);
        System.out.println("longVar = "+ longVar);
        System.out.println("floatVar = "+ floatVar);
        System.out.println("doubleVar = "+ doubleVar);
        System.out.println("doubleVar2 = "+ doubleVar2);
        System.out.println("booleanVar = "+ booleanVar);
        System.out.println("charVar = "+ charVar);
        System.out.println("charVar2 = "+ charVar2);
    }
 }
```

Output:

```
byteVar = 123
shortVar = 1234
intVar = 123456
longVar = 3124567891
floatVar = 123.45
doubleVar = 12345.6789
doubleVar2 = 125.67
booleanVar = true
charVar = A
charVar2 = C
```

Non Primitive data type

Non-primitive data types are generally created by the programmer. In java every class or interface acts like a data type. Any class or interface created by you or already created in java are non primitive data types. Classes, interfaces, arrays, String, StringBuilder, Integer, Character etc defined by java or by programmer is basically the non-primitive data type. Non primitive data types are generally built using primitive data types, for example classes contains primitive variables.

Variables of non-primitive data type doesn't contain the value directly, instead they contain a **reference**(address) to an object in memory. That is why non primitive types are also called as **reference types**. If you access a non-primitive variable, it will return a reference not the value. The objects of non primitive type may contain any type of data, primitive or non-primitive. Except primitive data type everything is non-primitive data type in java.

Non primitive data type program in Java

```
class NonPrimitiveDataTypeExample {
   public static void main(String[] args) {
      // String is a non primitive data type define in Java
      String nonPrimStr = "String is a non primitive data type";
      System.out.println("nonPrimStr = "+ nonPrimStr);
      // Integer is a non primitive data type define in Java
      Integer intVal = new Integer(10);
      System.out.println("intVal = "+ intVal);
      // This class itself is a non primitive data type
      NonPrimitiveDataTypeExample np = new NonPrimitiveDataTypeExample();
      System.out.println("np = "+ np.toString());
   }
}
```

Output:

```
nonPrimStr = String is a non primitive data type
intVal = 10
np = NonPrimitiveDataTypeExample@15db9742
```

Literals in Java are a synthetic representation of boolean, character, numeric, or string data. They are a means of expressing particular values within a program. They are constant values that directly appear in a program and can be assigned now to a variable. For example, here is an integer variable named ‘’/count assigned as an integer value in this statement:

```
int count = 0;
```

“int count” is the integer variable, and a literal ‘0’ represents the value of zero.

Therefore, a constant value that is assigned to the variable can be called a literal.

Literals in Java:

Literals in Java are typically classified into six types and then into various sub-types. The primary literal types are:

1. Integral Literals

Integral literals consist of digit sequences and are broken down into these sub-types:

- Decimal Integer: Decimal integers use a base ten and digits ranging from 0 to 9. They can have a negative (-) or a positive (+), but non-digit characters or commas aren't allowed between characters. Example: 2022, +42, -68.
- Octal Integer: Octal integers use a base eight and digits ranging from 0 to 7. Octal integers always begin with a "0." Example: 007, 0295.
- Hexa-Decimal: Hexa-decimal integers work with a base 16 and use digits from 0 to 9 and the characters of A through F. The characters are case-sensitive and represent a 10 to 15 numerical range. Example: 0xf, 0xe.
- Binary Integer: Binary integers uses a base two, consisting of the digits "0" and "1." The prefix "0b" represents the Binary system. Example: 0b11011.

2. Floating-Point Literals

Floating-point literals are expressed as exponential notations or as decimal fractions. They can represent either a positive or negative value, but if it's not specified, the value defaults to positive. Floating-point literals come in these formats:

- Floating: Floating format single precision (4 bytes) end with an "f" or "F." Example: 4f. Floating format double precision (8 bytes) end with a "d" or "D." Example: 3.14d.
- Decimal: This format uses 0 through 9 and can have either a suffix or an exponent. Example: 99638.440.
- Decimal in Exponent form: The exponent form may use an optional sign, such as a "-," and an exponent indicator, such as "e" or "E." Example: 456.5f.

3. Char Literals

Character (Char) literals are expressed as an escape sequence or a character, enclosed in single quote marks, and always a type of character in Java. Char literals are sixteen-bit Unicode characters ranging from 0 to 65535. Example: char ch = 077.

4. String Literals

String literals are sequences of characters enclosed between double quote ("") marks. These characters can be alphanumeric, special characters, blank spaces, etc.

Examples: "John", "2468", "\n", etc.

5. Boolean Literals

Boolean literals have only two values and so are divided into two literals:

- True represents a real boolean value
- False represents a false boolean value

So, Boolean literals represent the logical value of either true or false. These values aren't case-sensitive and are equally valid if rendered in uppercase or lowercase mode. Boolean literals can also use the values of "0" and "1."

Examples:

boolean b = true;

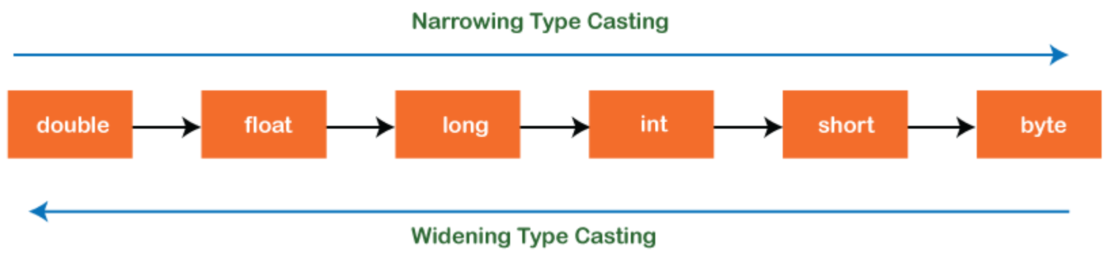
boolean d = false;

6. Null Literals

Null literals represent a null value and refer to no object. Nulls are typically used as a marker to indicate that a reference type object isn't available. They often describe an uninitialized state in the program. It is a mistake to try to dereference a null value. Example: Patient age = NULL;

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 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.

Example:

```
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

Example:

```
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
```

Operator & Expression:

Operators in Java are the symbols used for performing specific operations in Java.

Types of Operators in Java

There are multiple types of operators in Java all are mentioned below:

1. Arithmetic Operators
2. Unary Operators
3. Assignment Operator
4. Relational Operators
5. Logical Operators
6. Ternary Operator
7. Bitwise Operators
8. Shift Operators
9. instance of operator

1. Arithmetic Operators

They are used to perform simple arithmetic operations on primitive data types.

- *** :** Multiplication
- **/ :** Division

- **% :** Modulo
- **+ :** Addition
- **– :** Subtraction

Example:

```
// Arithmetic Operators
import java.io.*;

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

        int a = 10;
        int b = 3;

        System.out.println("a + b = " + (a + b));
        System.out.println("a - b = " + (a - b));
        System.out.println("a * b = " + (a * b));
        System.out.println("a / b = " + (a / b));
        System.out.println("a % b = " + (a % b));

    }
}
```

Output

```
a + b = 13
a - b = 7
a * b = 30
a / b = 3
a % b = 1
```

2. Unary Operators

Unary operators need only one operand. They are used to increment, decrement, or negate a value.

- **– : Unary minus**, used for negating the values.
- **+ : Unary plus** indicates the positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is the byte, char, or short. This is called unary numeric promotion.
- **++ : Increment operator**, used for incrementing the value by 1. There are two varieties of increment operators.
    - **Post-Increment:** Value is first used for computing the result and then incremented.
    - **Pre-Increment:** Value is incremented first, and then the result is computed.

- **– – : Decrement operator**, used for decrementing the value by 1. There are two varieties of decrement operators.
    - **Post-decrement:** Value is first used for computing the result and then decremented.
    - **Pre-Decrement: The value** is decremented first, and then the result is computed.
- **! : Logical not operator**, used for inverting a boolean value.

Example:

```
// Uniary Operators
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        int a = 10;
        int b = 10;

        System.out.println("Postincrement : " + (a++));
        System.out.println("Preincrement : " + (++a));
        System.out.println("Postdecrement : " + (b--));
        System.out.println("Predecrement : " + (--b));
    }
}
```

Output

```
Postincrement : 10
Preincrement : 12
Postdecrement : 10
Predecrement : 8
```

3. Assignment Operator

'=' Assignment operator is used to assign a value to any variable. It has right-to-left associativity, i.e. value given on the right-hand side of the operator is assigned to the variable on the left, and therefore right-hand side value must be declared before using it or should be a constant.
The general format of the assignment operator is: variable = value;

In many cases, the assignment operator can be combined with other operators to build a shorter version of the statement called a **Compound Statement**. For example, instead of a = a+5, we can write a += 5.

- **+=**, for adding the left operand with the right operand and then assigning it to the variable on the left.
- **-=**, for subtracting the right operand from the left operand and then assigning it to the variable on the left.
- ***=**, for multiplying the left operand with the right operand and then assigning it to the variable on the left.
- **/=**, for dividing the left operand by the right operand and then assigning it to the variable on the left.
- **%=**, for assigning the modulo of the left operand by the right operand and then assigning it to the variable on the left.

Example:

```
// Assignment Operators
import java.io.*;

class GFG {
   public static void main(String[] args)
   {
      int f = 7;
       System.out.println("f += 3: " + (f += 3));
       System.out.println("f -= 2: " + (f -= 2));
       System.out.println("f *= 4: " + (f *= 4));
       System.out.println("f /= 3: " + (f /= 3));
       System.out.println("f %= 2: " + (f %= 2));
       System.out.println("f &= 0b1010: " + (f &= 0b1010));
       System.out.println("f |= 0b1100: " + (f |= 0b1100));
       System.out.println("f ^= 0b1010: " + (f ^= 0b1010));
       System.out.println("f <<= 2: " + (f <<= 2));
       System.out.println("f >>= 1: " + (f >>= 1));
       System.out.println("f >>>= 1: " + (f >>>= 1));
   }
}
```

Output

```
f += 3: 10
f -= 2: 8
f *= 4: 32
f /= 3: 10
f %= 2: 0
f &= 0b1010: 0
f |= 0b1100: 12
f ^= 0b1010: 6
```

```
f <<= 2: 24
f >>= 1: 12
f >>>= 1: 6
```

4. Relational Operators

These operators are used to check for relations like equality, greater than, and less than. They return boolean results after the comparison and are extensively used in looping statements as well as conditional if-else statements.

The general format is, variable **relation_operator** value

Some of the relational operators are-

- **==, Equal to** returns true if the left-hand side is equal to the right-hand side.
- **!=, Not Equal to** returns true if the left-hand side is not equal to the right-hand side.
- **<, less than:** returns true if the left-hand side is less than the right-hand side.
- **<=, less than or equal to** returns true if the left-hand side is less than or equal to the right-hand side.
- **>, Greater than:** returns true if the left-hand side is greater than the right-hand side.
- **>=, Greater than or equal to** returns true if the left-hand side is greater than or equal to the right-hand side.

Example:

```
// Relational Operators
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        int a = 10;
        int b = 3;
        int c = 5;

        System.out.println("a > b: " + (a > b));
        System.out.println("a < b: " + (a < b));
        System.out.println("a >= b: " + (a >= b));
        System.out.println("a <= b: " + (a <= b));
        System.out.println("a == c: " + (a == c));
        System.out.println("a != c: " + (a != c));
    }
}
```

Output

```
a > b: true
```

```
a < b: false
a >= b: true
a <= b: false
a == c: false
a != c: true
```

5. Logical Operators

These operators are used to perform "logical AND" and "logical OR" operations, i.e., a function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e., it has a short-circuiting effect. Used extensively to test for several conditions for making a decision. Java also has "Logical NOT", which returns true when the condition is false and vice-versa

Conditional operators are:

- **&&, Logical AND:** returns true when both conditions are true.
- **||, Logical OR:** returns true if at least one condition is true.
- **!, Logical NOT:** returns true when a condition is false and vice-versa

Example:

```
// Logical operators
import java.io.*;
 class GFG {
    public static void main (String[] args) {
        boolean x = true;
        boolean y = false;
        System.out.println("x && y: " + (x && y));
        System.out.println("x || y: " + (x || y));
        System.out.println("!x: " + (!x));
    }
}
```

Output

```
x && y: false
x || y: true
!x: false
```

6. Ternary operator

The ternary operator is a shorthand version of the if-else statement. It has three operands and hence the name Ternary.

The general format is:

condition **?** if true **:** if false

The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

Example:

```
// max of three numbers using  ternary operator.
public class operators {
    public static void main(String[] args)
    {
        int a = 20, b = 10, c = 30, result;


        result= ((a > b) ? (a > c) ? a : c : (b > c) ? b : c);
        System.out.println("Max of three numbers = " + result);
    }
}
```

Output

```
Max of three numbers = 30
```

7. Bitwise Operators

These operators are used to perform the manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of the Binary indexed trees.

- **&, Bitwise AND operator:** returns bit by bit AND of input values.
- **|, Bitwise OR operator:** returns bit by bit OR of input values.
- **^, Bitwise XOR operator:** returns bit-by-bit XOR of input values.
- **~, Bitwise Complement Operator:** This is a unary operator which returns the one's complement representation of the input value, i.e., with all bits inverted.

Example

```
// bitwise operators
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        int d = 0b1010;
        int e = 0b1100;
        System.out.println("d & e: " + (d & e));
        System.out.println("d | e: " + (d | e));
        System.out.println("d ^ e: " + (d ^ e));
        System.out.println("~d: " + (~d));
        System.out.println("d << 2: " + (d << 2));
        System.out.println("e >> 1: " + (e >> 1));
        System.out.println("e >>> 1: " + (e >>> 1));
    }
}
```

Output

```
d & e: 8
d | e: 14
d ^ e: 6
~d: -11
d << 2: 40
e >> 1: 6
e >>> 1: 6
```

8. Shift Operators

These operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two.

General format- number **shift_op** number_of_places_to_shift;

- **<<, Left shift operator:** shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as multiplying the number with some power of two.
- **>>, Signed Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of the initial number. Similar effect to dividing the number with some power of two.
- **>>>, Unsigned Right shift operator:** shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

Program:

```
// shift operators
import java.io.*;

class GFG {
    public static void main(String[] args)
    {
        int a = 10;
        System.out.println("a<<1 : " + (a << 1));
        System.out.println("a>>1 : " + (a >> 1));
    }
}
```

Output

```
a<<1 : 20
```

```
a>>1 : 5
```

9. instanceof operator

The instance of the operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass, or an interface. General format-
object **instance of** class/subclass/interface

Program:

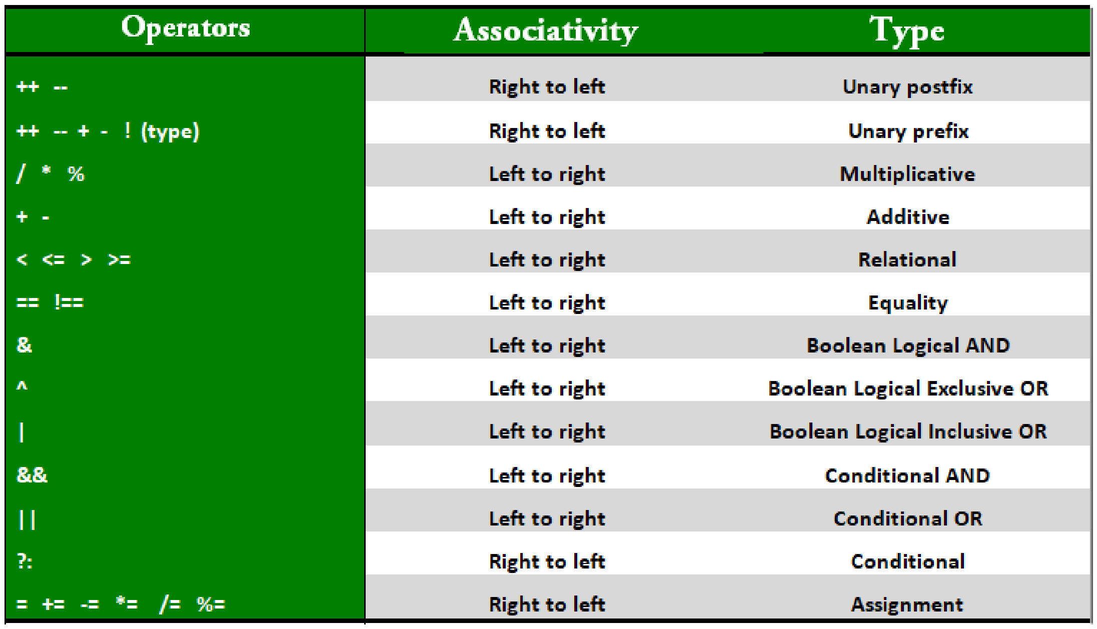
```
// instance of operator
class operators {
   public static void main(String[] args)
   {

       Person obj1 = new Person();
       Person obj2 = new Boy();

       // As obj is of type person, it is not an  instance of Boy or interface
       System.out.println("obj1 instanceof Person: "+ (obj1 instanceof Person));
       System.out.println("obj1 instanceof Boy: "+ (obj1 instanceof Boy));
       System.out.println("obj1 instanceof MyInterface: "+ (obj1 instanceof MyInterface));

       // Since obj2 is of type boy, whose parent class is person  and it implements the interface
//Myinterface  it is instance of all of these classes
       System.out.println("obj2 instanceof Person: " + (obj2 instanceof Person));
       System.out.println("obj2 instanceof Boy: " + (obj2 instanceof Boy));
       System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
   }
}
class Person {
}
class Boy extends Person implements MyInterface {
}
interface MyInterface {
}
```

Output

```
obj1 instanceof Person: true
obj1 instanceof Boy: false
obj1 instanceof MyInterface: false
obj2 instanceof Person: true
```

obj2 instanceof Boy: true
obj2 instanceof MyInterface: true

Precedence and Associativity of Java Operators:
Precedence and associative rules are used when dealing with hybrid equations involving more than one type of operator. In such cases, these rules determine which part of the equation to consider first, as there can be many different valuations for the same equation. The below table depicts the precedence of operators in decreasing order as magnitude, with the top representing the highest precedence and the bottom showing the lowest precedence.



Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ! (type)	Right to left	Unary prefix
/ * %	Left to right	Multiplicative
+ -	Left to right	Additive
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Control Flow Statement:

A programming language uses control statements to control the flow of execution of a program based on certain conditions. These are used to cause the flow of execution to advance and branch based on changes to the state of a program.
Java's Selection statements:

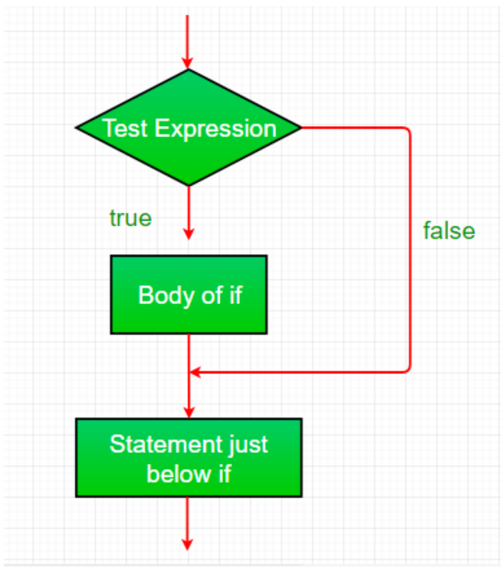
- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

1. if: if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax:

```
if(condition)
{
   // Statements to execute if
   // condition is true
}
```

Here, the **condition** after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements under it.



Example:

```
// Java program to illustrate If statement without curly block
import java.util.*;
 class IfDemo {
    public static void main(String args[])
    {
        int i = 10;
         if (i < 15)
            System.out.println("Inside If block");
            System.out.println("10 is less than 15");
            System.out.println("I am Not in if");
    }
}
```

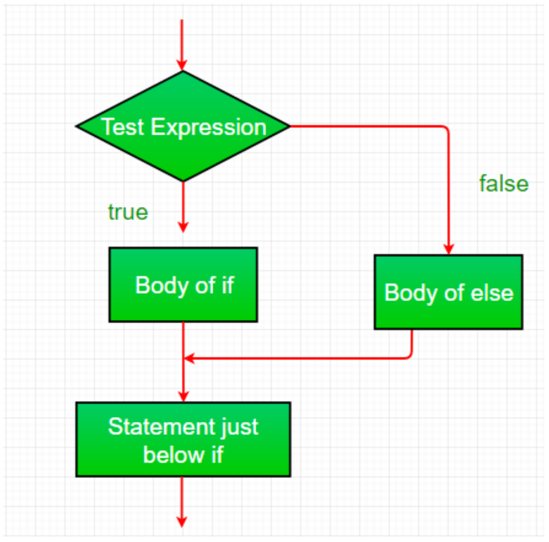
Output

```
Inside If block
10 is less than 15
I am Not in if
```

2. if-else: Here if the condition is true it will execute a true block of statements and if the condition is false it will execute a false block of statements.

Syntax:

```
if (condition)
{
   // Executes this block if
   // condition is true
}
else
{
   // Executes this block if
   // condition is false
}
```



Example:

```
// Java program to illustrate if-else statement
import java.util.*;
 class IfElseDemo {
   public static void main(String args[])
   {
      int i = 10;
      if (i < 15)
         System.out.println("i is smaller than 15");
      else
         System.out.println("i is greater than 15");
   }
}
```

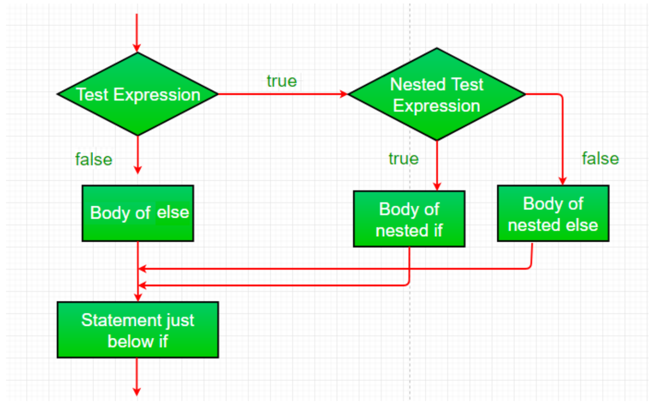
Output

```
i is smaller than 15
```

3. nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement.

Syntax:

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```



Example:

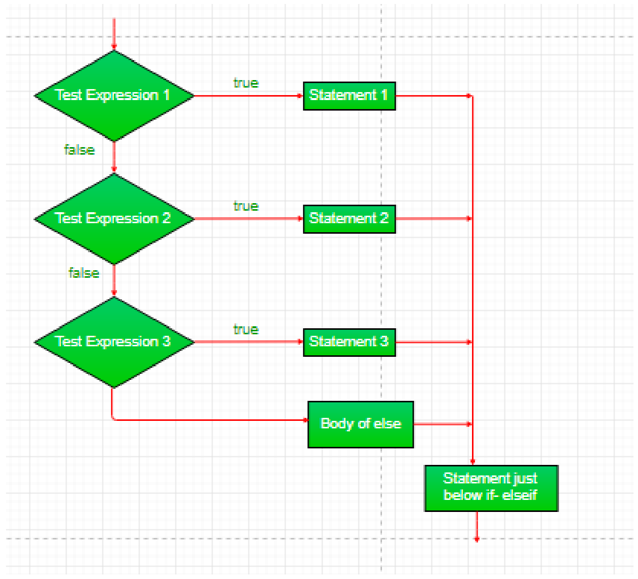
```
// Java program to illustrate nested-if statement
import java.util.*;
class NestedIfDemo {
    public static void main(String args[])
    {
        int i = 10;
         if (i == 10 || i<15) {
            // First if statement
            if (i < 15)
                System.out.println("i is smaller than 15");
             // Nested - if statement will only be executed if statement above is true
            if (i < 12)
                System.out.println(
                    "i is smaller than 12 too");
        } else{
                System.out.println("i is greater than 15");
        }
    }
}
```

Output

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

4. if-else-if ladder: Here, a user can decide among multiple options. The if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

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



Example:

```
// Java program to illustrate if-else-if ladder
import java.util.*;
class ifelseifDemo {
    public static void main(String args[])
    {
        int i = 20;
         if (i == 10)
            System.out.println("i is 10");
        else if (i == 15)
```

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

Output

```
i is 20
```

5. switch-case: The switch statement is a multiway branch statement. It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax:

```
switch (expression)
{
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;
  .
  .
  case valueN:
    statementN;
    break;
  default:
    statementDefault;
}
```

Program:

```
import java.io.*;
class GFG {
    public static void main (String[] args) {
        int num=20;
          switch(num){
          case 5 :  System.out.println("It is 5");
                    break;
          case 10 : System.out.println("It is 10");
                    break;
          case 15 : System.out.println("It is 15");
```

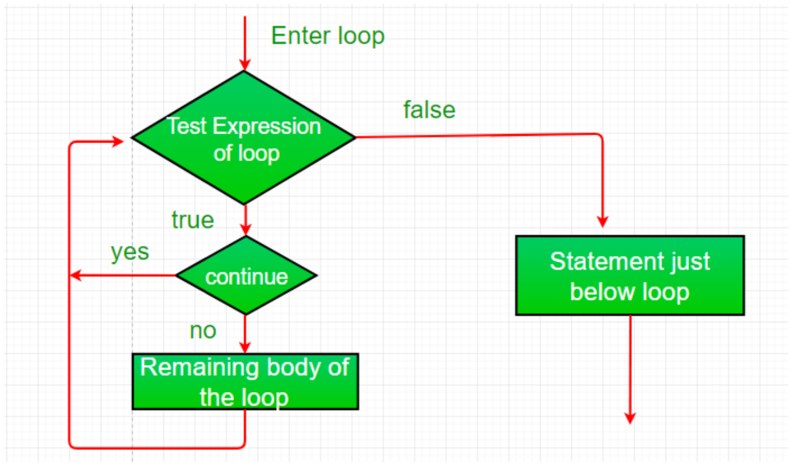
```
                break;
          case 20 : System.out.println("It is 20");
                break;
          default:  System.out.println("Not present");

        }
    }
}
```
Output

```
It is 20
```

6. jump: Java supports three jump statements: **break, continue** and **return**. These three statements transfer control to another part of the program.

- **Break:** In Java, a break is majorly used for:
    - Terminate a sequence in a switch statement.
    - To exit a loop.

  Continue: Sometimes it is useful to force an early iteration of a loop. This is used to continue running the loop but stop processing the remainder of the code in its body for this particular iteration.



Example:

```
// Java program to illustrate using continue in an if statement
import java.util.*;
class ContinueDemo {
   public static void main(String args[])
   {
       for (int i = 0; i < 10; i++) {
           // If the number is even skip and continue
```

```
            if (i % 2 == 0)
                continue;
             // If number is odd, print it
            System.out.print(i + " ");
        }
    }
}
```
Output

```
1 3 5 7 9
```

- **Return:** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

Example:

```
// Java program to illustrate using return
import java.util.*;

public class Return {
    public static void main(String args[])
    {
        boolean t = true;
        System.out.println("Before the return.");

        if (t)
            return;
         System.out.println("This won't execute.");
    }
}
```
Output

```
Before the return.
```

Loops in Java:

Looping in programming languages is a feature which facilitates the execution of a set of instructions/functions repeatedly while some condition evaluates to true. Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

java provides Three types of Conditional statements this second type is loop statement .

- **while loop:** A while loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax :

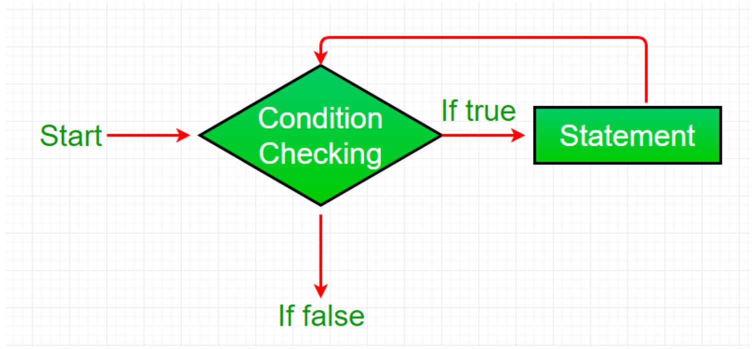
```
while (boolean condition)
{
  loop statements...
}
```

Program:

```
import java.io.*;
class GFG {
    public static void main (String[] args) {
      int i=0;
      while (i<=10)
      {
        System.out.println(i);
        i++;
      }
    }
}
```

Output

```
0
1
2
3
4
5
6
7
8
9
10
```



- While loop starts with the checking of Boolean condition. If it evaluated to true, then the loop body statements are executed otherwise first statement following the loop is executed. For this reason it is also called **Entry control loop**

    - Once the condition is evaluated to true, the statements in the loop body are executed. Normally the statements contain an update value for the variable being processed for the next iteration.
    - When the condition becomes false, the loop terminates which marks the end of its life cycle.
- **for loop:** for loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

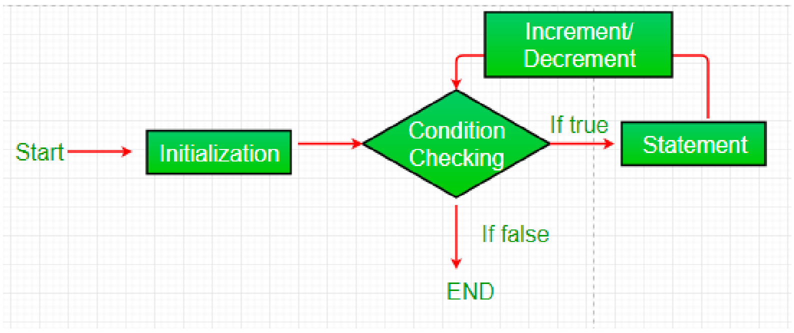
Syntax:

```
for (initialization condition; testing condition;increment/decrement)

{
   statement(s)
}
```

```
import java.io.*;
 class GFG {
    public static void main (String[] args) {
       for (int i=0;i<=10;i++)
       {
         System.out.println(i);
       }
    }
}
```

Output

```
0
1
2
3
4
5
6
7
8
9
10
```



- **Initialization condition:** Here, we initialize the variable in use. It marks the start of a for loop. An already declared variable can be used or a variable can be declared, local to loop only.
- **Testing Condition:** It is used for testing the exit condition for a loop. It must return a boolean value. It is also an **Entry Control Loop** as the condition is checked prior to the execution of the loop statements.
- **Statement execution:** Once the condition is evaluated to true, the statements in the loop body are executed.
- **Increment/ Decrement:** It is used for updating the variable for next iteration.
- **Loop termination:** When the condition becomes false, the loop terminates marking the end of its life cycle.

- **do while:** do while loop is similar to while loop with only difference that it checks for condition after executing the statements, and therefore is an example of **Exit Control Loop.**

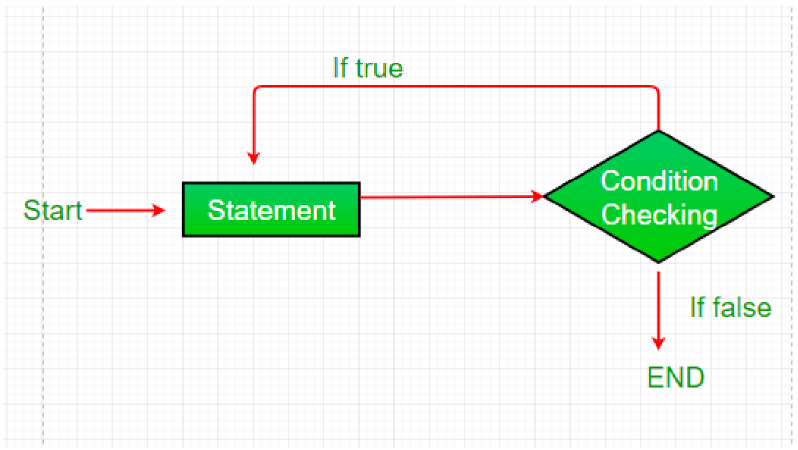
Syntax:

```
do

{

   statements..

}

while (condition);
```

```
import java.io.*;
 class GFG {
    public static void main (String[] args) {
      int i=0;
      do
      {
        System.out.println(i);
        i++;
      }while(i<=10);
    }
}
```

Output

```
0
1
2
3
4
5
6
7
8
9
10
```



- do while loop starts with the execution of the statement(s). There is no checking of any condition for the first time.
- After the execution of the statements, and update of the variable value, the condition is checked for true or false value. If it is evaluated to true, next iteration of loop starts.
- When the condition becomes false, the loop terminates which marks the end of its life cycle.
- It is important to note that the do-while loop will execute its statements atleast once before any condition is checked, and therefore is an example of exit control loop.

COMMAND LINE ARGUMENT:

The java command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input. So, it provides a convenient way to check the behavior of the program for the different values.

Example

```
// Program to receive a string from command line and print it on the console
class CommandLineExample{
public static void main(String args[]){
System.out.println("Your first argument is: "+args[0]);
}
}
compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo
```

Example:

Here the program prints all the arguments passed from the command-line by using for loop.

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

for(int i=0;i<args.length;i++)
System.out.println(args[i]);

}
}
compile by > javac A.java
run by > java A sonoo jaiswal 1 3 abc

Output: sonoo
        jaiswal
        1
        3
        abc
```

Chapter-3
OBJECT AND CLASSES

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class. An object is an entity that has state and behavior. It can be physical or logical (tangible and intangible).

An object has three characteristics:

- State: represents the data (value) of an object.
- Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

Class in Java:

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- 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 a 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. In object or instance is created.

Example:(main within the class)
Here a Student class which has two data members id and name. An object is created by new keyword and printing the object's value.

```
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

In real time development, we create classes and use it from another class.
Example:

```
//Java Program to demonstrate having the main method in another class
class Student{
 int id;
 String name;
}
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

Initializing object:

There are 3 ways to initialize object in Java.

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

1) Initialization through reference

Initializing an object means storing data into the object. In this example we are initializing the object through a reference variable.

Program:

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Sonoo
```

2) Initialization through method.

Here two objects of Student class are created and initializing the value to these objects by invoking the insertRecord method.

Program:

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
Output:
111 Karan
222 Aryan
```

3) Initialization through a constructor

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.

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
3. A Java constructor cannot be abstract, static, final, and synchronized

There are two types of constructors in Java:

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

Java Default Constructor:
A constructor is called "Default Constructor" when it doesn't have any parameter.
Syntax of default constructor:
<class_name>(){}

Example

```
//Java Program to create and call a default constructor
class Bike1{
Bike1(){System.out.println("Bike is created");}
public static void main(String args[]){
Bike1 b=new Bike1();
}
}
```
Output:
```
Bike is created
```

Java Parameterized Constructor:

A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects.

Example

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
   Student4(int i,String n){
   id = i;
   name = n;
   }
   void display(){System.out.println(id+" "+name);}

   public static void main(String args[]){
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```
Output:
```
111 Karan
222 Aryan
```

Constructor Overloading in Java:

In Java, we can also overloaded the constructor like Java methods. Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example

```
//Java program to overload constructors
```

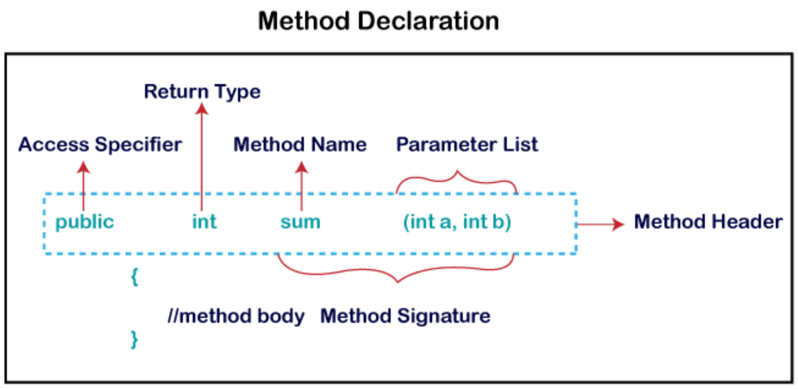
```
class Student5{
    int id;
    String name;
    int age;
    Student5(int i,String n){
    id = i;
    name = n;
    }
    Student5(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student5 s1 = new Student5(111,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Method in Java:

A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.



Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.

Access Specifier: Access specifier or modifier is the access type of the method. It specifies the visibility of the method.

Java provides four types of access specifier:

- Public: The method is accessible by all classes when we use public specifier in our application.
- Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
- Default: When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

The method name must be a verb and start with a lowercase letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in uppercase except the first word. For example:
Single-word method name: sum(), area()
Multi-word method name: areaOfCircle(), stringComparision()

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined Method:
In Java, predefined methods are the method that is already defined in the Java class libraries. It is also known as the standard library method or built-in method. It can directly be used by calling them in the program at any point. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Example:
```
public class Demo
{
public static void main(String[] args)
{
// using the max() method of Math class
System.out.print("The maximum number is: " + Math.max(9,7));
}
}
```
Output:
```
The maximum number is: 9
```

User-defined Method:
The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Example:
```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
```

```
{
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
int num=scan.nextInt();
findEvenOdd(num);
}
public static void findEvenOdd(int num)
{
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
```

Output:

```
Enter the number: 12
12 is even
```

Chapter-5
INHERITANCE

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 (Object Oriented programming system). When one class inherit from an existing class, it can reuse methods and fields of the parent class and can add new methods and fields also. Inheritance represents the IS-A relationship which is also known as a *parent-child* relationship.

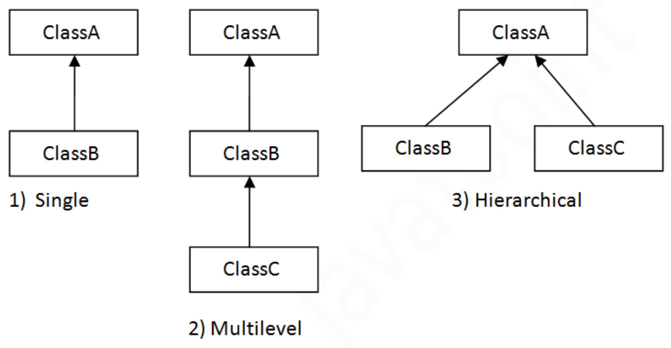
syntax :

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

The extends keyword indicates that a new class derives from an existing parent class. The class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in java

There can be three types of inheritance in java: single, multilevel and hierarchical. Multiple and hybrid inheritance is supported through interface only.



Single Inheritance

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.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

Multilevel Inheritance

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.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating..
```

Hierarchical Inheritance:

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.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}  }
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();
//c.bark();//C.T.Error
}}
```

Output:

```
meowing...
eating..
```

Interface in Java

An interface in Java is a blueprint of a class. It has static constants and abstract methods. The interface in Java is *a mechanism to achieve abstraction*. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritances in Java. It cannot be instantiated just like the abstract class. Java uses interfaces to achieve abstraction, multiple inheritance and loose coupling.

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

Syntax:

```
interface <interface_name>{

   // declare constant fields
   // declare methods that abstract by default.
}
```

In inheritance, a class extends another class, an interface extends another interface, but a class implements an interface.

Example:

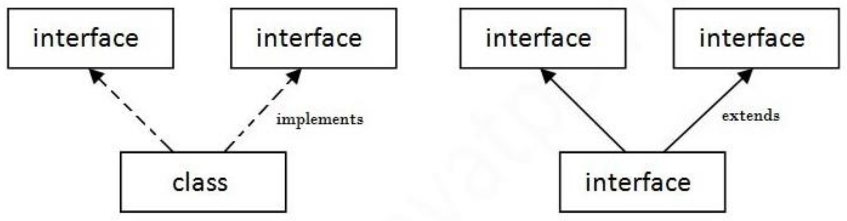
```
interface printable{
void print();  }
class A6 implements printable{
public void print(){System.out.println("Hello");}
public static void main(String args[]){
A6 obj = new A6();
obj.print();
 } }
```

Output:

```
Hello
```

Multiple inheritance

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

Example:

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 } }
```

Output:

```
Hello
Welcome
```

Chapter-6
POLYMORPHISM

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. Polymorphism in java can be formed by method overloading and method overriding.

Runtime Polymorphism

Runtime polymorphism or Dynamic Method Dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile-time. In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

```
class A{}
class B extends A{}
A a=new B();//upcasting
```

Example:

Here two classes Bike and Splendor are created. Splendor class extends Bike class and overrides its run() method. Now the run method is called by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism

```
class Bike{
  void run(){System.out.println("running");}
}
class Splendor extends Bike{
  void run(){System.out.println("running safely with 60km");}
    public static void main(String args[]){
    Bike b = new Splendor();//upcasting
    b.run();
  }
}
```

Output:

```
running safely with 60km.
```

A method is overridden but not the data members, so runtime polymorphism can't be achieved by data members.
Example:
Here both the classes have a data member speedlimit. When accessing the data member by the reference variable of Parent class which refers to the subclass object it will access the data member of the Parent class always not the data member of child class.

```
class Bike{
 int speedlimit=90;
}
class Honda3 extends Bike{
 int speedlimit=150;
 public static void main(String args[]){
  Bike obj=new Honda3();
  System.out.println(obj.speedlimit);//90
}
```
Output:
```
90
```

Method Overloading
If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1**) By changing no. of arguments**
Example:
Here two methods have been created, first add() method performs addition of two numbers and second add method performs addition of three numbers.
.
```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```
Output:
```
22
33
```

2) Bychanging data type of arguments

Example:

Here two methods are created which differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

Method Overriding

If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

Example:

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
  //defining the same method as in the parent class
  void run(){System.out.println("Bike is running safely");}

  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

Super Keyword :

The super keyword in Java is a reference variable which is used to refer immediate parent class object.Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword
super can be used to refer immediate parent class instance variable.
super can be used to invoke immediate parent class method.
super() can be used to invoke immediate parent class constructor.

Example:
Here Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden. Here Animal and Dog both classes have eat() method if eat() method is called from Dog class, it will call the eat() method of Dog class by default because priority is given to local. To call the parent class method, we need to use super keyword.

Example:

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}
```

Output:

```
eating...
barking...
```

The super keyword can also be used to invoke the parent class constructor.

Example:

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Output:

```
animal is created
dog is created
```

Final Keyword

The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable

2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these.

1) **final variable**

If a variable is declared as final, then its value cannot be changed. (It will be constant).The final variable cannot be inherited.

Example :

There is a final variable speedlimit, This program is going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }  }
```
Output:Compile Time Error

2) final method

If a method is declared as final, then it cannot be overridden. The final method is inherited but cannot be override it.

Example :
```
class Bike{
  final void run(){System.out.println("running");}
}
 class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
    public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```
Output:Compile Time Error

Example:
```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```
Output:running...

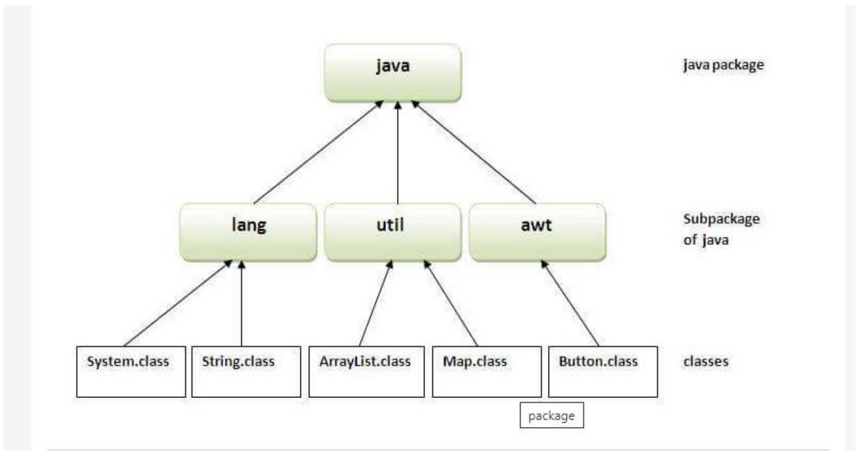
3) final class
If a class is declared as final then it cannot be inherited.

Example :
```
Bike{}
 class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
   public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```
Output:Compile Time Error

Chapter-7

PACKAGE: PUTTING CLASSES TOGETHER

A java package is a group of similar types of classes, interfaces and sub-packages.Package in java can be categorized in two form, built-in package and user-defined package.There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Advantage of Java Package
1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.



User Defined package:

The package keyword is used to create a package in java.

Syntax:

Package <Package name>;

To compile:

javac -d directory javafilename

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

To Run: java <Package name>.<class name>

Example:

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
    System.out.println("Welcome to package");
   }
}
```

Compile: javac –d Simple.java

Run: java mypack.Simple

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

If package.* is used then all the classes and interfaces of this package will be accessible but not subpackages.The import keyword is used to make the classes and interface of another package accessible to the current package.If package.classname is used then only declared class of this package will be accessible.If fully qualified name is used then only declared class of this package will be accessible. Now there is no need to import. But fully qualified name is used every time when accessing the class or interface.

Example:

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```
Output:Hello

Example:
```
//save by A.java
 package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```
Output:Hello

Example:
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
Output:Hello
```

Static Import:

The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.

Advantage:

- Less coding is required if you have access any static member of a class oftenly.

Disadvantage:

- If you overuse the static import feature, it makes the program unreadable and unmaintainable.

Example:

```
import static java.lang.System.*;
class StaticImportExample{
  public static void main(String args[]){

   out.println("Hello");//Now no need of System.out
   out.println("Java");

 }
}
Output:Hello
 Java
```

Difference between import and static import:

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.

Chapter-8
JAVA FILES 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.

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.

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
To print **output and an error** message to the console andto get **input** from consolethe following code is usefull.

System.out.println("simple message");

System.err.println("error message");

int i=System.in.read();//returns ASCII code of 1st character

System.out.println((**char**)i);//will print the character

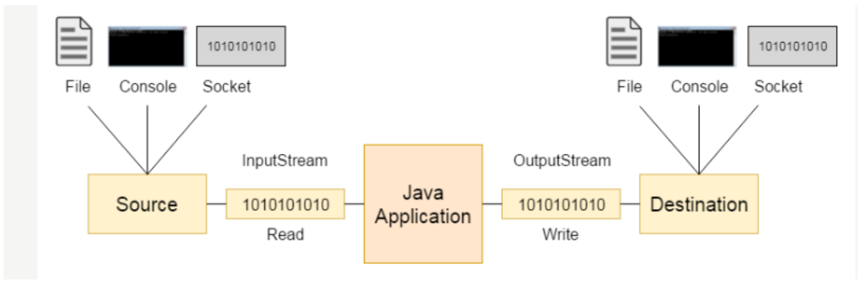
OutputStream vs InputStream:

OutputStream

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

InputStream

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



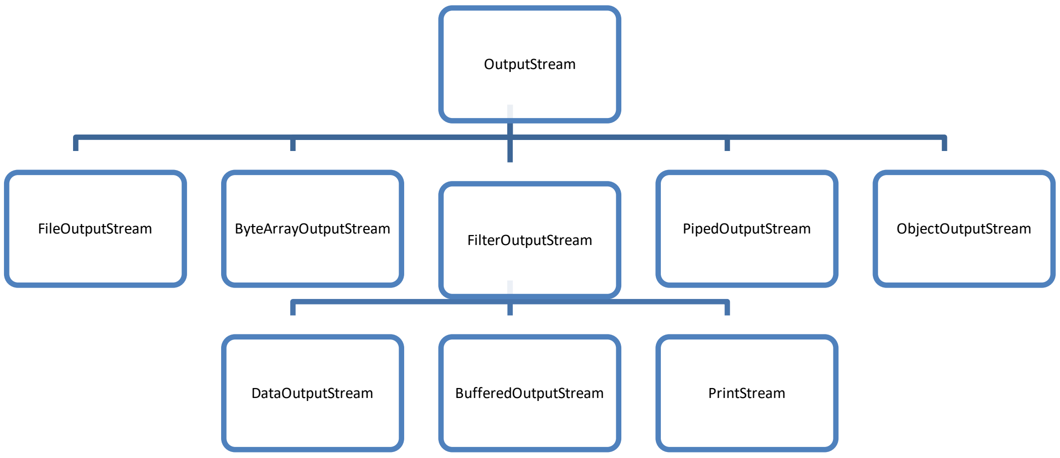
OutputStream class:
OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream
1. public void write(int)throws IOException: It is used to write a byte to the current output stream.

2. public void write(byte[])throws IOException : It is used to write an array of byte to the current output stream.

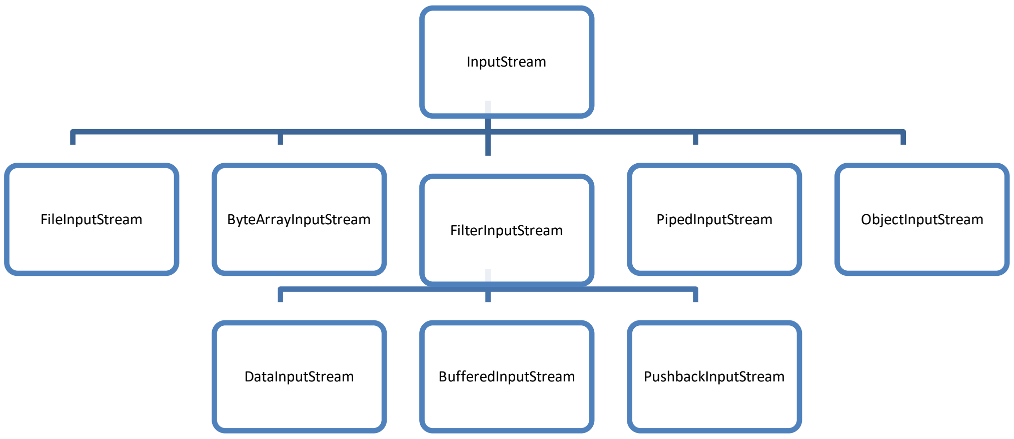
3. public void flush()throws IOException : It flushes the current output stream.

4. public void close()throws IOException: It is used to close the current output stream.



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
1. public abstract int read()throws IOException: It 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: It returns an estimate of the number of bytes that can be read from the current input stream.
3. public void close()throws IOException: It is used to close the current input stream.



Read from a file:
Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class. It is character-oriented class which is used for file handling in java.

public class FileReader extends InputStreamReader

FileReader(String file): (Constructor) It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

FileReader(File file): (Constructor) It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

int read():It is used to return a character in ASCII form. It returns -1 at the end of file.

void close():It is used to close the FileReader class.

```
package com.javatpoint;
 import java.io.FileReader;
public class FileReaderExample {
   public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("D:\\testout.txt");
        int i;
        while((i=fr.read())!=-1)
        System.out.print((char)i);
        fr.close();
   }
}
```

Output:

```
Welcome to javaTpoint.
```

Creating a file and write into the file:

To create a file in Java, createNewFile() method is used. This method returns a boolean value: true if the file was successfully created, and false if the file already exists. It is enclosed in a try...catch block because it throws an IOException if an error occurs.

```
import java.io.File;  // Import the File class
import java.io.IOException;  // Import the IOException class to handle errors
public class CreateFile {
  public static void main(String[] args) {
    try {
      File myObj = new File("filename.txt");
      if (myObj.createNewFile()) {
        System.out.println("File created: " + myObj.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }  }}
```

Output:

```
File created: filename.txt
```

Write To a File

FileWriter class together with write() method is used to write some text to the file. After writing to the file is done, it should close it with the close() method:

```
import java.io.FileWriter;   // Import the FileWriter class
import java.io.IOException;  // Import the IOException class to handle errors
public class WriteToFile {
  public static void main(String[] args) {
    try {
      FileWriter myWriter = new FileWriter("filename.txt");
      myWriter.write("Files in Java might be tricky, but it is fun enough!");
      myWriter.close();
      System.out.println("Successfully wrote to the file.");
    } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
    }
  }
}
```

Output:

```
Successfully wrote to the file.
```

Read Content From One File and Write it into Another File:

Here FileReader class is used to read the contents from a class and the FileWriter class to write it on another file.

Methods: In order to read contents from a file and write it into another file, one can use any one of the following.

1. Using the variable
2. Without using any variable

Method 1: Using the variable

```
// Java program to read content from one file  and write it into another file

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;


class GFG {
	public static void main(String[] args)
	{
		try {
		FileReader fr = new FileReader("gfgInput.txt");
		FileWriter fw = new FileWriter("gfgOutput.txt");
		String str = "";
		int i;
		while ((i = fr.read()) != -1) {
			str += (char)i; }
			System.out.println(str);
			fw.write(str);
			fr.close();
			fw.close();
	System.out.println( "File reading and writing both done");
		}catch (IOException e) {
		System.out.println("There are some IOException");
		}	} }
```

Output:

```
We loves GeekofGeeks.
File reading and writing both done
```

Method 2: Without using any variable

```
// Java program to read content from one file and write it into another file

import java.io.FileWriter;
import java.io.IOException;
class GFG {
	public static void main(String[] args)
	{
        try {
                FileWriter fw = new FileWriter("gfg.txt");
                fw.write("We love GeeksForGeeks");
                fw.close();
                System.out.println("\nFile write done");
        }catch (IOException e) {
        System.out.println("There are some IOException");
        }       }}
```

Output:

```
File reading and writing both done
```

Chapter-9
Exception Handling

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

A user has entered an invalid data.

A file that needs to be opened cannot be found.

A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

Checked exceptions − A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

Unchecked exceptions − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

```
public class Unchecked_Demo {

    public static void main(String args[]) {
    int      num[]      =      {1,      2,      3,      4};
        System.out.println(num[5]);
    }
}
```

Catching Exceptions

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following −

Syntax

```
try {
    //   Protected code
} catch (ExceptionName e1) {
     // Catch block
 }
```

The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it. Every try block should be immediately followed either by a catch block or finally block.

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

```
import java.io.*;

public class ExcepTest {

    public static void main(String args[]) { try {
            int a[] = new int[2];
            System.out.println("Access element three :" + a[3]);
        }   catch   (ArrayIndexOutOfBoundsException   e)   {
            System.out.println("Exception thrown :" + e);
        }
        System.out.println("Out of the block");
    }
}
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following −

Syntax

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

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack

```
 public class MultipleCatchBlock1 {

public static void main(String[] args) {

          try{
              int a[]=new int[5];
              a[5]=30/0;
             }
             catch(ArithmeticException e)
               {
                System.out.println("Arithmetic Exception occurs");
               }
             catch(ArrayIndexOutOfBoundsException e)
               {
                System.out.println("ArrayIndexOutOfBounds Exception occurs");
               }
             catch(Exception e)
              {
                System.out.println("Parent Exception occurs");
               }
             System.out.println("rest of the code");
   }
}
```

The Throws/Throw Keywords

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

Try to understand the difference between throws and throw keywords, *throws* is used to postpone the handling of a checked exception and *throw* is used to invoke an exception explicitly.

The following method declares that it throws a RemoteException −

Example

```
import java.io.*;
public class className {

   public void deposit(double amount) throws RemoteException {
      //   Method      implementation
      throw new RemoteException();
   }
   // Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException

Example

```
import java.io.*;
public class className {

   public void withdraw(double amount) throws RemoteException,
      InsufficientFundsException {
      // Method implementation
   }
   // Remainder of class definition
}
```

The Finally Block

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax −

Syntax

```
try {
   //   Protected code
} catch (ExceptionType1 e1) {
   //   Catch block
} catch (ExceptionType2 e2) {
    //   Catch block
} catch (ExceptionType3 e3) {
    //   Catch block
}finally {
    // The finally block always executes.
}
```

Example

```
public class ExcepTest {

    public static void main(String args[]) { int a[]
        = new int[2];
        try {
            System.out.println("Access element three :" + a[3]);
        }   catch   (ArrayIndexOutOfBoundsException   e)   {
            System.out.println("Exception thrown :" + e);
        }finally {
            a[0] = 6;
            System.out.println("First    element    value:    "    +    a[0]);
            System.out.println("The finally statement is executed");
        }
    }
}
```

This will produce the following result −

Output

```
Exception thrown      :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following −

A catch clause cannot exist without a try statement.

It is not compulsory to have finally clauses whenever a try/catch block is present. The try block cannot be present without either catch clause or finally clause.

Any code cannot be present in between the try, catch, finally blocks.