# CHAPTER-1

# Introduction to Java programming

JAVA was developed by Sun Microsystems Inc in 1991, later acquired by Oracle Corporation. It was developed by James Gosling and Patrick Naughton. It is a simple programming language. Writing, compiling and debugging a program is easy in java. It helps to create modular programs and reusable code.

## Java terminology

Before we start learning Java, lets get familiar with common java terms.

**Java Virtual Machine (JVM)**
This is generally referred as JVM. Before, we discuss about JVM lets see the phases of program execution. Phases are as follows: we write the program, then we compile the program and at last we run the program.
1) Writing of the program is of course done by java programmer like you and me.
2) Compilation of program is done by javac compiler, javac is the primary java compiler included in java development kit (JDK). It takes java program as input and generates java bytecode as output.
3) In third phase, JVM executes the bytecode generated by compiler. This is called program run phase.

So, now that we understood that the primary function of JVM is to execute the bytecode produced by compiler. **Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems**. That is why we call java as platform independent language.

**bytecode**
As discussed above, javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. The bytecode is saved in a .class file by compiler.

**Java Development Kit(JDK)**

While explaining JVM and bytecode, I have used the term JDK. Let's discuss about it. As the name suggests this is complete java development kit that includes JRE (Java Runtime Environment), compilers and various tools like JavaDoc, Java debugger etc.
In order to create, compile and run Java program you would need JDK installed on your computer.

**Java Runtime Environment(JRE)**
JRE is a part of JDK which means that JDK includes JRE. When you have JRE installed on your system, you can run a java program however you won't be able to compile it. JRE includes JVM, browser plugins and applets support. When you only need to run a java program on your computer, you would only need JRE.

# Main Features of JAVA

## Java is a platform independent language

Compiler(javac) converts source code (.java file) to the byte code(.class file). As mentioned above, JVM executes the bytecode produced by compiler. This byte code can run on any platform such as Windows, Linux, Mac OS etc. Which means a program that is compiled on windows can run on Linux and vice-versa. Each operating system has different JVM, however the output they produce after execution of bytecode is same across all operating systems. That is why we call java as platform independent language.

## Java is an Object Oriented language

Object oriented programming is a way of organizing programs as collection of objects, each of which represents an instance of a class.

4 main concepts of Object Oriented programming are:

1. Abstraction
2. Encapsulation
3. Inheritance

4. Polymorphism

## Simple

Java is considered as one of simple language because it does not have complex features like Operator overloading, Multiple inheritance, pointers and Explicit memory allocation.

## Robust Language

Robust means reliable. Java programming language is developed in a way that puts a lot of emphasis on early checking for possible errors, that's why java compiler is able to detect errors that are not easy to detect in other programming languages. The main features of java that makes it robust are garbage collection, Exception Handling and memory allocation.

## Secure

We don't have pointers and we cannot access out of bound arrays (you get ArrayIndexOutOfBoundsException if you try to do so) in java. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

## Java is distributed

Using java programming language we can create distributed applications. RMI(Remote Method Invocation) and EJB(Enterprise Java Beans) are used for creating distributed applications in java. In simple words: The java programs can be distributed on more than one systems that are connected to each other using internet connection. Objects on one JVM (java virtual machine) can execute procedures on a remote JVM.

## Multithreading

Java supports multithreading. Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilisation of CPU.

## Portable

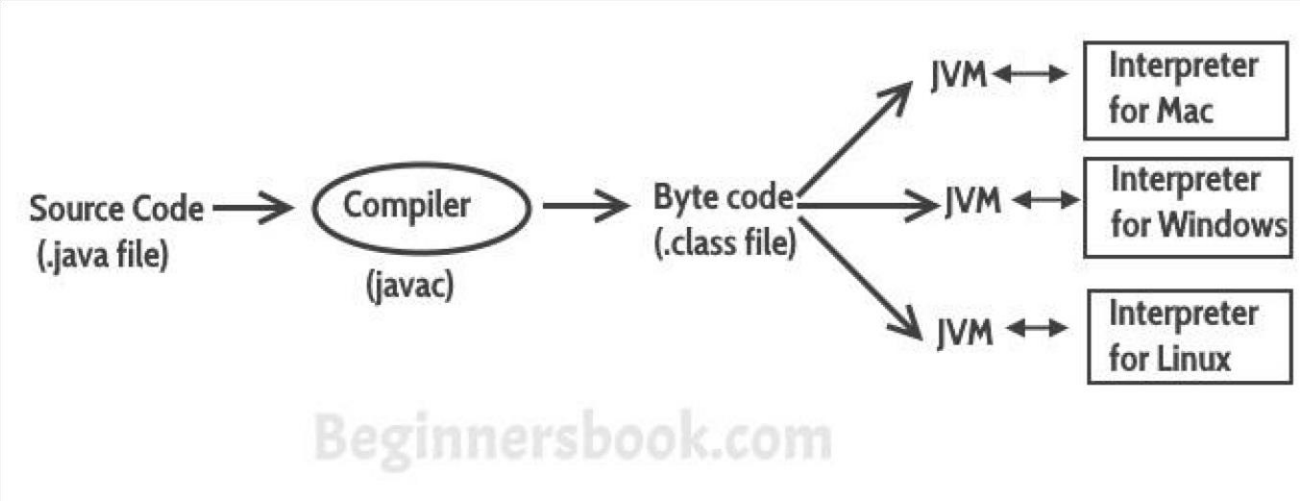
As discussed above, java code that is written on one machine can run on another machine. The platform independent byte code can be carried to any platform for execution that makes java code portable.

# Java Virtual Machine (JVM), Difference JDK, JRE & JVM – Core Java

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

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

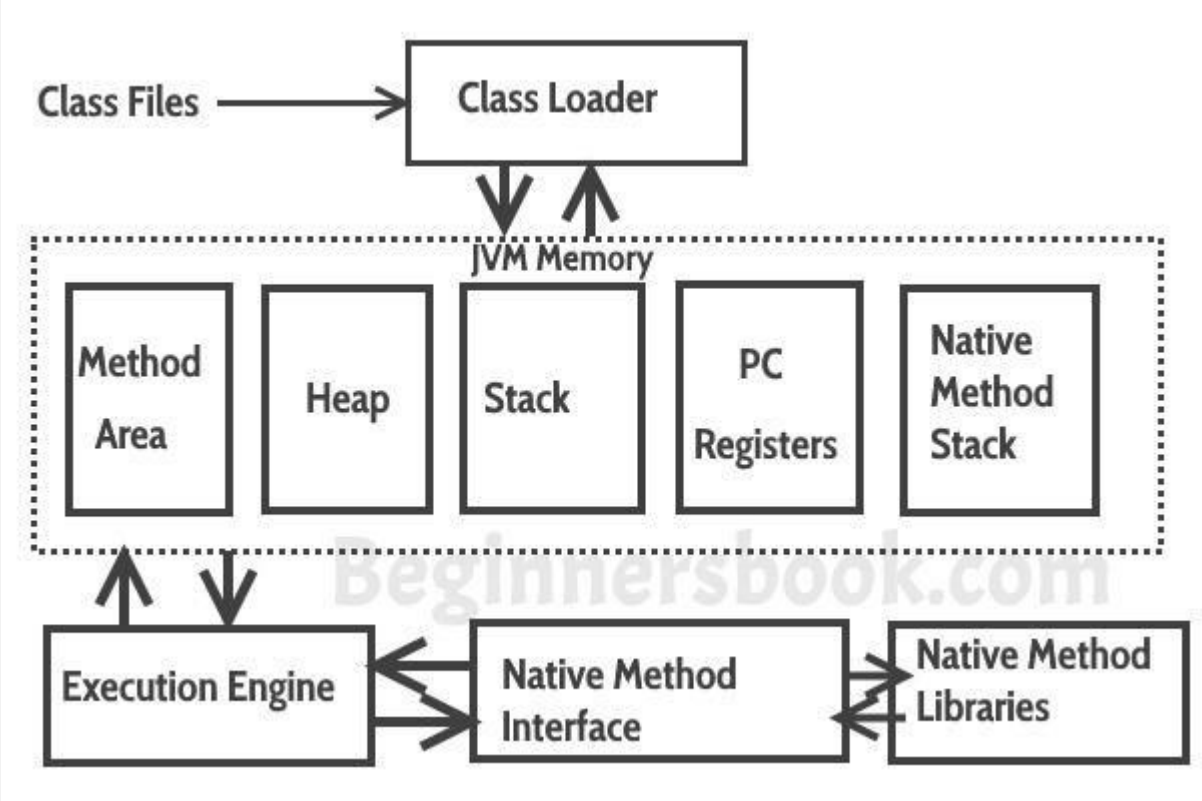
So, now we understood that the primary function of JVM is to execute the byte code produced by compiler. **Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.** Which means that the byte code generated on Windows can be run on Mac OS and vice versa. That is why we call java as platform independent language. The same thing can be seen in the diagram below:



**So to summarise everything:** The Java Virtual machine (JVM) is the virtual machine that runs on actual machine (your computer) and executes Java byte code. The JVM doesn't understand Java source code, that's why we need to have javac compiler that compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM. JVM makes java portable (write once,

run anywhere). Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.

## JVM Architecture




**Lets see how JVM works**:
**Class Loader:** The class loader reads the .class file and save the byte code in the **method area**.

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

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

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

**PC Registers**: This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC register.
**Native Method stack:** A native method can access the runtime data areas of the virtual machine.

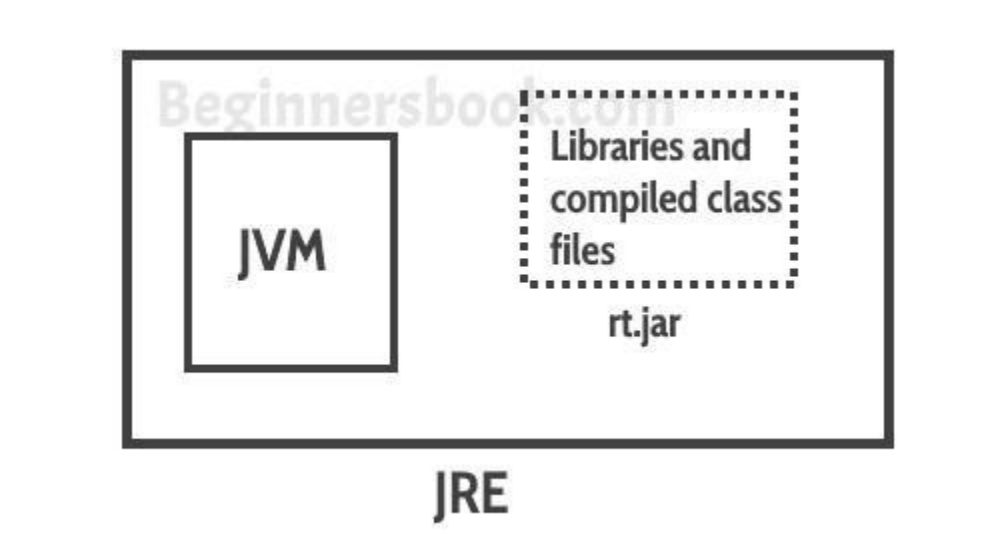
**Native Method interface**: It enables java code to call or be called by native applications. Native applications are programs that are specific to the hardware and OS of a system.

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

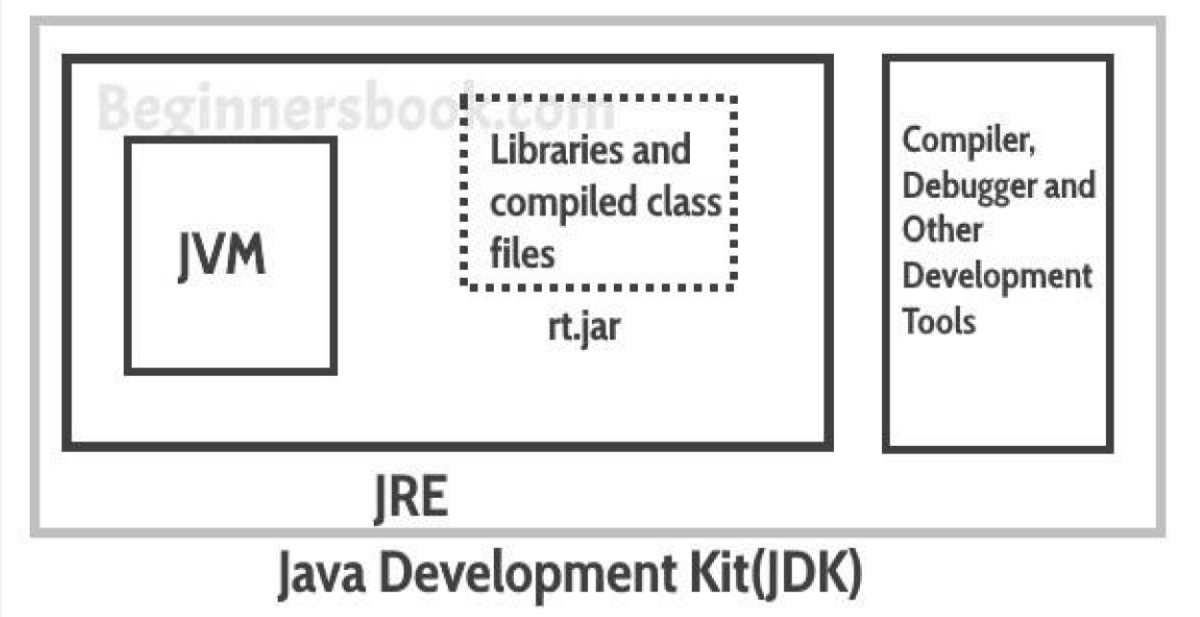
## JVM Vs JRE Vs JDK

JRE: JRE is the environment within which the java virtual machine runs. JRE contains Java virtual Machine(JVM), class libraries, and other files excluding development tools such as compiler and debugger.
Which means you can run the code in JRE but you can’t develop and compile the code in JRE.

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




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




# CHAPTER-2

# How to Compile and Run your First Java Program

In this tutorial, we will see how to write, compile and run a java program. I will also cover java syntax, code conventions and several ways to run a java program.

## Simple Java Program:

```
public class FirstJavaProgram {
  public static void main(String[] args){
    System.out.println("This is my first program in java");
  }//End of main
}//End of FirstJavaProgram Class
```

**Output:** This is my first program in java

## How to compile and run the above program

**Prerequisite:** You need to have java installed on your system. You can get the java from here.
**Step 1:** Open a text editor, like Notepad on windows and TextEdit on Mac. Copy the above program and paste it in the text editor.

You can also use IDE like Eclipse to run the java program but we will cover that part later in the coming tutorials. For the sake of simplicity, I will only use text editor and command prompt (or terminal) for this tutorial

**Step 2:** Save the file as **FirstJavaProgram.java**. You may be wondering why we have named the file as FirstJavaProgram, the thing is that we should always name the file same as the public classname. In our program, the public class name is FirstJavaProgram, that's why our file name should be **FirstJavaProgram.java**.

**Step 3:** In this step, we will compile the program. For this, open **command prompt (cmd) on Windows**, if you are **Mac OS then open Terminal**. To compile the program, type the following command and hit enter.

```
javac FirstJavaProgram.java
```

You may get this error when you try to compile the program: "**javac' is not recognized as an internal or external command, operable program or batch file**". This error occurs when the java path is not set in your system If you get this error then you first need to set the path before compilation.

**Set Path in Windows:**
Open command prompt (cmd), go to the place where you have installed java on your system and locate the bin directory, copy the complete path and write it in the command like this.

```
set path=C:\Program Files\Java\jdk1.8.0_121\bin
```

**Note:** Your jdk version may be different. Since I have java version 1.8.0_121 installed on my system, I mentioned the same while setting up the path.

**Set Path in Mac OS X**
Open Terminal, type the following command and hit return.

```
export JAVA_HOME=/Library/Java/Home
```

Type the following command on terminal to confirm the path.

```
echo $JAVA_HOME
```

That's it.

The steps above are for setting up the path temporary which means when you close the command prompt or terminal, the path settings will be lost and you will have to set the path again next time you use it. I will share the permanent path setup guide in the coming tutorial.

**Step 4:** After compilation the .java file gets translated into the .class file(byte code). Now we can run the program. To run the program, type the following command and hit enter:

```
java FirstJavaProgram
```

Note that you should not append the .java extension to the file name while running the program.

## Closer look to the First Java Program

Now that we have understood how to run a java program, let have a closer look at the program we have written above.

```
public class FirstJavaProgram {
```

This is the first line of our java program. Every java application must have at least one class definition that consists of `class` keyword followed by class name. When I say keyword, it means that it should not be changed, we should use it as it is. However the class name can be anything.

I have made the class public by using public access modifier, I will cover access modifier in a separate post, all you need to know now that a java file can have any number of classes but it can have only one public class and the file name should be same as public class name. `public static void main(String[] args)  {`

This is our next line in the program, lets break it down to understand it:

`public`: This makes the main method public that means that we can call the method from outside the class.

`static`: We do not need to create object for static methods to run. They can run itself. `void`: It does not return anything.

`main`: It is the method name. This is the entry point method from which the JVM can run your program.

`(String[] args)`: Used for command line arguments that are passed as strings. We will cover that in a separate post.

```
System.out.println("This is my first program in java");
```

This method prints the contents inside the double quotes into the console and inserts a newline after.

# CHAPTER-3

## Variables in Java

A variable is a name which is associated with a value that can be changed. For example when I write `int i=10;` here variable name is **i** which is associated with value 10, int is a data type that represents that this variable can hold integer values. We will cover the data types in the next tutorial. In this tutorial, we will discuss about variables.

### How to Declare a variable in Java

To declare a variable follow this syntax:

```
data_type variable_name = value;
```

here value is optional because in java, you can declare the variable first and then later assign the value to it.

For example: Here num is a variable and int is a data type. We will discuss the data type in next tutorial so do not worry too much about it, just understand that `int` data type allows this num variable to hold integer values. You can read

data types here but I would recommend you to finish reading this guide before proceeding to the next one.

```
int num;
```

Similarly we can assign the values to the variables while declaring them, like this:

```
char ch = 'A';
int number =
100;
```

or we can do it like this:

```
char  ch;  int
number; ... ch
=  'A';  number
= 100;
```

## Variables naming convention in java

1) Variables naming cannot contain white spaces, for example: int num ber = 100; is invalid because the variable name has space in it.
2) Variable name can begin with special characters such as $ and _
3) As per the java coding standards the variable name should begin with a lower case letter, for example int number; For lengthy variables names that has more than one words do it like this: int smallNumber; int bigNumber; (start the second word with capital letter).
4) Variable names are case sensitive in Java.

## Types of Variables in Java

There are **three types of variables** in Java.
1) Local variable 2) Static (or class) variable 3) Instance variable

## Static (or class) Variable

Static variables are also known as class variable because they are associated with the class and common for all the instances of class. For example, If I create three objects of a class and access this static variable, it would be common for all, the

changes made to the variable using one of the object would reflect when you access it through other objects. Example of static variable

```
public class StaticVarExample {
   public static String myClassVar="class or static variable";

   public static void main(String args[]){
      StaticVarExample obj = new StaticVarExample();
      StaticVarExample obj2 = new StaticVarExample();
      StaticVarExample obj3 = new StaticVarExample();

      //All three will display "class or static variable"
      System.out.println(obj.myClassVar);
      System.out.println(obj2.myClassVar);
      System.out.println(obj3.myClassVar);

      //changing the value of static variable using obj2
obj2.myClassVar = "Changed Text";

      //All three will display "Changed Text"
      System.out.println(obj.myClassVar);
      System.out.println(obj2.myClassVar);
      System.out.println(obj3.myClassVar);
   }
}
```

**Output:**

```
class or static variable class
or static variable class or
static variable
Changed Text
Changed Text
Changed Text
```

As you can see all three statements displayed the same output irrespective of the instance through which it is being accessed. That's is why we can access the static variables without using the objects like this:

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

Do note that only static variables can be accessed like this. This doesn't apply for instance and local variables.

# Instance variable

Each instance(objects) of class has its own copy of instance variable. Unlike static variable, instance variables have their own separate copy of instance variable. We have changed the instance variable value using object obj2 in the following program and when we displayed the variable using all three objects, only the obj2 value got changed, others remain unchanged. This shows that they have their own copy of instance variable.

## Example of Instance variable

```
public class InstanceVarExample {
   String myInstanceVar="instance variable";
```

```
   public static void main(String args[]){
       InstanceVarExample obj = new InstanceVarExample();
       InstanceVarExample obj2 = new InstanceVarExample();
       InstanceVarExample obj3 = new InstanceVarExample();

       System.out.println(obj.myInstanceVar);
       System.out.println(obj2.myInstanceVar);
       System.out.println(obj3.myInstanceVar);

       obj2.myInstanceVar = "Changed Text";



       System.out.println(obj.myInstanceVar);
       System.out.println(obj2.myInstanceVar);
       System.out.println(obj3.myInstanceVar);
   }
}
```

**Output:**

```
instance variable
instance variable
instance variable
instance variable
Changed Text instance
variable
```

# Local Variable

These variables are declared inside method of the class. Their scope is limited to the method which means that You can't change their values and access them outside of the method.

In this example, I have declared the instance variable with the same name as local variable, this is to demonstrate the scope of local variables.

## Example of Local variable

```
public class VariableExample {
// instance variable
   public String myVar="instance variable";
        public void
myMethod(){
      // local variable
```

```
        String myVar = "Inside Method";
        System.out.println(myVar);
    }
    public static void main(String args[]){
        // Creating object
        VariableExample obj = new VariableExample();

        /* We are calling the method, that changes the
*       value of myVar. We are displaying myVar again after
*       the method call, to demonstrate that the local
* variable scope is limited to the method itself.
        */
        System.out.println("Calling Method");
        obj.myMethod();
        System.out.println(obj.myVar);
    }
}
```

**Output:**

```
Calling Method
Inside Method
instance variable
```

If I hadn't declared the instance variable and only declared the local variable inside method then the statement System.out.println(obj.myVar); would have thrown compilation error. As you cannot change and access local variables outside the method.

# Data Types in Java

**Data type** defines the values that a variable can take, for example if a variable has int data type, it can only take integer values. In java we have two categories of data type: 1) Primitive data types 2) Non-primitive data types – Arrays and Strings are non-primitive data types, we will discuss them later in the coming tutorials. Here we will discuss primitive data types and literals in Java.

Java is a statically typed language. A language is statically typed, if the data type of a variable is known at compile time. This means that you must specify the type of the variable (Declare the variable) before you can use it.
In the last tutorial about Java Variables, we learned how to declare a variable, lets recall it:

```
int num;
```

So in order to use the variable num in our program, we must declare it first as shown above. It is a good programming practice to declare all the variables ( that you are going to use) in the beginning of the program.

# 1) Primitive data types

In Java, we have eight primitive data types: boolean, char, byte, short, int, long, float and double. Java developers included these data types to maintain the portability of java as the size of these primitive data types do not change from one operating system to another.

**byte**, **short**, **int** and **long** data types are used for storing whole numbers.

**float** and **double** are used for fractional numbers. **char** is used for storing

characters(letters).

**boolean** data type is used for variables that holds either true or false.

## byte:

This can hold whole number between -128 and 127. Mostly used to save memory and when you are certain that the numbers would be in the limit specified by byte data type.
Default size of this data type: 1 byte.
Default value: 0 Example:

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

```
        byte num;

        num = 113;
        System.out.println(num);
    }
}
```

Output:

```
113
```

Try the same program by assigning value assigning 150 value to variable num, you would get **type mismatch** error because the value 150 is out of the range of byte data type. The range of byte as I mentioned above is -128 to 127. short:

This is greater than byte in terms of size and less than integer. Its range is 32,768 to 32767.
Default size of this data type: 2 byte

```
short num = 45678;
```

**int**: Used when short is not large enough to hold the number, it has a wider range: -2,147,483,648 to 2,147,483,647
Default size: 4 byte
Default value: 0 Example:

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

        short num;

        num = 150;
        System.out.println(num);
    }
}
```

Output:

```
150
```

The byte data type couldn't hold the value 150 but a short data type can because it has a wider range.

## long:

Used when int is not large enough to hold the value, it has wider range than int data type, ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
size: 8 bytes Default
value: 0 Example:

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

       long num = -12332252626L;
System.out.println(num);
    }
}
```

Output:

```
-12332252626
```

**double**: Sufficient for holding 15 decimal digits
size: 8 bytes Example:

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

       double num = -42937737.9d;
       System.out.println(num);
    }
}
```

Output:

```
-4.29377379E7
```

**float**: Sufficient for holding 6 to 7 decimal digits
size: 4 bytes

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

```
        float num = 19.98f;
        System.out.println(num);
    }
}
```

Output:

```
19.98
```

**boolean**: holds either true of false.

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

        boolean b = false;
        System.out.println(b);
    }
}
```

Output:

```
false
```

**char**: holds characters. size:
2 bytes

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

        char ch = 'Z';
        System.out.println(ch);
    }
}
```

Output:

```
Z
```

# Literals in Java

A literal is a fixed value that we assign to a variable in a Program.

```
int num=10;
```

Here value 10 is a Integer literal.

```
char ch = 'A';
```

Here A is a char literal

## Integer Literal

Integer literals are assigned to the variables of data type byte, short, int and long.

```
byte b = 100;
short s = 200;
int num = 13313131;
long l = 928389283L;
```

## Float Literals

Used for data type float and double.

```
double num1 = 22.4; float
num2 = 22.4f;
```

Note: Always suffix float value with the “f” else compiler will consider it as double.

## Char and String Literal

Used for char and String type.

```
char ch = 'Z';
String str = "BeginnersBook";
```

# CHAPTER-4

# Operators in Java

An operator is a character that **represents an action**, for example + is an arithmetic operator that represents addition.

## Types of Operator in Java

1) Basic Arithmetic Operators
2) Assignment Operators
3) Auto-increment and Auto-decrement Operators
4) Logical Operators

5) Comparison (relational) operators
6) Bitwise Operators 7) Ternary Operator

# 1) Basic Arithmetic Operators

Basic arithmetic operators are: +, -, *, /, % **+**
is for addition.

**–** is for subtraction.

***** is for multiplication.

***/*** is for division.

**%** is for modulo.
Note: Modulo operator returns remainder, for example 10 % 5 would return 0

## Example of Arithmetic Operators

```
public class ArithmeticOperatorDemo {
public static void main(String args[]) {
int num1 = 100;       int num2 = 20;

      System.out.println("num1 + num2: " + (num1 + num2) );
      System.out.println("num1 - num2: " + (num1 - num2) );
      System.out.println("num1 * num2: " + (num1 * num2) );
      System.out.println("num1 / num2: " + (num1 / num2) );
System.out.println("num1 % num2: " + (num1 % num2) );
   }
}
```

**Output:**

```
num1 + num2: 120
num1 - num2: 80
num1 * num2: 2000
num1 / num2: 5
num1 % num2: 0
```

# 2) Assignment Operators

Assignments operators in java are: =, +=, -=, *=, /=, %= **num2 = num1** would assign value of variable num1 to the variable.

**num2+=num1** is equal to num2 = num2+num1

**num2-=num1** is equal to num2 = num2-num1

**num2*=num1** is equal to num2 = num2*num1

**num2/=num1** is equal to num2 = num2/num1

**num2%=num1** is equal to num2 = num2%num1

## Example of Assignment Operators

```
public class AssignmentOperatorDemo {
   public static void main(String args[]) {
int num1 = 10;       int num2 = 20;

      num2 = num1;
      System.out.println("= Output: "+num2);

      num2 += num1;
      System.out.println("+= Output: "+num2);

      num2 -= num1;
      System.out.println("-= Output: "+num2);

      num2 *= num1;
      System.out.println("*= Output: "+num2);

      num2 /= num1;
      System.out.println("/= Output: "+num2);

      num2 %= num1;
      System.out.println("%= Output: "+num2);
   }
}
```

**Output:**

```
= Output: 10
+= Output: 20
-= Output: 10
*= Output: 100
/= Output: 10
%= Output: 0
```

# 3) Auto-increment and Auto-decrement Operators

++ and —
**num++** is equivalent to `num=num+1;` **num—-**

is equivalent to `num=num-1;`

## Example of Auto-increment and Auto-decrement Operators

```
public class AutoOperatorDemo {
   public static void main(String args[]){
int num1=100;       int num2=200;
num1++;       num2--;
      System.out.println("num1++ is: "+num1);
      System.out.println("num2-- is: "+num2);
   }
}
```

**Output:**

```
num1++ is: 101 num2-
- is: 199
```

# 4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in java are: &&, ||, !

Let's say we have two boolean variables b1 and b2.

**b1&&b2** will return true if both b1 and b2 are true else it would return false.

**b1||b2** will return false if both b1 and b2 are false else it would return true.

**!b1** would return the opposite of b1, that means it would be true if b1 is false and it would return false if b1 is true.

## Example of Logical Operators

```
public class LogicalOperatorDemo {
public static void main(String args[]) {
boolean b1 = true;       boolean b2 =
false;

      System.out.println("b1 && b2: " + (b1&&b2));
      System.out.println("b1 || b2: " + (b1||b2));
      System.out.println("!(b1 && b2): " + !(b1&&b2));
```

```
   }
}
```

**Output:**

```
b1 && b2: false
b1 || b2: true
!(b1 && b2): true
```

# 5) Comparison(Relational) operators

We have six relational operators in Java: ==, !=, >, <, >=, <=

**==** returns true if both the left side and right side are equal
**!=** returns true if left side is not equal to the right side of operator.

**>** returns true if left side is greater than right.

**<** returns true if left side is less than right side.

**>=** returns true if left side is greater than or equal to right side.

**<=** returns true if left side is less than or equal to right side.

## Example of Relational operators

**Note:** This example is using if-else statement which is our next tutorial, if you are finding it difficult to understand then refer if-else in Java.

```
public class RelationalOperatorDemo {
public static void main(String args[]) {
int num1 = 10;       int num2 = 50;
if (num1==num2) {
        System.out.println("num1 and num2 are equal");
      }
else{
        System.out.println("num1 and num2 are not equal");
}
      if( num1 != num2 ){
        System.out.println("num1 and num2 are not equal");
```

```
      }
else{
        System.out.println("num1 and num2 are equal");
}
      if( num1 > num2 ){
        System.out.println("num1 is greater than num2");
      }
else{
        System.out.println("num1 is not greater than num2");
}
      if( num1 >= num2 ){
        System.out.println("num1 is greater than or equal to num2");
      }
else{
        System.out.println("num1 is less than num2");
      }
      if( num1 < num2 ){
        System.out.println("num1 is less than num2");
      }
else{
        System.out.println("num1 is not less than num2");
}
      if( num1 <= num2){
        System.out.println("num1 is less than or equal to num2");
      }
else{
        System.out.println("num1 is greater than num2");
      }
   }
}
```

**Output:**

```
num1 and num2 are not equal
num1 and num2 are not equal
num1 is not greater than num2
num1 is less than num2 num1 is
less than num2
num1 is less than or equal to num2
```

# 6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>

num1 = 11; /* equal to 00001011*/ num2
= 22; /* equal to 00010110 */

Bitwise operator performs bit by bit processing.
**num1 & num2** compares corresponding bits of num1 and num2 and generates 1 if both bits are equal, else it returns 0. In our case it would return: 2 which is 00000010 because in the binary form of num1 and num2 only second last bits are matching.

**num1 | num2** compares corresponding bits of num1 and num2 and generates 1 if either bit is 1, else it returns 0. In our case it would return 31 which is 00011111 **num1 ^ num2** compares corresponding bits of num1 and num2 and generates 1 if they are not equal, else it returns 0. In our example it would return 29 which is equivalent to 00011101

**~num1** is a complement operator that just changes the bit from 0 to 1 and 1 to 0. In our example it would return -12 which is signed 8 bit equivalent to 11110100

**num1 << 2** is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. In our case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

**num1 >> 2** is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. In our case output is 2 which is equivalent to 00000010

## Example of Bitwise Operators

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

```
      int num1 = 11;  /* 11 = 00001011 */
int num2 = 22;  /* 22 = 00010110 */
int result = 0;

     result = num1 & num2;
     System.out.println("num1 & num2: "+result);

     result = num1 | num2;
     System.out.println("num1 | num2: "+result);

     result = num1 ^ num2;
     System.out.println("num1 ^ num2: "+result);

     result = ~num1;
     System.out.println("~num1: "+result);

     result = num1 << 2;
     System.out.println("num1 << 2: "+result); result = num1 >> 2;
System.out.println("num1 >> 2: "+result);
  }
}
```

**Output:**

```
num1 & num2: 2
num1 | num2: 31
num1 ^ num2: 29
~num1: -12
num1 << 2: 44 num1 >> 2: 2
```

# 7) Ternary Operator

This operator evaluates a boolean expression and assign the value based on the result. **Syntax:**

```
variable num1 = (expression) ? value if true : value if false
```

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

## Example of Ternary Operator

```
public class TernaryOperatorDemo {
```

```
   public static void main(String args[]) {
int num1, num2;         num1 = 25;
        /* num1 is not equal to 10 that's why
* the second value after colon is assigned
* to the variable num2
        */
       num2 = (num1 == 10) ? 100: 200;
       System.out.println( "num2: "+num2);
       /* num1 is equal to 25 that's why
* the first value is assigned
* to the variable num2
        */
       num2 = (num1 == 25) ? 100: 200;
       System.out.println( "num2: "+num2);
   }
}
```

**Output:**

```
num2: 200
num2: 100
```

# Operator Precedence in Java

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

Unary Operators

**++ – – ! ~**

Multiplicative *****

**/ %**

Additive **+**

**–**

Shift

**<< >> >>>**

Relational
**> >= < <=**

Equality **==**
**!=**

Bitwise AND **&**

Bitwise XOR **^**

Bitwise OR
**|**

Logical AND **&&**

Logical OR
**||**

Ternary **?:**

Assignment
**= += -= *= /= %= > >= < <= &= ^= |=**

# CHAPTER-5

# If, If..else Statement in Java with Examples

When we need to execute a set of statements based on a condition then we need to use **control flow statements**. For example, if a number is greater than zero then we want to print "Positive Number" but if it is less than zero then we want to print "Negative Number". In this case we have two print statements in the program, but only one print statement executes at a time based on the input value. We will see how to write such type of conditions in the java program using control statements.

In this tutorial, we will see four types of control statements that you can use in java programs

based on the requirement: In this tutorial we will cover following conditional statements: a) if

statement

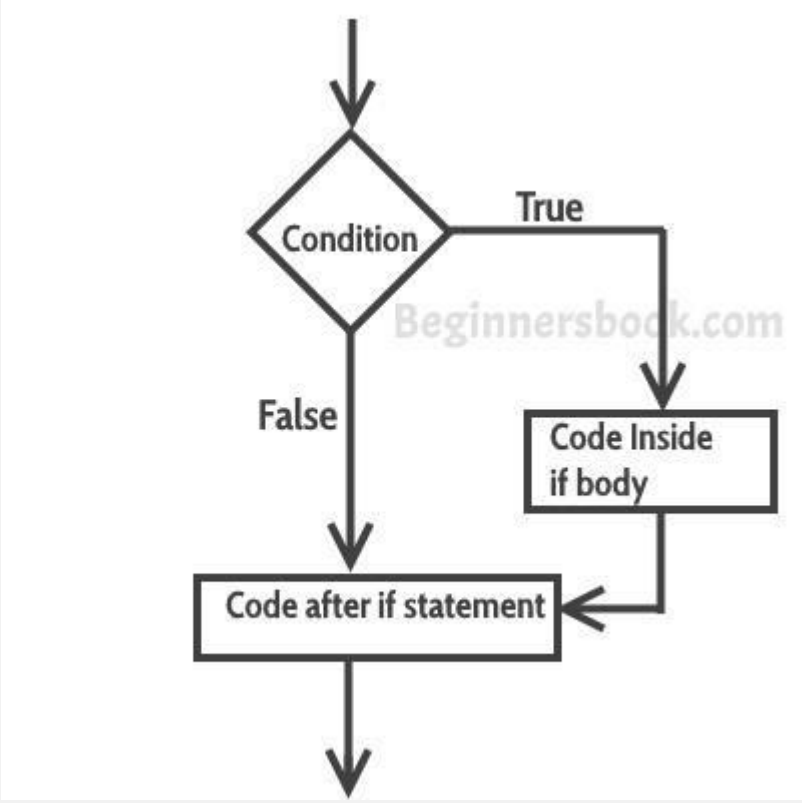
b) nested if statement
c) if-else statement
d) if-else-if statement

## If statement

If statement consists a condition, followed by statement or a set of statements as shown below:

```
if(condition){
Statement(s);
}
```

The statements gets executed only when the given condition is true. If the condition is false then the statements inside if statement body are completely ignored.




## Example of if statement

```
public class IfStatementExample {

   public static void main(String args[]){
int num=70;       if( num < 100 ){
         /* This println statement will only execute,
          * if the above condition is true
          */
         System.out.println("number is less than 100");
}
   }
}
```

**Output:**

```
number is less than 100
```

# Nested if statement in Java

When there is an if statement inside another if statement then it is called the **nested if statement**. The structure of nested if looks like this:

```
if(condition_1) {
Statement1(s);
    if(condition_2)
{
Statement2(s);
   }
}
```

Statement1 would execute if the condition_1 is true. Statement2 would only execute if both the conditions( condition_1 and condition_2) are true.

## Example of Nested if statement

```
public class NestedIfExample {


   public static void main(String args[]){
int num=70;
      if( num < 100 ){
           System.out.println("number is less than 100");
if(num > 50){
             System.out.println("number is greater than 50");
          }
      }
   }
}
```

**Output:**

```
number is less than 100 number
is greater than 50
```

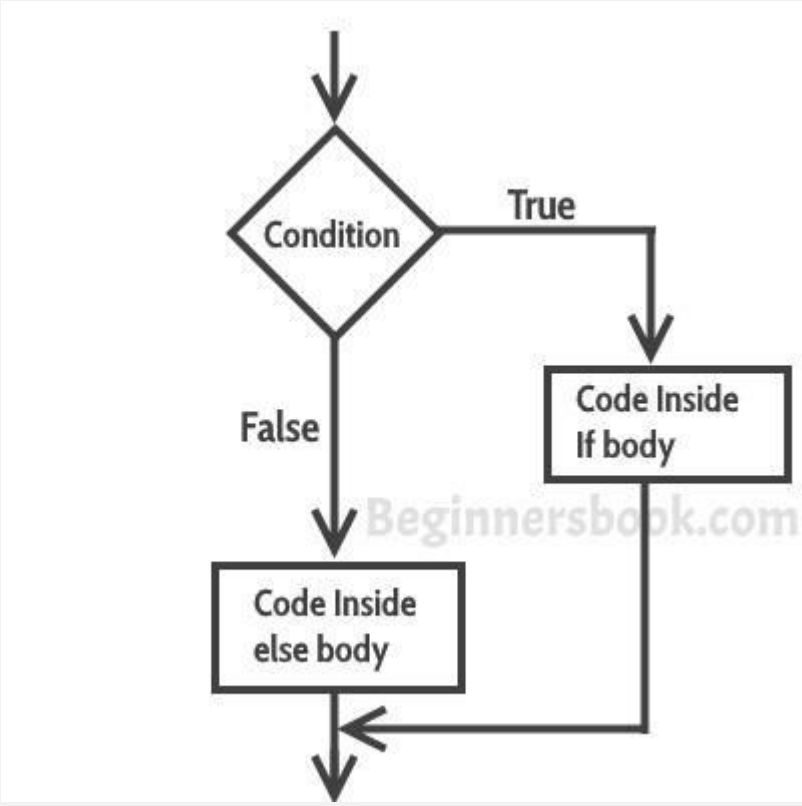
# If else statement in Java

This is how an if-else statement looks:

```
if(condition) {
Statement(s);
} else
{
```

```
   Statement(s);
}
```

The statements inside “if” would execute if the condition is true, and the statements inside “else” would execute if the condition is false.




## Example of if-else statement

```
public class IfElseExample {

   public static void main(String args[]){
int num=120;      if( num < 50 ){
      System.out.println("num is less than 50");
    }
else {
      System.out.println("num is greater than or equal 50");
    }
```

```
   }
}
```

**Output:**

```
num is greater than or equal 50
```

## if-else-if Statement

if-else-if statement is used when we need to check multiple conditions. In this statement we have only one "if" and one "else", however we can have multiple "else if". It is also known as **if else if ladder**. This is how it looks:

```
if(condition_1) {
   /*if condition_1 is true execute this*/
   statement(s);
}
else if(condition_2) {
   /* execute this if condition_1 is not met and
* condition_2 is met
    */
   statement(s);
}
else if(condition_3) {
   /* execute this if condition_1 & condition_2 are
* not met and condition_3 is met
    */
   statement(s);
} .
. .
else {
   /* if none of the condition is true
* then these statements gets executed
    */
   statement(s);
}
```

**Note:** The most important point to note here is that in if-else-if statement, as soon as the condition is met, the corresponding set of statements get executed, rest gets ignored. If none of the condition is met then the statements inside "else" gets executed.

## Example of if-else-if

```
public class IfElseIfExample {

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

```
      int num=1234;       if(num <100 && num>=1) {
        System.out.println("Its a two digit number");
      }
      else if(num <1000 && num>=100) {
        System.out.println("Its a three digit number");
      }
      else if(num <10000 && num>=1000) {
        System.out.println("Its a four digit number");
      }
      else if(num <100000 && num>=10000) {
          System.out.println("Its a five digit number");
        }
        else {
          System.out.println("number is not between 1 & 99999");
      }
  } }
```

**Output:**

```
Its a four digit number
```

---

# Switch Case statement in Java with example

**Switch case statement** is used when we have number of options (or choices) and we may need to perform a different task for each choice. The syntax of Switch case statement looks like this –

```
switch (variable or an integer expression)
{      case
constant:
     //Java code
     ;      case
constant:
     //Java code
     ;
default:
//Java code
     ;
}
```

Switch Case statement is mostly used with break statement even though it is optional. We will first see an example without break statement and then we will discuss switch case with break

## A Simple Switch Case Example

```
public class SwitchCaseExample1 {


   public static void main(String args[]){
int num=2;      switch(num+2)
     {
case 1:
        System.out.println("Case1: Value is: "+num);
      case 2:
        System.out.println("Case2: Value is: "+num);
      case 3:
        System.out.println("Case3: Value is: "+num);
default:
```

```
        System.out.println("Default: Value is: "+num);
}
   }
}
```

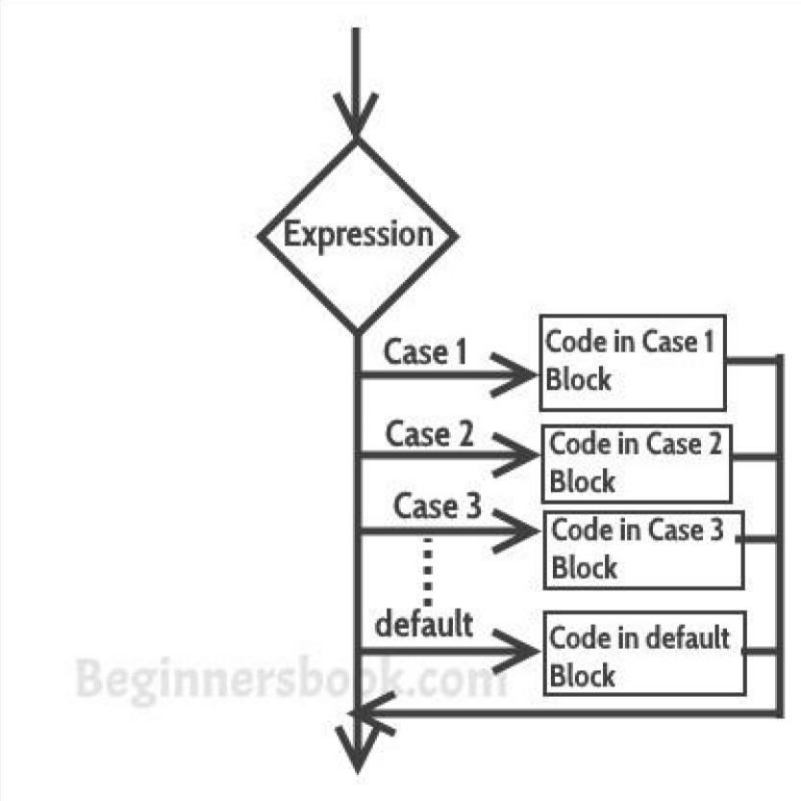
**Output:**

```
Default: Value is: 2
```

**Explanation:** In switch I gave an expression, you can give variable also. I gave num+2, where num value is 2 and after addition the expression resulted 4. Since there is no case defined with value 4 the default case got executed. This is why we should use default in switch case, so that if there is no catch that matches the condition, the default block gets executed.

## Switch Case Flow Diagram

First the variable, value or expression which is provided in the switch parenthesis is evaluated and then based on the result, the corresponding case block is executed that matches the result.




## Break statement in Switch Case

Break statement is optional in switch case but you would use it almost every time you deal with switch case. Before we discuss about break statement, Let's have a look at the example below where I am not using the break statement:

```
public class SwitchCaseExample2 {

   public static void main(String args[]){
      int i=2;
switch(i)       {
       case 1:
```

```
          System.out.println("Case1 ");
        case 2:
          System.out.println("Case2 ");
        case 3:
          System.out.println("Case3 ");
        case 4:
           System.out.println("Case4 ");
default:
          System.out.println("Default ");
      }
   }
}
```

**Output:**

```
Case2
Case3
Case4
Default
```

In the above program, we have passed integer value 2 to the switch, so the control switched to the case 2, however we don't have break statement after the case 2 that caused the flow to pass to the subsequent cases till the end. The solution to this problem is break statement

Break statements are used when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the execution flow would directly come out of the switch, ignoring rest of the cases

Let's take the same example but this time with break statement.

## Example with break statement

```
public class SwitchCaseExample2 {
    public static void main(String
args[]){       int i=2;       switch(i)
      {
        case 1:
          System.out.println("Case1 ");
```

```
            break;
case 2:
            System.out.println("Case2 ");
            break;
case 3:
            System.out.println("Case3 ");
            break;
case 4:
             System.out.println("Case4 ");
break;          default:
            System.out.println("Default ");
        }
    }
}
```

**Output:**

```
Case2
```

Now you can see that only case 2 had been executed, rest of the cases were ignored. **Why didn't I use break statement after**

**default?**

The control would itself come out of the switch after default so I didn't use it, however if you still want to use the break after default then you can use it, there is no harm in doing that.

# Few points about Switch Case

1) Case doesn't always need to have order 1, 2, 3 and so on. It can have any integer value after case keyword. Also, case doesn't need to be in an ascending order always, you can specify them in any order based on the requirement.

2) You can also use characters in switch case. for example –

```
public class SwitchCaseExample2 {

   public static void main(String args[]){
char ch='b';       switch(ch)
      {
        case 'd':
          System.out.println("Case1 ");
          break;     case 'b':
          System.out.println("Case2 ");
          break;
case 'x':
          System.out.println("Case3 ");
          break;
case 'y':
            System.out.println("Case4 ");
break;         default:
          System.out.println("Default ");
      }
   } }
```

3) The expression given inside switch should result in a constant value otherwise it would not be valid. For example:

**Valid expressions for switch:**

```
switch(1+2+23)
switch(1*2+3%4)
```

**Invalid switch expressions:**

```
switch(ab+cd)
switch(a+b+c)
```

4) Nesting of switch statements are allowed, which means you can have switch statements inside another switch. However nested switch statements should be avoided as it makes program more complex and less readable.

# For loop in Java with example

BY CHAITANYA SINGH | FILED UNDER: LEARN JAVA

Loops are used to execute a set of statements repeatedly until a particular condition is satisfied. In Java we have three types of basic loops: for, while and do-while. In this tutorial we will learn how to use "**for loop**" in Java.

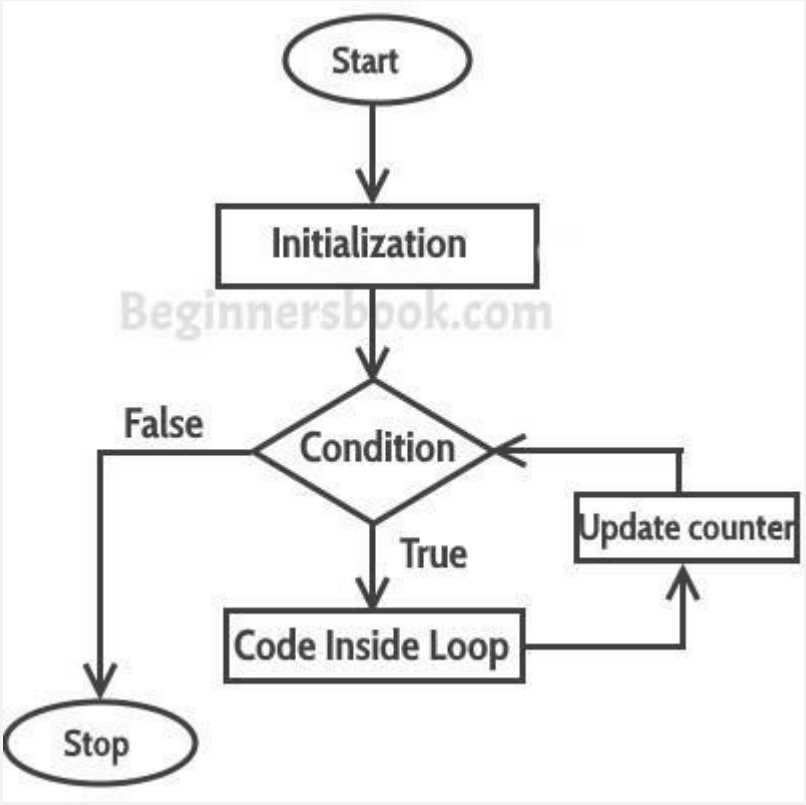
## Syntax of for loop:

```
for(initialization; condition ; increment/decrement)
{
   statement(s);
}
```

## Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the control flow, or the flow of execution of the

program.




**First step**: In for loop, initialization happens first and only one time, which means that the initialization part of for loop only executes once.

**Second step**: Condition in for loop is evaluated on each iteration, if the condition is true then the statements inside for loop body gets executed. Once the condition returns false, the statements in for loop does not execute and the control gets transferred to the next statement in the program after for loop.

**Third step**: After every execution of for loop's body, the increment/decrement part of for loop executes that updates the **loop counter**.

**Fourth step**: After third step, the control jumps to second step and condition is re-evaluated.

## Example of Simple For loop

```
class ForLoopExample {
    public static void main(String args[]){          for(int i=10; i>1; i--){
              System.out.println("The value of i is: "+i);
         }
    } }
```

The output of this program is:

```
The value of i is: 10
The value of i is: 9
The value of i is: 8
The value of i is: 7
The value of i is: 6
The value of i is: 5
The value of i is: 4
The value of i is: 3
The value of i is: 2
```

In the above program:
int i=1 is initialization expression i>1
is condition(Boolean expression) i–
Decrement operation

# Infinite for loop

The importance of Boolean expression and increment/decrement operation coordination:

```
class ForLoopExample2 {
    public static void main(String args[]){
         for(int i=1; i>=1; i++){
              System.out.println("The value of i is: "+i);
         }
    }
}
```

This is an infinite loop as the condition would never return false. The initialization

step is setting up the value of variable i to 1, since we are incrementing the value of i, it would always be greater than 1 (the Boolean expression: i>1) so it would never return false. This would eventually lead to the infinite loop condition. Thus it is important to see the co-ordination between Boolean expression and increment/decrement operation to determine whether the loop would terminate at some point of time or not.

Here is another example of infinite for loop:

```
// infinite loop for
( ; ; ) {     //
statement(s)
}
```

## For loop example to iterate an array:

Here we are iterating and displaying array elements using the for loop.

```
class ForLoopExample3 {
    public  static  void  main(String  args[]){
int arr[]={2,11,45,9};
         //i starts with 0 as array index starts with 0 too
         for(int i=0; i<arr.length; i++){
System.out.println(arr[i]);
         }
    }
}
```

Output:

```
2
11
45
9
```

## Enhanced For loop

Enhanced for loop is useful when you want to iterate Array/Collections, it is easy to write and understand.

Let's take the same example that we have written above and rewrite it using **enhanced for loop**.

```
class ForLoopExample3 {
   public static void main(String args[]){
int arr[]={2,11,45,9};       for (int num
: arr) {          System.out.println(num);
      }
   }
}
```

**Output:**

```
2
11
45
9
```

**Note:** In the above example, I have declared the num as int in the enhanced for loop. This will change depending on the data type of array. For example, the enhanced for loop for string type would look like this:

```
String arr[]={"hi","hello","bye"};
for (String str : arr) {
System.out.println(str); }
```

# While loop in Java with examples

In the last tutorial, we discussed for loop. In this tutorial we will discuss while loop. As discussed in previous tutorial, loops are used to execute a set of statements repeatedly until a particular condition is satisfied.

## Syntax of while loop

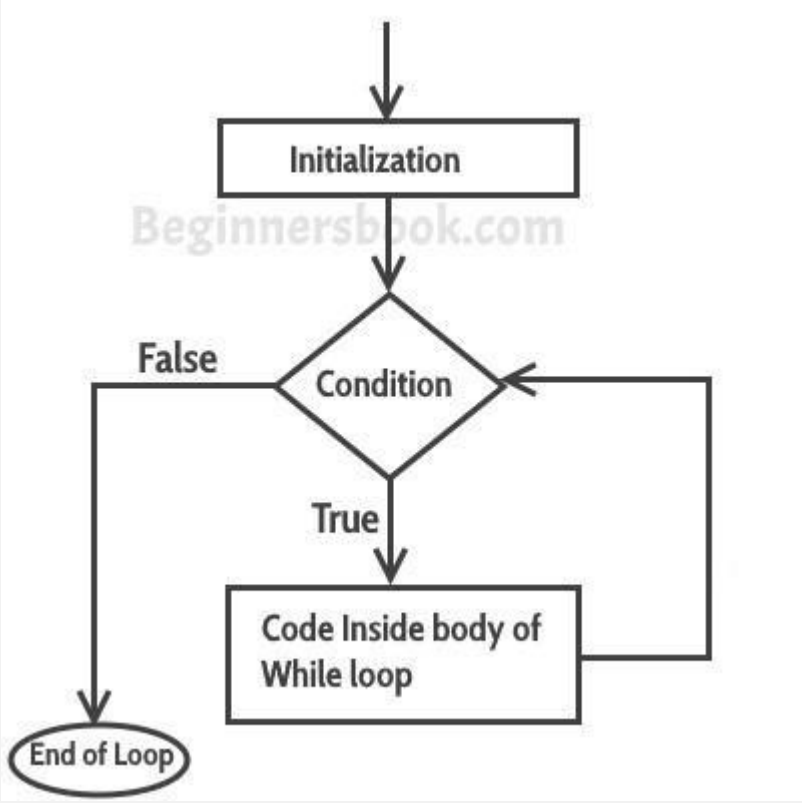
```
while(condition)
{
```

```
   statement(s);
}
```

## How while Loop works?

In while loop, condition is evaluated first and if it returns true then the statements inside while loop execute. When condition returns false, the control comes out of loop and jumps to the next statement after while loop.

Note: The important point to note when using while loop is that we need to use increment or decrement statement inside while loop so that the loop variable gets changed on each iteration, and at some point condition returns false. This way we can end the execution of while loop otherwise the loop would execute indefinitely.




## Simple while loop example

```
class WhileLoopExample {
    public static void main(String args[]){
int i=10;          while(i>1){
```

```
            System.out.println(i);
i--;
        }
    }
}
```

**Output:**

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

## Infinite while loop

```
class WhileLoopExample2 {
    public static void main(String args[]){
int i=10;          while(i>1)
        {
            System.out.println(i);
i++;
        }
    }
}
```

This loop would never end, its an infinite while loop. This is because condition is i>1 which would always be true as we are incrementing the value of i inside while loop.

Here is another example of infinite while loop:

```
while (true){
statement(s);
}
```

## Example: Iterating an array using while loop

Here we are iterating and displaying array elements using while loop.

```
class WhileLoopExample3 {
    public  static  void  main(String  args[]){
int arr[]={2,11,45,9};
         //i starts with 0 as array index starts with 0 too
int i=0;          while(i<4){
              System.out.println(arr[i]);
i++;
         }
    }
}
```

**Output:**

```
2
11
45
9
```

## do-while loop in Java with example

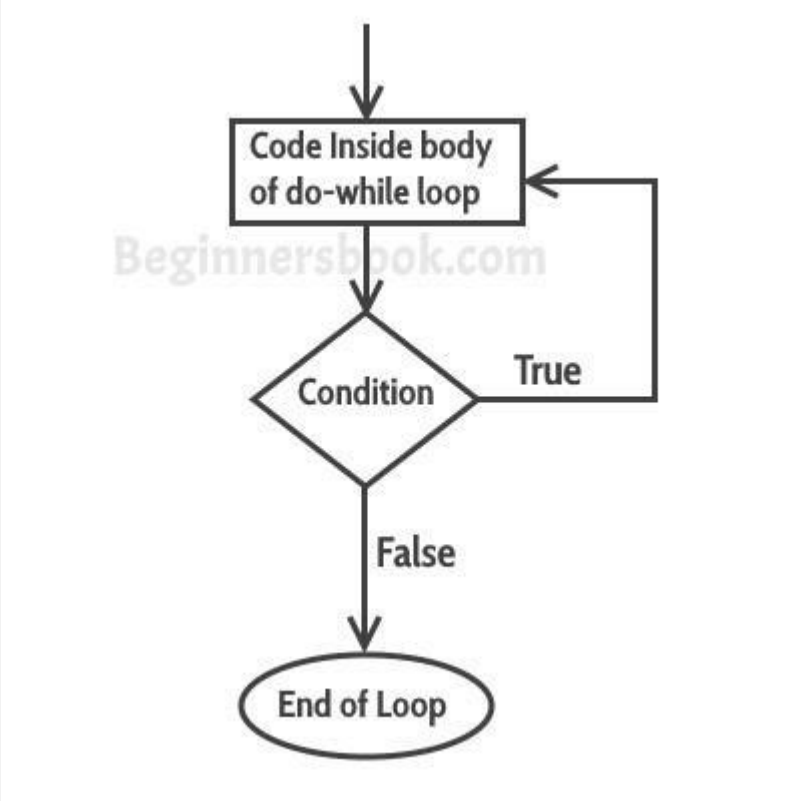
In the last tutorial, we discussed while loop. In this tutorial we will discuss dowhile loop in java. do-while loop is similar to while loop, however there is a difference between them: In while loop, condition is evaluated before the execution of loop's body but in do-while loop condition is evaluated after the execution of loop's body.

## Syntax of do-while loop:

```
do
{
   statement(s);  }
while(condition);
```

# How do-while loop works?

First, the statements inside loop execute and then the condition gets evaluated, if the condition returns true then the control gets transferred to the "do" else it jumps to the next statement after do-while.




# do-while loop example

```
class DoWhileLoopExample {
    public static void main(String args[]){
int i=10;
         do{
              System.out.println(i);
i--;

}while(
i>1);
    } }
```

**Output:**

```
10
9
8
7
6
```

```
5
4
3
2
```

## Example: Iterating array using do-while loop

Here we have an integer array and we are iterating the array and displaying each element using do-while loop.

```
class DoWhileLoopExample2 {
    public  static  void  main(String  args[]){
int arr[]={2,11,45,9};
         //i starts with 0 as array index starts with 0
int i=0;          do{
              System.out.println(arr[i]);
i++;          }while(i<4);
    }
}
```

Output:

2
11
45 9

# Continue Statement in Java with example

**Continue statement** is mostly used inside loops. Whenever it is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of statements inside loop's body for the current iteration. This is particularly useful when you want to continue the loop but do not want the rest of the statements(after continue statement) in loop body to execute for that particular iteration.

**Syntax:** continue word followed by semi colon.

```
continue;
```

## Example: continue statement inside for loop

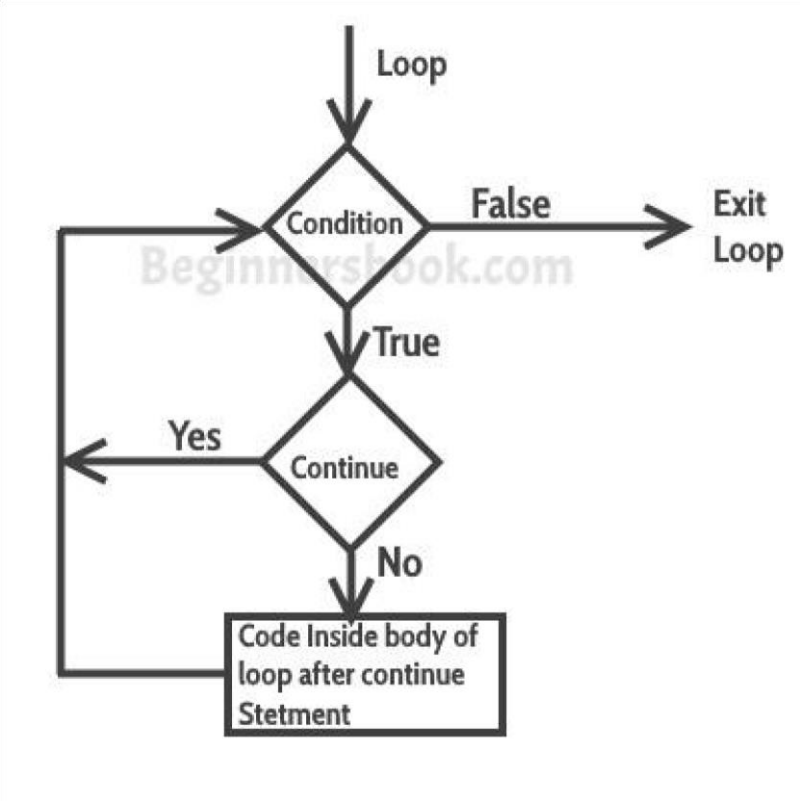
```
public class ContinueExample {

   public static void main(String args[]){  for
(int j=0; j<=6; j++)
      {
           if (j==4)
           {
             continue;
          }

           System.out.print(j+" ");
      }
   }
}
```

**Output:**

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

As you may have noticed, the value 4 is missing in the output, why? because when the value of variable j is 4, the program encountered a **continue statement**, which makes it to jump at the beginning of for loop for next iteration, skipping the statements for current iteration (that's the reason println didn't execute when the value of j was 4).

## Flow Diagram of Continue Statement




## Example: Use of continue in While loop

Same thing you can see here. We are iterating this loop from 10 to 0 for `counter` value and when the counter value is 7 the loop skipped the print statement and started next iteration of the while loop.

```
public class ContinueExample2 {

   public static void main(String args[]){
       int counter=10;
       while (counter >=0)
       {
            if (counter==7)
```

```
            {
               counter--;
continue;
            }
            System.out.print(counter+" ");
counter--;
       }
   }
}
```

**Output:**

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

## Example of continue in do-While loop

```
public class ContinueExample3 {

   public static void main(String args[]){
       int j=0;
         do
       {
          if (j==7)
          {
               j++;
               continue;
          }
          System.out.print(j+ " ");
          j++;
        }while(j<10);

   }
}
```

**Output:**

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

# Break statement in Java with example

The **break statement** is usually used in following two scenarios:

a) Use break statement to come out of the loop instantly. Whenever a break statement is encountered inside a loop, the control directly comes out of loop and
the loop gets terminated for rest of the iterations. It is used along with if statement, whenever used inside loop so that the loop gets terminated for a particular condition.

The important point to note here is that when a break statement is used inside a nested loop, then only the inner loop gets terminated.

b) It is also used in switch case control. Generally all cases in switch case are followed by a break statement so that whenever the program control jumps to a case, it doesn't execute subsequent cases (see the example below). As soon as a break is encountered in switch-case block, the control comes out of the switchcase body.

**Syntax of break statement:**

"break" word followed by semi colon

```
break;
```

## Example – Use of break in a while loop

In the example below, we have a while loop running from o to 100 but since we have a break statement that only occurs when the loop value reaches 2, the loop gets terminated and the control gets passed to the next statement in program after the loop body.

```
public class BreakExample1 {
   public static void main(String args[]){
int num =0;       while(num<=100)
      {
          System.out.println("Value of variable is: "+num);
if (num==2)
          {
             break;
```

```
          }
num++;
      }
      System.out.println("Out of while-loop");
  }
}
```

**Output:**

```
Value of variable is: 0
Value of variable is: 1
Value of variable is: 2
Out of while-loop
```

## Example – Use of break in a for loop

The same thing you can see here. As soon as the `var` value hits 99, the for loop gets terminated.

```
public class BreakExample2 {

   public static void main(String args[]){
        int var;
        for (var =100; var>=10; var --)
```

```
        {
            System.out.println("var: "+var);
            if (var==99)
            {
                 break;
            }
         }
         System.out.println("Out of for-loop");
    }
}
```

**Output:**

```
var: 100 var:
99
Out of for-loop
```

## Example – Use of break statement in switch-case

```
public class BreakExample3 {

   public static void main(String args[]){
       int num=2;

       switch (num)
       {
           case 1:
              System.out.println("Case 1 ");
              break;
case 2:
              System.out.println("Case 2 ");
              break;
case 3:
              System.out.println("Case 3 ");
              break;
default:
              System.out.println("Default ");
       }
   }
}
```

**Output:**

```
Case 2
```

In this example, we have break statement after each Case block, this is because

if we don't have it then the subsequent case block would also execute. The output of the same program without break would be `Case 2 Case 3 Default`.

# CHAPTER-6

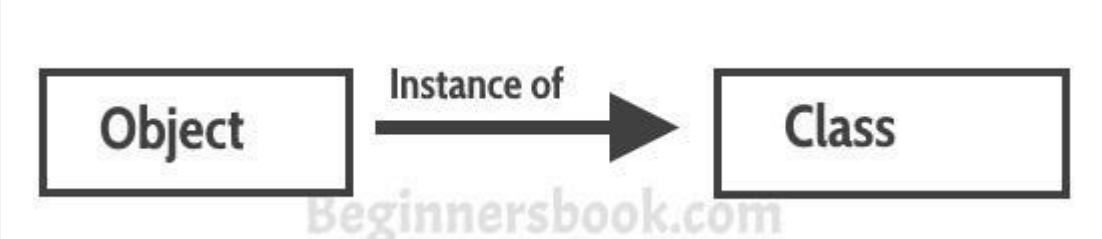
# OOPs concepts in Java

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of "objects" that contain data and methods. The primary purpose of object-oriented programming is to increase the flexibility and maintainability of programs. Object oriented programming brings together data and its behaviour(methods) in a single location(object) makes it easier to understand how a program works. We will cover each and every feature of OOPs in detail so that you won't face any difficultly understanding **OOPs Concepts**.

## OOPs Concepts – Table of Contents



## What is an Object




**Object:** is a bundle of data and its behaviour(often known as methods).

Objects have two characteristics: They have **states** and **behaviors**.

**Examples of states and behaviors Example 1:**
**Object**: House
**State**: Address, Color, Area
**Behavior**: Open door, close door

So if I had to write a class based on states and behaviours of House. I can do it like this: States can be represented as instance variables and behaviours as methods of the class. We will see how to create classes in the next section of this guide.

```
class House {    String
address;    String
color;    double are;
void openDoor() {
//Write code here
   }
   void closeDoor() {
//Write code here
   }
...
 ...
}
```

**Example 2:**

Let's take another example.
**Object**: Car
**State**: Color, Brand, Weight, Model

**Behavior**: Break, Accelerate, Slow Down, Gear change.

**Note:** As we have seen above, the states and behaviors of an object, can be represented by variables and methods in the class respectively.

## Characteristics of Objects:

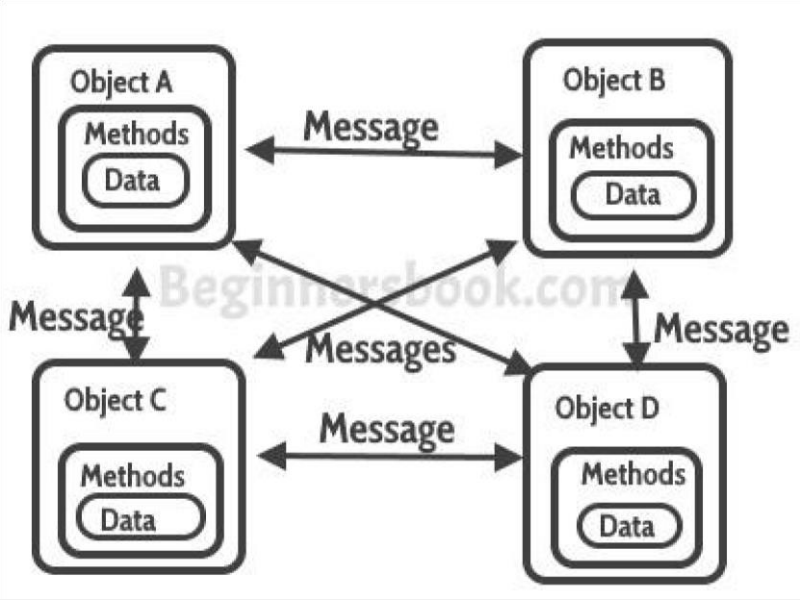
If you find it hard to understand Abstraction and Encapsulation, do not worry as I have covered these topics in detail with examples in the next section of this guide.

1. Abstraction
2. Encapsulation
3. Message passing

**Abstraction**: Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

**Encapsulation**: Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.

**Message passing**
A single object by itself may not be very useful. An application contains many objects. One object interacts with another object by invoking methods on that object. It is also referred to as **Method Invocation**. See the diagram below.




# What is a Class in OOPs Concepts

A class can be considered as a blueprint using which you can create as many objects as you like. For example, here we have a class Website that has two data members (also known as fields, instance variables and object states). This is just a blueprint, it does not represent any website, however using this we can create Website objects (or instances) that represents the websites. We have created two objects, while creating objects we provided separate properties to the objects using constructor.

```
public class Website {
   //fields (or instance variable)
```

```
   String webName;
int webAge;

   // constructor
   Website(String name, int age){
this.webName = name;
      this.webAge = age;
   }
   public static void main(String args[]){
      //Creating objects
      Website obj1 = new Website("beginnersbook", 5);
Website obj2 = new Website("google", 18);
     //Accessing object data through reference
     System.out.println(obj1.webName+" "+obj1.webAge);
System.out.println(obj2.webName+" "+obj2.webAge);
   }
}
```

**Output:**

```
beginnersbook 5 google
18
```

# What is a Constructor

Constructor looks like a method but it is in fact not a method. It's name is same as class name and it does not return any value. You must have seen this statement in almost all the programs I have shared above:

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

If you look at the right side of this statement, we are calling the default constructor of class myClassto create a new object (or instance).

We can also have parameters in the constructor, such constructors are known as parametrized constructors.

## Example of constructor

```
public class ConstructorExample {
    int age;
String name;
```

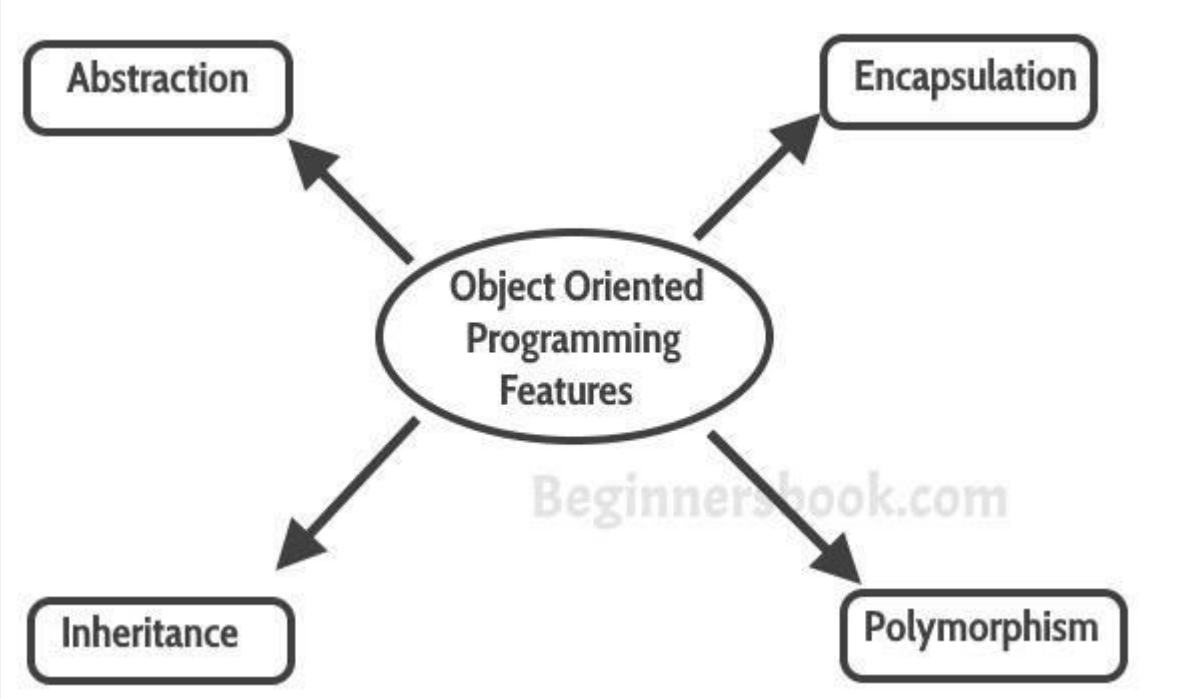
```
    //Default constructor
    ConstructorExample(){
        this.name="Chaitanya";
        this.age=30;
    }

    //Parameterized constructor
    ConstructorExample(String n,int a){
        this.name=n;        this.age=a;
    }
    public static void main(String args[]){

        ConstructorExample obj2 =
                    new ConstructorExample("Steve", 56);
        System.out.println(obj1.name+" "+obj1.age);
        System.out.println(obj2.name+" "+obj2.age);
}
}
```

**Output:**

```
Chaitanya 30 Steve
56
```

# Object Oriented Programming features

```
        ConstructorExample obj1 = new ConstructorExample();
```




These four features are the main OOPs Concepts that you must learn to understand the Object Oriented Programming in Java

## Abstraction

Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user. For example, when you login to your bank account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from the you. Read more about it here: Abstraction in Java.

# Encapsulation

Encapsulation simply means binding object state(fields) and behavior(methods) together. If you are creating class, you are doing encapsulation.

## Encapsulation example in Java

How to
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EmployeeCount
{
   private int numOfEmployees = 0;
   public void setNoOfEmployees (int count)
   {
       numOfEmployees = count;
   }
   public double getNoOfEmployees ()
   {
       return numOfEmployees;
   }
}
public class EncapsulationExample
{
   public static void main(String args[])
   {
      EmployeeCount obj = new EmployeeCount ();
      obj.setNoOfEmployees(5613);
      System.out.println("No Of Employees: "+(int)obj.getNoOfEmployees());
    }
}
```

**Output:**

```
No Of Employees: 5613
```

The class EncapsulationExample that is using the Object of class EmployeeCount will not

able to get the NoOfEmployees directly. It has to use the setter and getter methods of the same class to set and get the value.
**So what is the benefit of encapsulation in java programming**
Well, at some point of time, if you want to change the implementation details of the class EmployeeCount, you can freely do so without affecting the classes that are using it.

# CHAPTER-7

## Inheritance

The process by which one class acquires the properties and functionalities of another class is called inheritance. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
2. Inheritance allows us to reuse of code, it improves reusability in your java application.
3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

**Note:** The biggest advantage of Inheritance is that the code in base class need not be rewritten in the child class.
The **variables** and **methods** of the base class can be used in the **child class** as well.

## Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

```
class A extends B
{
}
```

## Inheritance Example

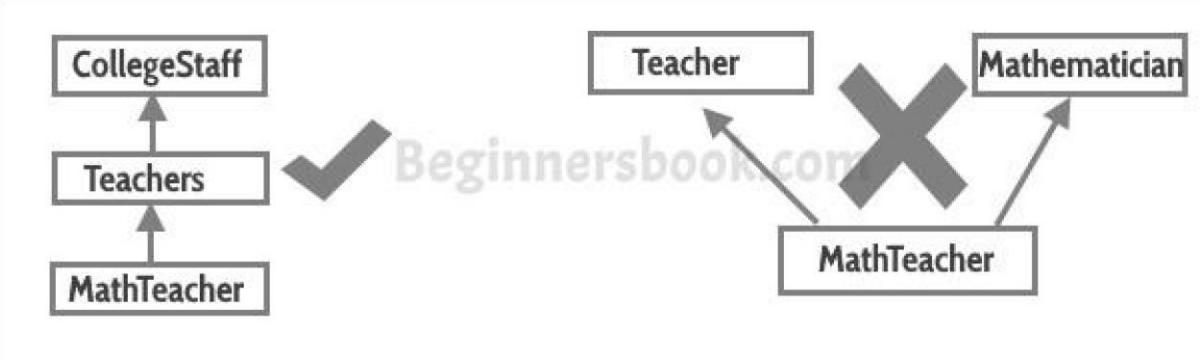
In this example, we have a parent class `Teacher` and a child class `MathTeacher`. In the `MathTeacher`class we need not to write the same code which is already present in the present class. Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher class does not need to write this code, the common data members and methods can inherited from the `Teacher` class.

```
class Teacher {
   String designation = "Teacher";
String college = "Beginnersbook";
void does(){
      System.out.println("Teaching");
}
}
public class MathTeacher extends Teacher{
String mainSubject = "Maths";    public
static void main(String args[]){
MathTeacher obj = new MathTeacher();
      System.out.println(obj.college);
      System.out.println(obj.designation);
System.out.println(obj.mainSubject);
obj.does();
   }
}
```

**Output:**

```
Beginnersbook
Teacher
Maths
Teaching
```

**Note: Multi-level inheritance** is allowed in Java but **not multiple inheritance**




Types of Inheritance:

**Single Inheritance**: refers to a child and parent class relationship where a class extends the another class.

**Multilevel inheritance**: refers to a child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C.

**Hierarchical inheritance**: refers to a child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A.

**Multiple Inheritance**: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. Java doesn't support multiple inheritance, read more about it here.

Most of the new **OO languages** like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

## Polymorphism

Polymorphism is a object oriented programming feature that allows us to perform a single action in different ways. For example, lets say we have a class `Animal` that has a method `animalSound()`, here we cannot give implementation to this method as we do not know which Animal class would extend `Animal` class. So, we make this method abstract like this:

```
public abstract class Animal{
   ...
   public abstract void animalSound();
}
```

Now suppose we have two `Animal` classes `Dog` and `Lion` that extends `Animal` class. We can provide the implementation detail there.

```
public class Lion extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Roar");
    }
}
```

and

```
public class Dog extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Woof");
    }
}
```

As you can see that although we had the common action for all subclasses `animalSound()` but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways).

## Types of Polymorphism

1) Static Polymorphism
2) Dynamic Polymorphism

## Static Polymorphism:

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading can be considered as static polymorphism example.

**Method Overloading**: This allows us to have more than one methods with same name in a class that differs in signature.

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
public class ExampleOverloading
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
obj.disp('a');        obj.disp('a',10);
   }
}
```

**Output:**

```
a a
10
```

When I say method signature I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

## Dynamic Polymorphism

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime rather, thats why it is called runtime polymorphism. **Example**

```
class Animal{
   public void animalSound(){
       System.out.println("Default Sound");
   }
}
public class Dog extends Animal{

   public void animalSound(){
       System.out.println("Woof");
   }
   public static void main(String args[]){
       Animal obj = new Dog();
       obj.animalSound();
   }
}
```

**Output:**

```
Woof
```

Since both the classes, child class and parent class have the same method animalSound. Which of the method will be called is determined at runtime by JVM.

**Few more overriding examples:**

```
Animal obj = new Animal(); obj.animalSound();
// This would call the Animal class method

Dog obj = new Dog(); obj.animalSound();
// This would call the Dog class method

Animal obj = new Dog(); obj.animalSound();
// This would call the Dog class method
```

## IS-A & HAS-A Relationships

A Car **IS-A** Vehicle and **HAS-A** License then the code would look like this:

```
public class Vehicle{ } public
class Car extends Vehicle{
private License myCarLicense; }
```

# Abstract Class and methods in OOPs Concepts

**Abstract method:**
1) A method that is declared but not defined. Only method signature no body.
2) Declared using the abstract keyword 3) Example :

```
abstract public void playInstrument();
```

5) Used to put some kind of compulsion on the class who inherits the class has abstract methods. The class that inherits must provide the implementation of all the abstract methods of parent class else declare the subclass as abstract. 6) These cannot be abstract

- Constructors
- Static methods
- Private methods
- Methods that are declared “final”

**Abstract Class**
An abstract class outlines the methods but not necessarily implements all the methods.

```
abstract class A{
   abstract    void    myMethod();
void anotherMethod(){
      //Does something
   }
}
```

**Note 1:** There can be some scenarios where it is difficult to implement all the methods in the base class. In such scenarios one can define the base class as an abstract class which signifies that this base class is a special kind of class which is not complete on its own.

A class derived from the abstract base class must implement those methods that are not implemented(means they are abstract) in the abstract class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of abstract class. To use this class, you need to create another class that extends this abstract class provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract parent class methods as well as implemented methods(those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of parent class(the abstract class), then the child class must need to be declared abstract.

## Example of Abstract class and Methods

Here we have an abstract class Animal that has an abstract method animalSound(), since the animal sound differs from one animal to another, there is no point in giving the implementation to this method as every child class must override this method to give its own implementation details. That's why we made it abstract.
Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

```
//abstract class abstract
class Animal{
//abstract method
   public abstract void animalSound();
}
public class Dog extends Animal{

   public void animalSound(){
       System.out.println("Woof");
   }
   public static void main(String args[]){
       Animal obj = new Dog();
       obj.animalSound();
   }
}
```

Output:

```
Woof
```

# Interfaces in Java

An interface is a blueprint of a class, which can be declared by using **interface** keyword. Interfaces can contain only constants and abstract methods (methods with only signatures no body).Like abstract classes, Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces. Interface is a common way to achieve full abstraction in Java.

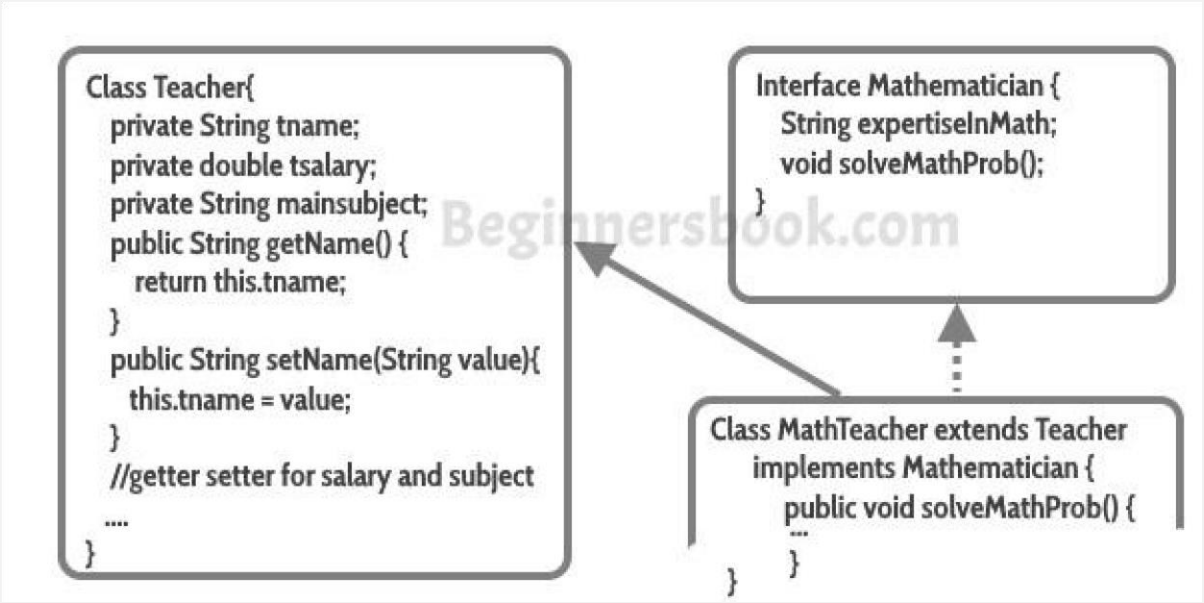
**Note:**

1. Java does not support Multiple Inheritance, however a class can implement more than one interfaces
2. Interface is similar to an abstract class but it contains only abstract methods.
3. Interfaces are created by using interface keyword instead of the keyword class
4. We use implements keyword while implementing an interface(similar to extending a class with extends keyword)

**Interface: Syntax**

```
class ClassName extends Superclass implements Interface1, Interface2, ....
```

## Example of Interface:



**Note**:

1. All **methods in an interface** are implicitly public and abstract. Using the keyword `abstract`before each method is optional.
2. An **interface** may contain final variables.
3. A class can **extend only one other class**, but it can **implement any number of interfaces.**
4. When a class implements an interface it has to give the definition of all the abstract methods of interface, else it can be declared as abstract class
5. An interface reference can point to **objects** of its implementing classes.

**Generalization and Specialization:**
In order to implement the concept of inheritance in an OOPs, one has to first identify the similarities among different classes so as to come up with the base class.

This process of identifying the similarities among different classes is called **Generalization**. Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes or methods.
In contrast to generalization, specialization means creating new subclasses from an existing class. If it turns out that certain attributes or methods only apply to some of the objects of the class, a subclass can be created.

## Access Specifiers

Well, you must have seen public, private keyword in the examples I have shared above. They are called access specifiers as they decide the scope of a data member, method or class.

There are **four types** of access specifiers in java: **public:** Accessible to all. Other objects can also access this member variable or function.
**private:** Not accessible by other objects. Private members can be accessed only by the methods in the same class. **Object accessible only in class in which they are declared. protected:** The scope of a protected variable is within the class which declares it and in the class which inherits from the class (Scope is class and subclass). **Default:** Scope is Package Level. We do not need to

explicitly mention default as when we do not mention any access specifier it is considered as default.

## What will we learn in the next tutorials on OOPs Concepts

Although we have covered almost all the OOPs concepts here, but whatever we have learned in this guide is in brief, these topics are wide and there is so much scope to learn these topics in detail with the help of examples. Thats why I have covered each and every topic in detail along with examples and diagrams in the next tutorials.
**How can you read the next tutorials in a sequential manner?** There are couple of ways to do it – 1) Tutorial links are provided in the left sidebar, go though them in the given sequence.
2) Go to the main java tutorial page that has all the links to the tutorials in the sequential manner.
If you find any difficulty understanding these OOPs Concepts then drop a comment below and I will get back to you as soon as possible.

# Constructors in Java – A complete study!!

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it's not a method as it doesn't have a return type. In short constructor and method are different(More on this at the end of this guide). People often refer constructor as special type of method in Java.

Constructor has same name as the class and looks like this in a java code.

```
public class MyClass{
//This is the constructor
   MyClass(){
}    .. }
```

Note that the constructor name matches with the class name and it doesn't have a return type.

# How does a constructor work

To understand the working of constructor, lets take an example. lets say we have a class MyClass.
When we create the object of MyClass like this:

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

The **new keyword** here creates the object of class MyClass and invokes the constructor to initialize this newly created object.

You may get a little lost here as I have not shown you any initialization example, lets have a look at the code below:

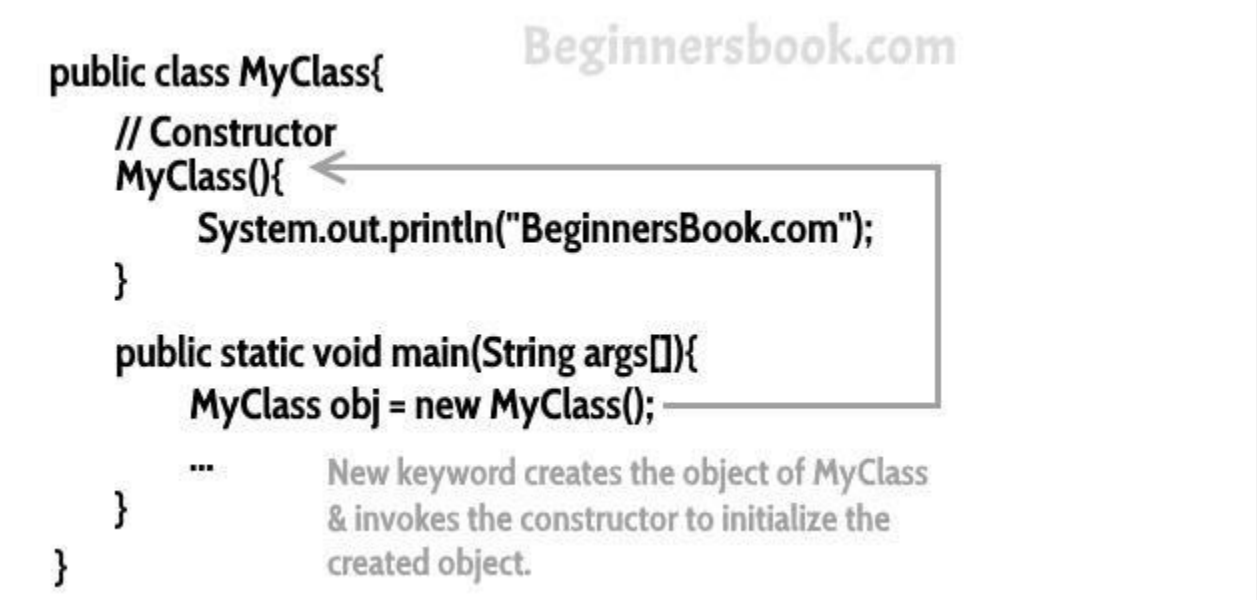
## A simple constructor program in java

Here we have created an object obj of class Hello and then we displayed the instance variable nameof the object. As you can see that the output is BeginnersBook.com which is what we have passed to the name during initialization in constructor. This shows that when we created the object obj the constructor got invoked. In this example we have used **this keyword**, which refers to the current object, object obj in this example. We will cover this keyword in detail in the next tutorial.

```
public class Hello {
   String name;
   //Constructor
Hello(){
      this.name = "BeginnersBook.com";
   }
   public static void main(String[] args) {
      Hello obj = new Hello();
      System.out.println(obj.name);
   }
}
```

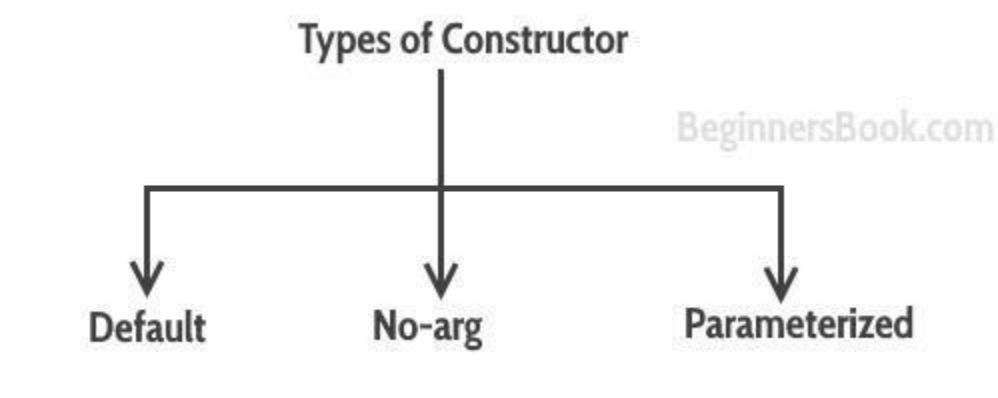
**Output:**

```
BeginnersBook.com
```



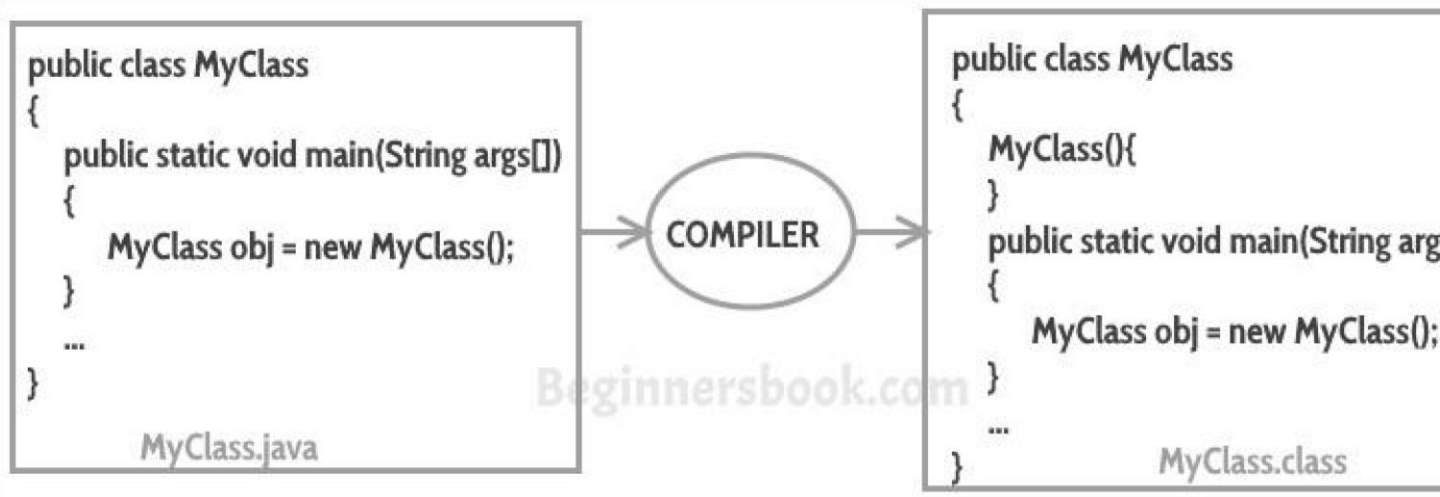

## Types of Constructors

There are three types of constructors: Default, No-arg constructor and Parameterized.




### Default constructor

If you do not implement any constructor in your class, Java compiler inserts a default constructorinto your code on your behalf. This constructor is known as default constructor. You would not find it in your source code(the java file) as it would be inserted into the code during compilation and exists in .class file. This process is shown in the diagram below:




If you implement any constructor then you no longer receive a default constructor from Java compiler.

## no-arg constructor:

Constructor with no arguments is known as **no-arg constructor**. The signature is same as default constructor, however body can have any code unlike default constructor where the body of the constructor is empty.

Although you may see some people claim that that default and no-arg constructor is same but in fact they are not, even if you write **public Demo() { }** in your class Demo it cannot be called default constructor since you have written the code of it.

### Example: no-arg constructor

```
class Demo
{
     public Demo()
     {
         System.out.println("This is a no argument constructor");
     }
     public static void main(String args[]) {
        new Demo();
     }
}
```

Output:

This is a no argument constructor

## Parameterized constructor

Constructor with arguments(or you can say parameters) is known as Parameterized constructor.

### Example: parameterized constructor

In this example we have a parameterized constructor with two parameters id and name. While creating the objects obj1 and obj2 I have passed two arguments so that this constructor gets invoked after creation of obj1 and obj2.

```
public class Employee {
    int empId;
String empName;

   //parameterized constructor with two parameters
   Employee(int id, String name){
this.empId = id;
       this.empName = name;
   }
   void info(){
        System.out.println("Id: "+empId+" Name: "+empName);
   }

   public static void main(String args[]){
       Employee obj1 = new Employee(10245,"Chaitanya");
       Employee obj2 = new Employee(92232,"Negan");
       obj1.info();        obj2.info();
   }
}
```

**Output:**

```
Id: 10245 Name: Chaitanya
Id: 92232 Name: Negan
```

## Example2: parameterized constructor

In this example, we have two constructors, a default constructor and a parameterized constructor. When we do not pass any parameter while creating the object using new keyword then default constructor is invoked, however when you pass a parameter then parameterized constructor that matches with the passed parameters list gets invoked.

```
class Example2
{
      private int var;
//default constructor
public Example2()
      {
            this.var = 10;
      }
      //parameterized constructor
public Example2(int num)
      {
            this.var = num;
```

```
	}
	public int getValue()
	{
		return var;
	}
	public static void main(String args[])
	{
		Example2 obj = new Example2();
		Example2 obj2 = new Example2(100);
		System.out.println("var is: "+obj.getValue());
		System.out.println("var is: "+obj2.getValue());
	}
}
```

**Output:**

```
var is: 10
var is: 100
```

## What if you implement only parameterized constructor in class

```
class Example3
{
	private int var;
	public Example3(int num)
	{
		var=num;
	}
	public int getValue()
	{
		return var;
	}
	public static void main(String args[])
	{
		Example3 myobj = new Example3();
		System.out.println("value of var is: "+myobj.getValue());
	}
}
```

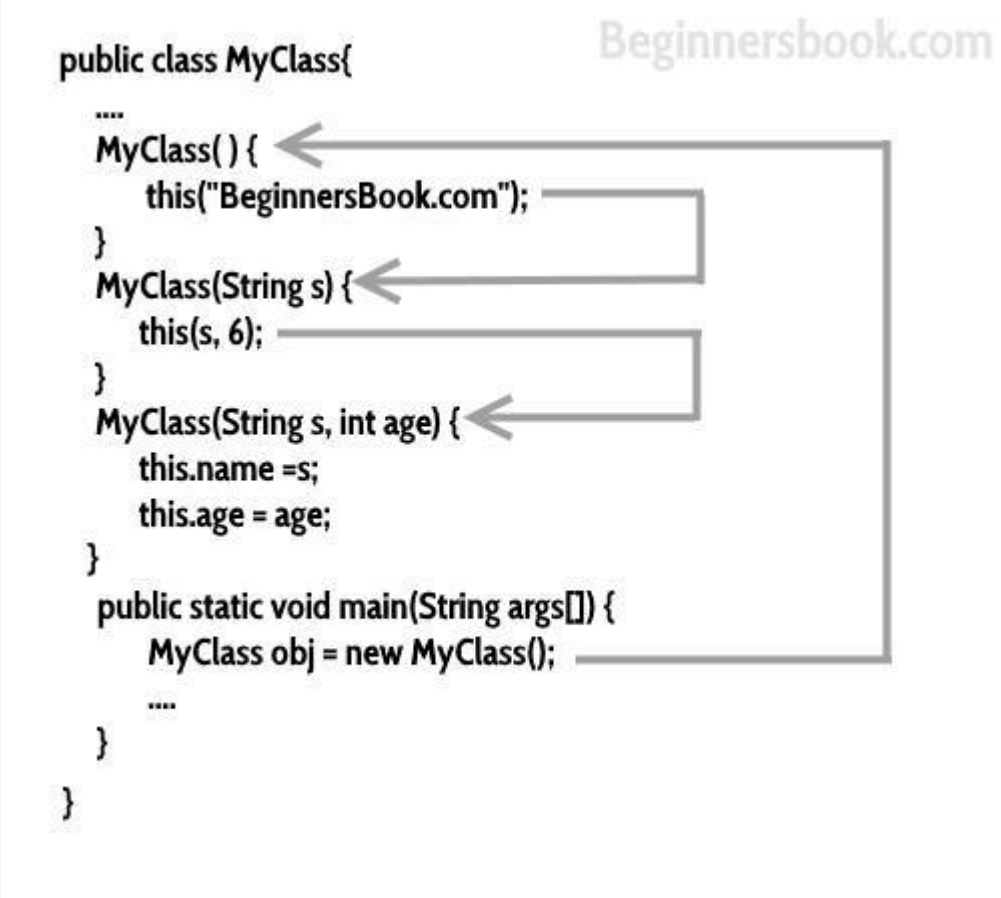
**Output**: It will throw a compilation error. The reason is, the statement `Example3 myobj = new Example3()` is invoking a default constructor which we don't have in our program. when you don't implement any constructor in your class, compiler inserts the default constructor into your code, however when you implement any constructor (in above example I have implemented parameterized constructor

with int parameter), then you don't receive the default constructor by compiler into your code.

If we remove the parameterized constructor from the above code then the program would run fine, because then compiler would insert the default constructor into your code.

## Constructor Chaining

When A constructor calls another constructor of same class then this is called constructor chaining. Read more about it here.




## Super()

Whenever a child class constructor gets invoked it implicitly invokes the constructor of parent class. You can also say that the compiler inserts a `super();` statement at the beginning of child class constructor.

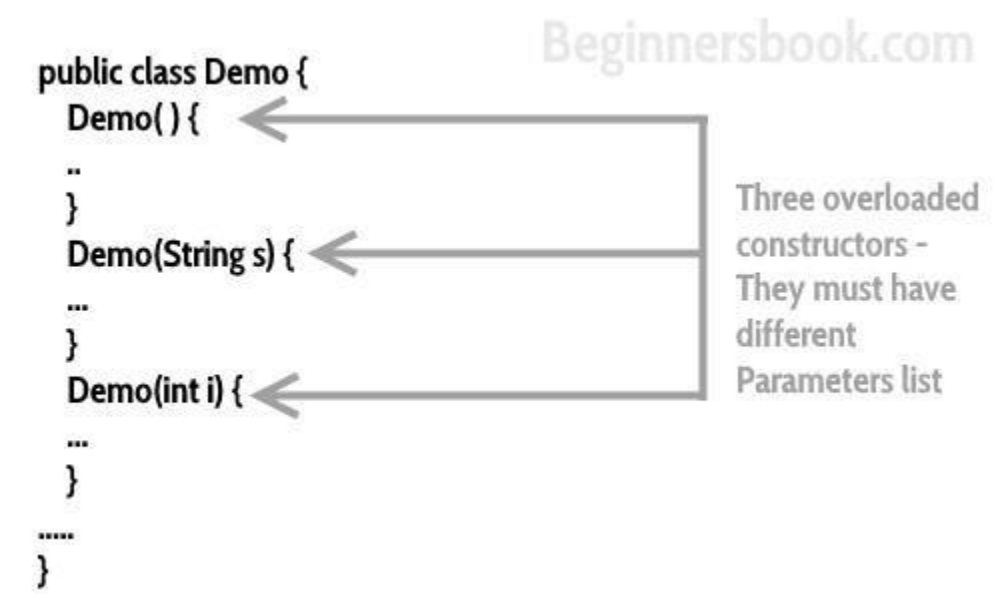
```
class MyParentClass {
   MyParentClass(){
       System.out.println("MyParentClass Constructor");
   }
}
class MyChildClass extends MyParentClass{
   MyChildClass() {
       System.out.println("MyChildClass Constructor");
   }
   public static void main(String args[]) {
       new MyChildClass();
   }
}
```

**Output:**

```
MyParentClass Constructor MyChildClass
Constructor
```

# Constructor Overloading

Constructor overloading is a concept of having more than one constructor with different parameters list, in such a way so that each constructor performs a different task.




## Java Copy Constructor

A copy constructor is used for copying the values of one object to another object.

```
class JavaExample{
   String web;
   JavaExample(String w){
       web = w;
   }

   /* This is the Copy Constructor, it
* copies the values of one object
* to the another object (the object
* that invokes this constructor)
    */
   JavaExample(JavaExample je){
       web = je.web;
   }
   void disp(){
       System.out.println("Website: "+web);
   }
   public static void main(String args[]){
       JavaExample obj1 = new JavaExample("BeginnersBook");

       /* Passing the object as an argument to the constructor
```

```
* This will invoke the copy constructor
        */
       JavaExample obj2 = new JavaExample(obj1);
       obj1.disp();
       obj2.disp();
    }
}
```

Output:

```
Website: BeginnersBook
Website: BeginnersBook
```

# Quick Recap

1. Every class has a constructor whether it's a normal class or a abstract class.
2. Constructors are not methods and they don't have any return type.
3. Constructor name should match with class name .
4. Constructor can use any access specifier, they can be declared as private also. Private constructors are possible in java but there scope is within the class only.
5. **Like constructors method can also have name same as class name, but still they have return type, though which we can identify them that they are methods not constructors.**
6. If you don't implement any constructor within the class, compiler will do it for.
7. **this() and super() should be the first statement in the constructor code.** If you don't mention them, compiler does it for you accordingly.
8. Constructor overloading is possible but overriding is not possible. Which means we can have overloaded constructor in our class but we can't override a constructor.
9. Constructors can not be inherited.
10. If Super class doesn't have a no-arg(default) constructor then compiler would not insert a default constructor in child class as it does in normal scenario.
11. Interfaces do not have constructors.
12. Abstract class can have constructor and it gets invoked when a class, which implements interface, is instantiated. (i.e. object creation of concrete class).
13. A constructor can also invoke another constructor of the same class – By using this(). If you want to invoke a parameterized constructor then do it like this: **this(parameter list)**.

## Difference between Constructor and Method

I know I should have mentioned it at the beginning of this guide but I wanted to cover everything in a flow. Hope you don't mind :)

1. The purpose of constructor is to initialize the object of a class while the purpose of a method is to perform a task by executing java code.
2. Constructors cannot be abstract, final, static and synchronised while methods can be.
3. Constructors do not have return types while methods do.

# Java – Static Class, Block, Methods and Variables

Static keyword can be used with class, variable, method and block. Static members belong to the class instead of a specific instance, this means if you make a member static, you can access it without object. Let's take an example to understand this:

Here we have a static method `myMethod()`, we can call this method without any object because when we make a member static it becomes class level. If we remove the static keyword and make it non-static then we must need to create an object of the class in order to call it.

Static members are common for all the instances(objects) of the class but nonstatic members are separate for each instance of class.

```
class SimpleStaticExample
{
    // This is a static method
static void myMethod()
    {
        System.out.println("myMethod");
    }
    public static void main(String[] args)
```

```
    {
          /* You can see that we are calling this
* method without creating any object.
           */
           myMethod();
    }
}
```

Output:

```
myMethod
```

## Static Block

Static block is used for initializing the static variables.This block gets executed when the class is loaded in the memory. A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

### Example 1: Single static block

As you can see that both the static variables were intialized before we accessed them in the main method.

```
class JavaExample{
static int num;
static String mystr;
static{       num = 97;
      mystr = "Static keyword in Java";
   }
   public static void main(String args[])
   {
      System.out.println("Value of num: "+num);
      System.out.println("Value of mystr: "+mystr);
   }
}
```

Output:

```
Value of num: 97
Value of mystr: Static keyword in Java
```

### Example 2: Multiple Static blocks

Lets see how multiple static blocks work in Java. They execute in the given order which means the first static block executes before second static block. That's the reason, values initialized by first block are overwritten by second block.

```
class JavaExample2{
static int num;
static String mystr;
//First Static block
   static{
      System.out.println("Static Block 1");
num = 68;
      mystr = "Block1";
}
  //Second static block
  static{
      System.out.println("Static Block 2");
num = 98;
      mystr = "Block2";
  }
  public static void main(String args[])
  {
      System.out.println("Value of num: "+num);
      System.out.println("Value of mystr: "+mystr);
   }
}
```

**Output:**

```
Static Block 1
Static Block 2
Value of num: 98
Value of mystr: Block2
```

# Java Static Variables

A static variable is common to all the instances (or objects) of the class because it is a class level variable. In other words you can say that only a single copy of static variable is created and shared among all the instances of the class. Memory allocation for such variables only happens once when the class is loaded in the memory.

Few Important Points:

- Static variables are also known as Class Variables.
- Unlike **non-static variables**, such variables can be accessed directly in static and non-static methods.

## Example 1: Static variables can be accessed directly in Static

## method

Here we have a static method `disp()` and two static variables `var1` and `var2`. Both the variables are accessed directly in the static method.

```
class JavaExample3{
  static int var1;
  static String var2;
  //This is a Static Method
  static void disp(){
      System.out.println("Var1 is: "+var1);
      System.out.println("Var2 is: "+var2);
  }
  public static void main(String args[])
  {
      disp();
  }
}
```

**Output:**

```
Var1 is: 0
Var2 is: null
```

## Example 2: Static variables are shared among all the instances of class

In this example, String variable is non-static and integer variable is Static. As you can see in the output that the non-static variable is different for both the objects but the static variable is shared among them, thats the reason the changes made to the static variable by object `ob2` reflects in both the objects.

```
class JavaExample{
   //Static integer variable
static int var1=77;
   //non-static string variable
   String var2;

   public static void main(String args[])
   {
       JavaExample ob1 = new JavaExample();
       JavaExample ob2 = new JavaExample();
       /* static variables can be accessed directly without
        * any instances. Just to demonstrate that static variables
```

```
* are shared, I am accessing them using objects so that
* we can check that the changes made to static variables
* by one object, reflects when we access them using other
* objects
       */
        //Assigning the value to static variable using object ob1
      ob1.var1=88;
      ob1.var2="I'm Object1";
       /* This will overwrite the value of var1 because var1 has a single
* copy shared among both the objects.
        */
ob2.var1=99;
      ob2.var2="I'm Object2";
      System.out.println("ob1 integer:"+ob1.var1);
      System.out.println("ob1 String:"+ob1.var2);
      System.out.println("ob2 integer:"+ob2.var1);
      System.out.println("ob2 STring:"+ob2.var2);
}
}
```

Output:

```
ob1 integer:99 ob1
String:I'm Object1 ob2
integer:99 ob2
STring:I'm Object2
```

# Java Static Methods

Static Methods can access class variables(static variables) without using object(instance) of the class, however non-static methods and non-static variables can only be accessed using objects.
Static methods can be accessed directly in static and non-static methods.
**Syntax:**
Static keyword followed by return type, followed by method name.

```
static return_type method_name();
```

## Example 1: static method main is accessing static variables without object

```
class JavaExample{
static int i = 10;
   static String s = "Beginnersbook";
```

```
    //This is a static method
    public static void main(String args[])
    {
        System.out.println("i:"+i);
        System.out.println("s:"+s);
```

```
   }
}
```

**Output:**

```
i:10
s:Beginnersbook
```

## Example 2: Static method accessed directly in static and non-static method

```
class JavaExample{
static int i = 100;
  static String s = "Beginnersbook";
  //Static method
static void display()
  {
     System.out.println("i:"+i);
     System.out.println("i:"+s);
  }

  //non-static method
void funcn()
  {
      //Static method called in non-static method
display();
  }
  //static method
  public static void main(String args[])
  {
         JavaExample obj = new JavaExample();
         //You need to have object to call this non-static method
obj.funcn();

      //Static method called in another static method
display();
   }
}
```

Output:

```
i:100
i:Beginnersbook
i:100
i:Beginnersbook
```

# Static Class

A class can be made **static** only if it is a nested class.

1. Nested static class doesn't need reference of Outer class
2. A static class cannot access non-static members of the Outer class

We will see these two points with the help of an example:

## Static class Example

```
class JavaExample{
   private static String str = "BeginnersBook";

   //Static class
   static class MyNestedClass{
       //non-static method
       public void disp() {

          /* If you make the str variable of outer class
*      non-static then you will get compilation error          * because: a nested
static class cannot access non-            * static members of the outer class.
           */
          System.out.println(str);
       }

}
   public static void main(String args[])
   {
        /* To create instance of nested class we didn't need the outer
*      class instance but for a regular nested class you would need
*      to create an instance of outer class first
         */
       JavaExample.MyNestedClass obj = new JavaExample.MyNestedClass();
       obj.disp();
   }
}
```

Output:

```
BeginnersBook
```

# Inheritance in Java Programming with examples

The process by which one class acquires the properties(data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.
**Child Class:**
The class that extends the features of another class is known as child class, sub class or derived class.

**Parent Class:**
The class whose properties and functionalities are used(inherited) by another class is known as parent class, super class or Base class.

Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
Inheritance allows us to reuse of code, it improves reusability in your java application.
Note: The biggest **advantage of Inheritance** is that the code that is already present in base class need not be rewritten in the child class.

This means that the data members(instance variables) and methods of the parent class can be used in the child class as.

## Syntax: Inheritance in Java

To inherit a class we use extends keyword. Here class XYZ is child class and class ABC is parent class. The class XYZ is inheriting the properties and methods of ABC class.

```
class XYZ extends ABC
{
}
```

# Inheritance Example

In this example, we have a base class Teacher and a sub class PhysicsTeacher. Since class PhysicsTeacher extends the designation and college properties and work() method from base class, we need not to declare these properties and method in sub class.
Here we have collegeName, designation and work() method which are common

to all the teachers so we have declared them in the base class, this way the child classes like MathTeacher, MusicTeacher and PhysicsTeacher do not need to write this code and can be used directly from base class.

```
class Teacher {
   String designation = "Teacher";
   String collegeName = "Beginnersbook";
   void does(){
         System.out.println("Teaching");
   }
}

public class PhysicsTeacher extends Teacher{
   String mainSubject = "Physics";
   public static void main(String args[]){
         PhysicsTeacher obj = new PhysicsTeacher();
         System.out.println(obj.collegeName);
         System.out.println(obj.designation);
         System.out.println(obj.mainSubject);
         obj.does();
   }
}
```

Output:

```
Beginnersbook
Teacher
Physics
Teaching
```

Based on the above example we can say that PhysicsTeacher **IS-A** Teacher. This means that a child class has IS-A relationship with the parent class. This is inheritance is known as **IS-A relationship** between child and parent class

**Note:**
The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as private then the derived class cannot use them directly. The private members can be accessed only in its own class. Such private members can only be accessed using public or protected getter and setter methods of super class as shown in the example below.

```
class Teacher {
```

```
   private String designation = "Teacher";
private String collegeName = "Beginnersbook";
public String getDesignation() {        return
designation;
   }
   protected void setDesignation(String designation) {
       this.designation = designation;
   }
   protected String getCollegeName() {
       return collegeName;
   }
   protected void setCollegeName(String collegeName) {
       this.collegeName = collegeName;
   }
   void does(){
       System.out.println("Teaching");
   }
}
public class JavaExample extends Teacher{
String mainSubject = "Physics";    public
static void main(String args[]){
      JavaExample obj = new JavaExample();
      /* Note: we are not accessing the data members
* directly we are using public getter method
* to access the private members of parent class
       */
      System.out.println(obj.getCollegeName());
      System.out.println(obj.getDesignation());
      System.out.println(obj.mainSubject);
      obj.does();
   }
}
```

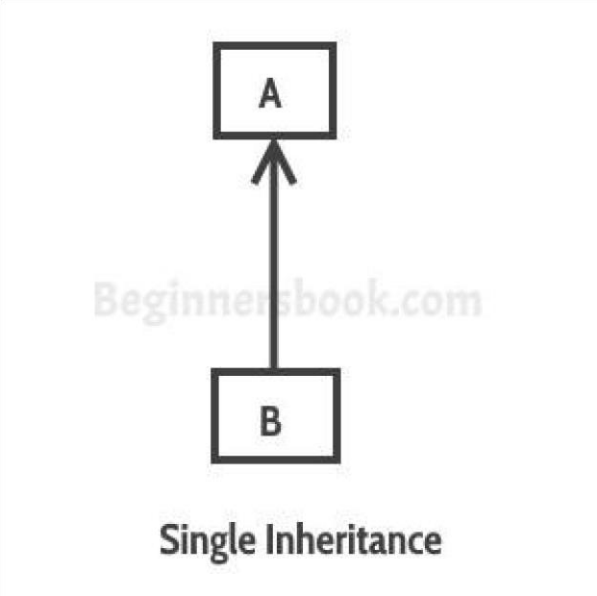
The output is:

```
Beginnersbook
Teacher
Physics
Teaching
```

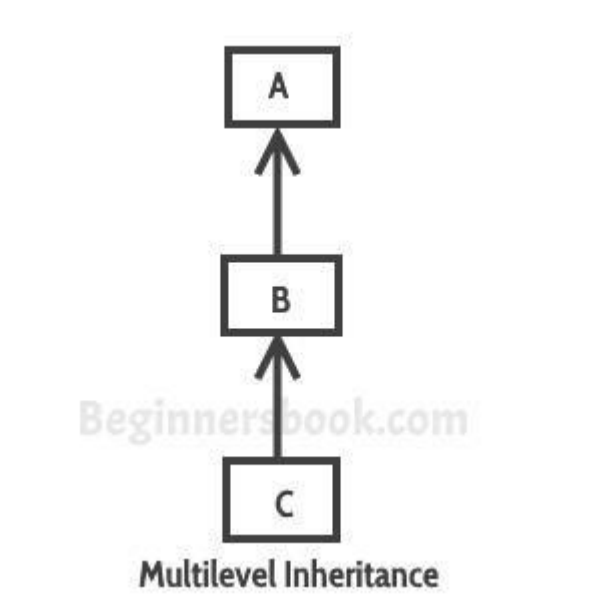
The important point to note in the above example is that the child class is able to access the private members of parent class through **protected methods** of parent class. When we make a instance variable(data member) or method **protected**, this means that they are accessible only in the class itself and in child class. These public, protected, private etc. are all access specifiers and we will discuss them in the coming tutorials.

# Types of inheritance

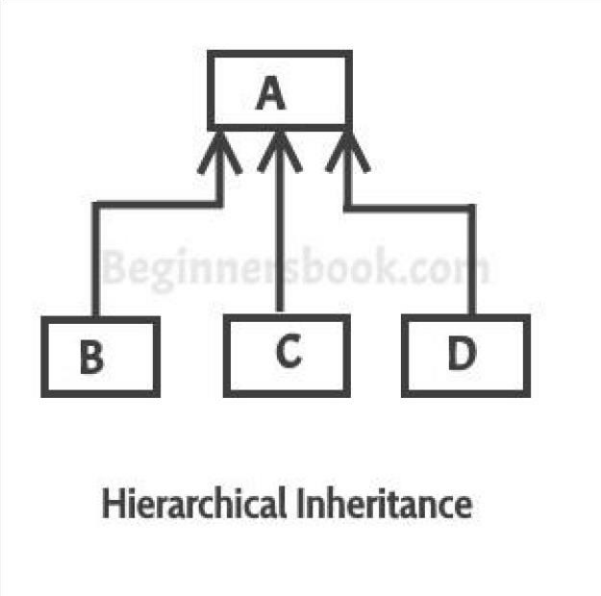
To learn types of inheritance in detail, refer: Types of Inheritance in Java.
**Single Inheritance**: refers to a child and parent class relationship where a class extends the another class.




Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class C extends class B and class B extends class A.

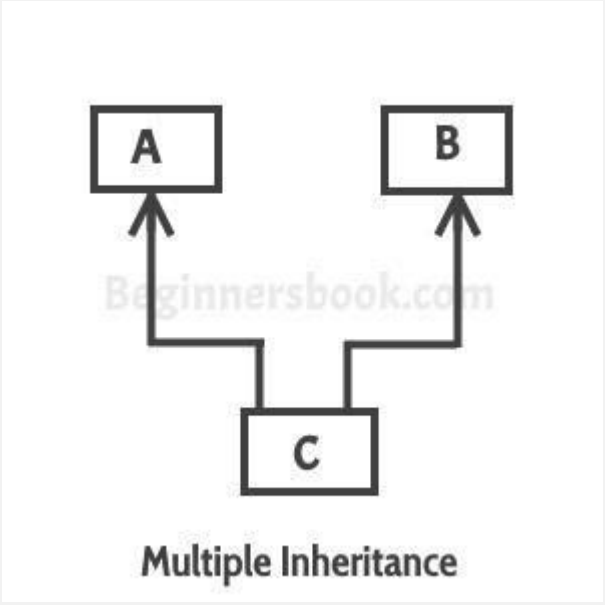



Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.




**Multiple Inheritance**: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. For example

class C extends both classes A and B. Java doesn't support multiple inheritance, read more about it here.




Hybrid inheritance: Combination of more than one types of inheritance in a single program. For example class A & B extends class C and another class D extends class A then this is a hybrid inheritance example because it is a combination of single and hierarchical inheritance.

# Constructors and Inheritance

constructor of sub class is invoked when we create the object of subclass, it by default invokes the default constructor of super class. Hence, in inheritance the objects are constructed top-down. The superclass constructor can be called explicitly using the super keyword, but it should be first statement in a constructor. The super keyword refers to the superclass, immediately above of the calling class in the hierarchy. The use of multiple super keywords to access an ancestor class other than the direct parent is not permitted.

```
class ParentClass{
   //Parent class constructor
   ParentClass(){
       System.out.println("Constructor of Parent");
} }
class JavaExample extends ParentClass{
   JavaExample(){
       /* It by default invokes the constructor of parent class
* You can use super() to call the constructor of parent.
* It should be the first statement in the child class
* constructor, you can also call the parameterized constructor
* of parent class by using super like this: super(10), now
* this will invoke the parameterized constructor of int arg
        */
       System.out.println("Constructor of Child");
   }    public static void main(String
args[]){     //Creating the object of child
class
       new JavaExample();
   }
}
```

Output:

```
Constructor of Parent Constructor
of Child
```

# CHAPTER-8

# Inheritance and Method Overriding

When we declare the same method in child class which is already present in the parent class the this is called method overriding. In this case when we call the method from child class object, the child class version of the method is called. However we can call the parent class method using super keyword as I have shown in the example below:

```
class ParentClass{
   //Parent class constructor
   ParentClass(){
       System.out.println("Constructor of Parent");
   }
   void disp(){
       System.out.println("Parent Method");
   }
}
class JavaExample extends ParentClass{
   JavaExample(){
       System.out.println("Constructor of Child");
   }
   void disp(){
       System.out.println("Child Method");
        //Calling the disp() method of parent class
       super.disp();
   }    public static void main(String
args[]){     //Creating the object of child
class
       JavaExample obj = new JavaExample();
       obj.disp();
   }
}
```

The output is :

```
Constructor of Parent
Constructor of Child
Child Method
Parent Method
```

# OOPs concepts – What is Aggregation in java?

Aggregation is a special form of association. It is a relationship between two classes like association, however its a **directional** association, which means it is strictly a **one way association.** It represents a **HAS-A** relationship.

## Aggregation Example in Java

For example consider two classes Student class and Address class. Every student has an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make any sense as an Address doesn't need to have a Student necessarily. Lets write this example in a java program. Student Has-A Address

```
class Address
{
   int streetNum;
String city;
   String state;
   String country;
   Address(int street, String c, String st, String coun)
   {
       this.streetNum=street;
this.city =c;
this.state = st;
this.country = coun;
   }
}
class StudentClass
{
   int rollNum;
String studentName;
   //Creating HAS-A relationship with Address class
   Address studentAddr;
   StudentClass(int roll, String name, Address addr){
this.rollNum=roll;        this.studentName=name;
this.studentAddr = addr;
   }
   public static void main(String args[]){
       Address ad = new Address(55, "Agra", "UP", "India");
       StudentClass obj = new StudentClass(123, "Chaitanya", ad);
       System.out.println(obj.rollNum);
       System.out.println(obj.studentName);
       System.out.println(obj.studentAddr.streetNum);
       System.out.println(obj.studentAddr.city);
       System.out.println(obj.studentAddr.state);
```

```
        System.out.println(obj.studentAddr.country);
    }
}
```

Output:

```
123
Chaitanya
55
Agra
UP
India
```

The above example shows the **Aggregation** between Student and Address classes. You can see that in Student class I have declared a property of type Address to obtain student address. Its a typical example of Aggregation in Java.

# Why we need Aggregation?

**To maintain code re-usability**. To understand this lets take the same example again. Suppose there are two other classes College and Staff along with above two classes Student and Address. In order to maintain Student's address, College Address and Staff's address we don't need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:

```
Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address) Staff
Has-A Address (Has-a relationship between staff and address)
```

Hence we can improve code re-usability by using Aggregation relationship.

So if I have to write this in a program, I would do it like this:

```
class Address
{
   int streetNum;
String city;
   String state;
   String country;
   Address(int street, String c, String st, String coun)
   {
       this.streetNum=street;
this.city =c;
this.state = st;
this.country = coun;
   }
}
class StudentClass
{
   int rollNum;
String studentName;
   //Creating HAS-A relationship with Address class
   Address studentAddr;
   StudentClass(int roll, String name, Address addr){
this.rollNum=roll;        this.studentName=name;
this.studentAddr = addr;    }    ... }
class College
{
   String collegeName;
```

```
   //Creating HAS-A relationship with Address class
   Address collegeAddr;
   College(String name, Address addr){
this.collegeName = name;
       this.collegeAddr = addr;
   }
... }
class Staff
{
   String employeeName;
   //Creating HAS-A relationship with Address class
   Address employeeAddr;
   Staff(String name, Address addr){
this.employeeName = name;
this.employeeAddr = addr;    }
... }
```

As you can see that we didn't write the Address code in any of the three classes, we simply created the HAS-A relationship with the Address class to use the Address code. The dot dot(…) part in the above code can be replaced with the public static void main method, the code in it would be similar to what we have seen in the first example.

# OOPs concepts – What is Association in java?

In this article we will discuss **Association in Java**. Association establishes relationship between two separate **classes** through their **objects**. The relationship can be one to one, One to many, many to one and many to many.

## Association Example

```
class CarClass{
String carName;
int carId;
   CarClass(String name, int id)
   {
       this.carName = name;
       this.carId = id;
```

```
    }
}
class Driver extends CarClass{
    String driverName;
    Driver(String name, String cname, int cid){
        super(cname, cid);
        this.driverName=name;
    }
}
class TransportCompany{
    public static void main(String args[])
    {
        Driver obj = new Driver("Andy", "Ford", 9988);
        System.out.println(obj.driverName+" is a driver of car Id: "+obj.carId);
    } }
```

Output:

```
Andy is a driver of car Id: 9988
```

In the above example, there is a one to one relationship(**Association**) between two classes: `CarClass`and `Driver`. Both the classes represent two separate entities.

## Association vs Aggregation vs Composition

Lets discuss **difference between Association, Aggregation and Composition**:

Although all three are related terms, there are some major differences in the way they relate two classes. **Association** is a relationship between two separate classes and the association can be of any type say one to one, one to may etc. It joins two entirely separate entities.

Aggregation is a special form of association which is a unidirectional one way relationship between classes (or entities), for e.g. Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so its a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if Wallet class is not present, it does not mean that the Money class cannot exist.

**Composition** is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A

human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.

# Super keyword in java with example

The super keyword refers to the objects of immediate parent class.

## The use of super keyword

1) To access the data members of parent class when both parent and child class have member with same name
2) To explicitly call the no-arg and parameterized constructor of parent class
3) To access the method of parent class when child class has overridden that method.

**Now lets discuss them in detail with the help of examples**:

### 1) How to use super keyword to access the variables of parent class

When you have a variable in child class which is already present in the parent class then in order to access the variable of parent class, you need to use the super keyword.

Lets take an example to understand this: In the following program, we have a data member `num`declared in the child class, the member with the same name is already present in the parent class. There is no way you can access the `num` variable of parent class without using super keyword. .

```
//Parent class or Superclass or base class class
Superclass
```

```
{
   int num = 100;
}
//Child class or subclass or derived class class
Subclass extends Superclass
{
   /* The same variable num is declared in the Subclass
    * which is already present in the Superclass
    */
    int num = 110;
    void printNumber(){
       System.out.println(num);
    }
    public static void main(String args[]){
       Subclass obj= new Subclass();
       obj.printNumber();
    }
}
```

Output: 110

**Accessing the num variable of parent class:**

By calling a variable like this, we can access the variable of parent class if both the classes (parent and child) have same variable. `super.variable_name`

Let's take the same example that we have seen above, this time in print statement we are passing `super.num` instead of num.

```
class Superclass
{
   int num = 100;
}
class Subclass extends Superclass
{
   int num = 110;
   void printNumber(){
       /* Note that instead of writing num we are
* writing super.num in the print statement
* this refers to the num variable of Superclass
        */
       System.out.println(super.num);
```

```
   }    public static void main(String
args[]){     Subclass obj= new Subclass();
       obj.printNumber();
   }
}
```

Output:

100

As you can see by using super.num we accessed the num variable of parent class.

## 2) Use of super keyword to invoke constructor of parent class

When we create the object of sub class, the new keyword invokes the constructor of child class, which implicitly invokes the constructor of parent class. So the order to execution when we create the object of child class is: parent class constructor is executed first and then the child class constructor is executed. It happens because compiler itself adds super()(this invokes the noarg constructor of parent class) as the first statement in the constructor of child class.

Let's see an example to understand what I have explained above:

```
class Parentclass
{
   Parentclass(){
       System.out.println("Constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
       /* Compile implicitly adds super() here as the
        *  first statement of this constructor.
        */
       System.out.println("Constructor of child class");
   }
   Subclass(int num){
       /* Even though it is a parameterized constructor.
*      The compiler still adds the no-arg super() here
        */
       System.out.println("arg constructor of child class");
   }
   void display(){
       System.out.println("Hello!");
   }
   public static void main(String args[]){
       /* Creating object using default constructor. This
*      will invoke child class constructor, which will
*      invoke parent class constructor
        */
       Subclass obj= new Subclass();
       //Calling sub class method
       obj.display();
       /* Creating second object using arg constructor
*      it will invoke arg constructor of child class which will    * invoke no-arg
constructor of parent class automatically
        */
       Subclass obj2= new Subclass(10);
       obj2.display();
   }
}
```

**Output:**

```
Constructor of parent class Constructor
of child class
Hello!
Constructor of parent class arg
constructor of child class
Hello!
```

## Parameterized super() call to invoke parameterized constructor of parent class

We can call super() explicitly in the constructor of child class, but it would not make any sense because it would be redundant. It's like explicitly doing something which would be implicitly done otherwise.
However when we have a constructor in parent class that takes arguments then we can use parameterized super, like `super(100);` to invoke parameterized constructor of parent class from the constructor of child class. Let's see an example to understand this:

```
class Parentclass
{
   //no-arg constructor
   Parentclass(){
 System.out.println("no-arg constructor of parent class");    }
   //arg or parameterized constructor
   Parentclass(String str){
       System.out.println("parameterized constructor of parent class");
   }
}
class Subclass extends Parentclass
{
   Subclass(){
       /* super() must be added to the first statement of constructor
*      otherwise you will get a compilation error. Another important
*      point to note is that when we explicitly use super in constructor
       * the compiler doesn't invoke the parent constructor automatically.
       */
       super("Hahaha");
       System.out.println("Constructor of child class");

}
   void display(){
       System.out.println("Hello");
   }
```

```
   public static void main(String args[]){
       Subclass obj= new Subclass();
       obj.display();
   }
}
```

**Output:**

```
parameterized constructor of parent class
Constructor of child class Hello
```

There are few important points to note in this example:

1) super()(or parameterized super must be the first statement in constructor otherwise you will get the compilation error: “Constructor call must be the first statement in a constructor”
2) When we explicitly placed super in the constructor, the java compiler didn’t call the default no-arg constructor of parent class.

## 3) How to use super keyword in case of method overriding

When a child class declares a same method which is already present in the parent class then this is called method overriding. We will learn method overriding in the next tutorials of this series. For now you just need to remember this: When a child class overrides a method of parent class, then the call to the method from child class object always call the child class version of the method. However by using super keyword like this: super.method_name you can call the method of parent class (the method which is overridden). In case of method overriding, these terminologies are used: Overridden method: The method of parent class Overriding method: The method of child class Lets take an example to understand this concept:

```
class Parentclass
{
   //Overridden method
void display(){
       System.out.println("Parent class method");
   }
}
class Subclass extends Parentclass
{
   //Overriding method
void display(){
       System.out.println("Child class method");
```

```
    }
    void printMsg(){
        //This would call Overriding method
        display();
        //This would call Overridden method
        super.display();
    }
    public static void main(String args[]){
        Subclass obj= new Subclass();
        obj.printMsg();
    }
}
```

**Output:**

```
Child class method Parent
class method
```

## What if the child class is not overriding any method: No need of super

When child class doesn't override the parent class method then we don't need to use the super keyword to call the parent class method. This is because in this case we have only one version of each method and child class has access to the parent class methods so we can directly call the methods of parent class without using super.

```
class Parentclass
{
    void display(){
        System.out.println("Parent class method");
    }
}
class Subclass extends Parentclass
{
    void printMsg(){
        /* This would call method of parent class,
* no need to use super keyword because no other
* method with the same name is present in this class
         */
        display();
    }
    public static void main(String args[]){
```

```
      Subclass obj= new Subclass();
obj.printMsg();
   }
}
```

**Output:**

```
Parent class method
```

# Method Overloading in Java with examples

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.
let's get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

## Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.
For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.
For example:

```
add(int, float)
add(float, int)
```

**Invalid case of method overloading:**

When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

**Points to Note:**

1. Static Polymorphism is also known as compile time binding or early binding.
2. Static binding happens at compile time. Method overloading is an example of static binding where binding of method call to its definition happens at Compile time.

# Method Overloading examples

As discussed in the beginning of this guide, method overloading is done by declaring same method with different parameters. The parameters must be different in either of these: number, sequence or types of parameters (or arguments). Lets see examples of each of these cases.

Argument list is also known as parameter list

## Example 1: Overloading – Different Number of parameters in argument list

This example shows how method overloading is done by having different number of parameters

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
class Sample
{
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
obj.disp('a');        obj.disp('a',10);
   }
}
```

**Output:**

```
a a
10
```

In the above example – method disp() is overloaded based on the number of

parameters – We have two methods with the name `disp` but the parameters they have are different. Both are having different number of parameters.

## Example 2: Overloading – Difference in data type of parameters

In this example, method disp() is overloaded based on the data type of parameters – We have two methods with the name disp(), one with parameter of char type and another method with the parameter of int type.

```
class DisplayOverloading2
{
    public void disp(char c)
    {
        System.out.println(c);
    }
    public void disp(int c)
    {
       System.out.println(c );
    }
}

class Sample2
{
    public static void main(String args[])
    {
        DisplayOverloading2 obj = new DisplayOverloading2();
        obj.disp('a');
        obj.disp(5);
    }
}
```

Output:

```
a
5
```

## Example3: Overloading – Sequence of data type of arguments

Here method `disp()` is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
   public void disp(char c, int num)
   {
       System.out.println("I’m the first definition of method disp");
   }
   public void disp(int num, char c)
{
       System.out.println("I’m the second definition of method disp" );
   }
}
class Sample3
{
   public static void main(String args[])
   {
       DisplayOverloading3 obj = new DisplayOverloading3();
obj.disp('x', 51 );        obj.disp(52, 'y');
   }
}
```

**Output:**

```
I’m the first definition of method disp I’m
the second definition of method disp
```

# Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc.

**What it has to do with method overloading?**
Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.
Lets take an example to see what I am talking here:

```
class Demo{
   void disp(int a, double b){
```

```
        System.out.println("Method A");
    }    void disp(int a, double b, double
c){
        System.out.println("Method B");
    }
    public static void main(String args[]){
        Demo obj = new Demo();
        /* I am passing float value as a second argument but
* it got promoted to the type double, because there
* wasn't any method having arg list as (int, float)
         */
        obj.disp(100, 20.67f);
    }
}
```

Output:

```
Method A
```

As you can see that I have passed the float value while calling the disp() method but it got promoted to the double type as there wasn't any method with argument
list as (int, float)

But this type promotion doesn't always happen, lets see another example:

```
class Demo{
   void disp(int a, double b){
       System.out.println("Method A");
   }    void disp(int a, double b, double
c){
       System.out.println("Method B");
   }
   void disp(int a, float b){
       System.out.println("Method C");
   }
   public static void main(String args[]){
       Demo obj = new Demo();
       /* This time promotion won't happen as there is
        * a method with arg list as (int, float)
        */
       obj.disp(100, 20.67f);
   }
}
```

Output:

```
Method C
```

As you see that this time type promotion didn't happen because there was a method with matching argument type.

**Type Promotion table:**

The data type on the left side can be promoted to the any of the data type present in the right side of it.

```
byte → short → int → long
short → int → long int →
long → float → double float
```

```
→ double long → float →
double
```

# Lets see few Valid/invalid cases of method overloading

Case 1:

```
int mymethod(int a, int b, float c) int
mymethod(int var1, int var2, float var3)
```

Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

```
int mymethod(int a, int b)  int
mymethod(float var1, float var2)
```

Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

```
int mymethod(int a, int b)
int mymethod(int num)
```

Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

```
float mymethod(int a, float b)
float mymethod(float var1, int var2)
```

Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

```
int mymethod(int a, int b)  float
mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn't matter while overloading a method.

Guess the answers before checking it at the end of programs:
**Question 1 – return type, method name and argument list same.**

```
class Demo
{
   public int myMethod(int num1, int num2)
   {
       System.out.println("First myMethod of class Demo");
return num1+num2;
   }
   public int myMethod(int var1, int var2)
   {
       System.out.println("Second myMethod of class Demo");
return var1-var2;
   }
}
class Sample4
{
   public static void main(String args[])
   {
       Demo obj1= new Demo();
obj1.myMethod(10,10);
obj1.myMethod(20,12);
   }
}
```

**Answer:**

It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

## Question 2 – return type is different. Method name & argument list same.

```
class Demo2
{
   public double myMethod(int num1, int num2)
   {
      System.out.println("First myMethod of class Demo");
      return num1+num2;
   }
   public int myMethod(int var1, int var2)
   {
      System.out.println("Second myMethod of class Demo");
return var1-var2;
   }
}
class Sample5
{
   public static void main(String args[])
   {
      Demo2 obj2= new Demo2();
obj2.myMethod(10,10);
obj2.myMethod(20,12);
   }
}
```

**Answer:**

It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn't matter in case of overloading.**

# Method overriding in java with example

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method. In this guide, we will see what is method overriding in Java and why we use it.

## Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The `Boy` class extends `Human` class. Both the classes have a common method `void eat()`. Boy class is giving its own implementation to the `eat()` method or in other words it is overriding the `eat()` method.

The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
   //Overridden method
public void eat()
   {
      System.out.println("Human is eating");
   }
}
class Boy extends Human{
//Overriding method
public void eat(){
      System.out.println("Boy is eating");
   }
   public static void main( String args[]) {
      Boy obj = new Boy();
      //This will call the child class version of eat()
obj.eat();
   }
}
```

Output:

```
Boy is eating
```

# Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**.
This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

# Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method(parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
   //Overridden method
public void disp()
   {
       System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
//Overriding method
public void disp(){
       System.out.println("disp() method of Child class");
   }
   public void newMethod(){
       System.out.println("new method of child class");
   }
   public static void main( String args[]) {
 /* When Parent class reference refers to the parent class object   * then in
this case overridden method (the method of parent class)   *  is called.
        */
       ABC obj = new ABC();
       obj.disp();

       /* When parent class reference refers to the child class object
* then the overriding method (method of child class) is called.
        * This is called dynamic method dispatch and runtime polymorphism
        */
```

```
        ABC obj2 = new Demo();
        obj2.disp();
    }
}
```

Output:

```
disp() method of parent class disp()
method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).

**Note**: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object obj2 is calling the disp(). However if you try to call the newMethod() method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang.Error: Unresolved compilation  problem:
The method xyz() is undefined for the type ABC
```

# Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method(the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method ) cannot have private, protected and default Access modifier,because all of these three access modifiers are more restrictive than public.

For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
   public void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   protected void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
      obj.disp();
   }
}
```

Output:

```
Exception in thread "main" java.lang.Error: Unresolved compilation  problem:
Cannot reduce the visibility of the inherited method from MyBaseClass
```

However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
protected void disp()
   {
       System.out.println("Parent class method");
   }
}
class MyChildClass extends MyBaseClass{
   public void disp(){
      System.out.println("Child class method");
   }
   public static void main( String args[]) {
      MyChildClass obj = new MyChildClass();
obj.disp();
   }
}
```

Output:

```
Child class method
```

17. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
18. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
19. Binding of overridden methods happen at runtime which is known as dynamic binding.
20. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

# Super keyword in Method Overriding

The super keyword is used for calling the parent class method/constructor. `super.myMethod()` calls the myMethod() method of base class while `super()` calls the constructor of base class. Let's see the use of super in method Overriding.
As we know that we we override a method in child class, then call to the method using child class object calls the overridden method. By using super we can call the overridden method as shown in the example below:

```
class ABC{
   public void myMethod()
   {
       System.out.println("Overridden method");
   }
}
class Demo extends ABC{
public void myMethod(){
       //This will call the myMethod() of parent class
       super.myMethod();
       System.out.println("Overriding method");
   }    public static void main( String
args[]) {
       Demo obj = new Demo();
       obj.myMethod();
   }
}
```

Output:

```
Class ABC: mymethod() Class
Test: mymethod()
```

As you see using super keyword, we can access the overriden method.

# Polymorphism in Java with example

Polymorphism is one of the OOPs feature that allows us to perform a single action in different ways. For example, lets say we have a class Animal that has a method sound(). Since this is a generic class so we can't give it a implementation like: Roar, Meow, Oink etc. We had to give a generic message.

```
public class Animal{
   ...
   public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Now lets say we two subclasses of Animal Horse and Cat that extends class:

(see Inheritance) `Animal` class. We can provide the implementation to the same method like this:

```
public class Horse extends Animal{ ...
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
}
```

and

```
public class Cat extends Animal{
...
    @Override
    public void sound(){
        System.out.println("Meow");
    }
}
```

As you can see that although we had the common action for all subclasses `sound()` but there were different ways to do the same action. This is a perfect example of polymorphism (feature that allows us to perform a single action in different ways). It would not make any sense to just call the generic sound() method as each Animal has a different sound. Thus we can say that the action this method performs is based on the type of object.

# What is polymorphism in programming?

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations. As we have seen in the above example that we have defined the method sound() and have the multiple implementations of it in the different-2 sub classes.
Which sound() method will be called is determined at runtime so the example we gave above is a **runtime polymorphism example**.

Types of polymorphism and method overloading & overriding are covered in the separate tutorials. You can refer them here:
1. Method Overloading in Java – This is an example of compile time (or static polymorphism)
2. Method Overriding in Java – This is an example of runtime time (or dynamic polymorphism)
3. Types of Polymorphism – Runtime and compile time – This is our next tutorial where we have covered the types of polymorphism in detail. I would recommend you to go though method overloading and overriding before going though this topic.

Lets write down the complete code of it:

# Example 1: Polymorphism in Java

**Runtime Polymorphism example:**
Animal.java

```
public class Animal{
public void sound(){
      System.out.println("Animal is making a sound");
   }
}
```

Horse.java

```
class Horse extends Animal{
```

```
    @Override
    public void sound(){
        System.out.println("Neigh");
    }
    public static void main(String args[]){          Animal obj = new Horse();
       obj.sound();
    }
}
```

Output:

```
Neigh
```

Cat.java

```
public class Cat extends Animal{
    @Override
    public void sound(){
        System.out.println("Meow");
    }
    public static void main(String args[]){
       Animal obj = new Cat();
       obj.sound();
    }
}
```

Output:

```
Meow
```

# Example 2: Compile time Polymorphism

Method Overloading on the other hand is a compile time polymorphism example.

```
class Overload
{
    void demo (int a)
    {
       System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
       System.out.println ("a and b: " + a + "," + b);
    }
```

```
    double demo(double a) {
       System.out.println("double a: " + a);
return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
double result;          Obj .demo(10);
Obj .demo(10, 20);          result =
Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}
```

Here the method demo() is overloaded 3 times: first method has 1 int parameter, second method has 2 int parameters and third one is having double parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at runtime so this type of polymorphism is known as compile time polymorphism.

**Output:**

```
a: 10 a and b:
10,20 double a:
5.5 O/P : 30.25
```

# Types of polymorphism in java- Runtime and Compile time polymorphism

In the last tutorial we discussed Polymorphism in Java. In this guide we will see **types of polymorphism**. There are two types of polymorphism in java:

1) **Static Polymorphism** also known as compile time polymorphism
2) **Dynamic Polymorphism** also known as runtime polymorphism

# Compile time Polymorphism (or Static polymorphism)

Polymorphism that is resolved during compiler time is known as static polymorphism. Method overloading is an example of compile time polymorphism.
**Method Overloading**: This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters. We have already discussed Method overloading here: If you didn't read that guide, refer: Method Overloading in Java

## Example of static Polymorphism

Method overloading is one of the way java supports static polymorphism. Here we have two definitions of the same method add() which add method would be called is determined by the parameter list at the compile time. That is the reason this is also known as compile time polymorphism.

```
class SimpleCalculator
{
    int add(int a, int b)
    {
         return a+b;
    }
    int  add(int a, int b, int c)
    {
         return a+b+c;
    }
}
public class Demo
{
   public static void main(String args[])
   {
          SimpleCalculator obj = new SimpleCalculator();
```

```
        System.out.println(obj.add(10, 20));
        System.out.println(obj.add(10, 20, 30));
    }
}
```

**Output:**

```
30
60
```

# Runtime Polymorphism (or Dynamic polymorphism)

It is also known as Dynamic Method Dispatch. Dynamic polymorphism is a process in which a call to an overridden method is resolved at runtime, thats why it is called runtime polymorphism. I have already discussed method overriding in detail in a separate tutorial, refer it: Method Overriding in Java.

**Example**

In this example we have two classes ABC and XYZ. ABC is a parent class and XYZ is a child class. The child class is overriding the method myMethod() of parent class. In this example we have child class object assigned to the parent class reference so in order to determine which method would be called, the type of the object would be determined at run-time. It is the type of object that determines which version of the method would be called (not the type of reference).

To understand the concept of overriding, you should have the basic knowledge of inheritance in Java.

```
class ABC{
    public void myMethod(){
        System.out.println("Overridden Method");
    }
}
```

```
public class XYZ extends ABC{

   public void myMethod(){
       System.out.println("Overriding Method");
   }
   public static void main(String args[]){
       ABC obj = new XYZ();
       obj.myMethod();
   }
}
```

**Output:**

```
Overriding Method
```

When an overridden method is called through a reference of parent class, then type of the object determines which method is to be executed. Thus, this determination is made at run time.
Since both the classes, child class and parent class have the same method animalSound. Which version of the method(child class or parent class) will be called is determined at runtime by JVM.

**Few more overriding examples:**

```
ABC obj = new ABC();
obj.myMethod();
// This would call the myMethod() of parent class ABC

XYZ obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ

ABC obj = new XYZ();
obj.myMethod();
// This would call the myMethod() of child class XYZ
```

In the third case the method of child class is to be executed because which method is to be executed is determined by the type of object and since the object belongs to the child class, the child class version of myMethod() is called.

# Static and dynamic binding in java

Association of method call to the method body is known as binding. There are two types of binding: **Static Binding** that happens at compile time and **Dynamic Binding** that happens at runtime. Before I explain static and dynamic binding in java, lets see few terms that will help you understand this concept better.

## What is reference and object?

```
class Human{
.... }
class Boy extends Human{
   public static void main( String args[]) {
        /*This statement simply creates an object of class
         *Boy and assigns a reference of Boy to it*/
Boy obj1 = new Boy();

        /* Since Boy extends Human class. The object creation
* can be done in this way. Parent class reference
* can have child class reference assigned to it
         */
        Human obj2 = new Boy();
   }
}
```

## Static and Dynamic Binding in Java

As mentioned above, association of method definition to the method call is known as binding. There are two types of binding: Static binding and dynamic binding. Lets discuss them.

### Static Binding or Early Binding

The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private and final methods is compiletime. **Why?** The reason is that the these method cannot be overridden and the type of the class is determined at the compile time. Lets see an example to understand this:

## Static binding example

Here we have two classes Human and Boy. Both the classes have same method walk() but the method is static, which means it cannot be overriden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there.

```
class Human{
   public static void walk()
   {
       System.out.println("Human walks");
   }
}
class Boy extends Human{
public static void walk(){
       System.out.println("Boy walks");
   }    public static void main( String args[]) {
/* Reference is of Human type and object is
* Boy type
        */
       Human obj = new Boy();
       /* Reference is of HUman type and object is
* of Human type.
        */
       Human obj2 = new Human();
obj.walk();        obj2.walk();
   }
}
```

Output:

```
Human walks
```

```
Human walks
```

## Dynamic Binding or Late Binding

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late Binding. Method Overriding is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object**determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

## Dynamic binding example

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overriden method is determined at runtime by the type of object thus late binding happens.
Lets see an example to understand this:

```
class Human{
   //Overridden Method
public void walk()
{
```

```
        System.out.println("Human walks");
    } }
class Demo extends Human{
//Overriding Method
    public void walk(){
        System.out.println("Boy walks");
    }    public static void main( String args[]) {
/* Reference is of Human type and object is
* Boy type
         */
        Human obj = new Demo();
        /* Reference is of HUman type and object is
* of Human type.
         */
        Human obj2 = new Human();
obj.walk();         obj2.walk();
    }
}
```

Output:

```
Boy walks
Human walks
```

As you can see that the output is different than what we saw in the static binding example, because in this case while creation of object obj the type of the object is determined as a Boy type so method of Boy class is called. Remember the type of the object is determined at the runtime.

## Static Binding vs Dynamic Binding

Lets discuss the **difference between static and dynamic binding in Java**.

1. Static binding happens at compile-time while dynamic binding happens at runtime.
2. Binding of private, static and final methods always happen at compile time since these methods cannot be overridden. When the method overriding is actually happening and the reference of parent type is assigned to the object of child class type then such binding is resolved during runtime.
3. The binding of overloaded methods is static and the binding of overridden methods is dynamic.

# Abstract Class in Java with example

A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods(methods without body) as well as concrete methods (regular methods with body). A normal class(non-abstract class) cannot have abstract methods. In this guide we will learn what is a abstract class, why we use it and what are the rules that we must remember while working with it in Java.

An abstract class can not be **instantiated**, which means you are not allowed to create an **object**of it. Why? We will discuss that later in this guide.

## Why we need an abstract class?

Lets say we have a class `Animal` that has a method `sound()` and the subclasses(see inheritance) of it like `Dog`, `Lion`, `Horse`, `Cat` etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class. This is because every child class must override this method to give its own implementation details, like `Lion` class will say "Roar" in this method and `Dog` class will say "Woof".

So when we know that all the animal child classes will and should override this method, then there is no point to implement this method in parent class. Thus, making this method abstract would be the good choice as by making this method abstract we force all the sub classes to implement this method( otherwise you will get compilation error), also we need not to give any implementation to this method in parent class.

Since the `Animal` class has an abstract method, you must need to declare this class abstract.

Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

## Abstract class Example

```
//abstract parent class abstract
class Animal{    //abstract
method    public abstract void
sound();
}
//Dog class extends Animal class public
class Dog extends Animal{

   public void sound(){
       System.out.println("Woof");
   }
   public static void main(String args[]){
       Animal obj = new Dog();
       obj.sound();
   }
}
```

Output:

```
Woof
```

Hence for such kind of scenarios we generally declare the class as abstract and later **concrete classes** extend these classes and override the methods accordingly and can have their own methods as well.

## Abstract class declaration

An abstract class outlines the methods but not necessarily implements all the methods.

```
//Declaration  using  abstract  keyword
abstract class A{
   //This    is    abstract    method
abstract void myMethod();
```

```
   //This is concrete method with body
void anotherMethod(){
      //Does something
   }
}
```

## Rules

**Note 1:** As we seen in the above example, there are cases when it is difficult or often unnecessary to implement all the methods in parent class. In these cases, we can declare the parent class as abstract, which makes it a special class which is not complete on its own.

A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

**Note 2:** Abstract class cannot be instantiated which means you cannot create the object of it. To use this class, you need to create another class that extends this this class and provides the implementation of abstract methods, then you can use the object of that child class to call non-abstract methods of parent class as well as implemented methods(those that were abstract in parent but implemented in child class).

**Note 3:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

**Do you know?** Since abstract class allows concrete methods as well, it does not provide 100% abstraction. You can say that it provides partial abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.

Interfaces on the other hand are used for 100% abstraction.

## Why can’t we create the object of an abstract class?

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class then if someone calls the abstract method using that object then What would happen?There would be no actual implementation of the method to invoke.
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

## Example to demonstrate that object creation of abstract class is not allowed

As discussed above, we cannot instantiate an abstract class. This program throws a compilation error.

```
abstract class AbstractDemo{
public void myMethod(){
```

```
      System.out.println("Hello");
   }
   abstract public void anotherMethod();
}
public class Demo extends AbstractDemo{

   public void anotherMethod() {
        System.out.print("Abstract method");
   }
   public static void main(String args[])
   {
      //error: You can't create object of it
AbstractDemo obj = new AbstractDemo();
obj.anotherMethod();
   }
}
```

Output:

```
Unresolved compilation problem: Cannot instantiate the type AbstractDemo
```

Note: The class that extends the abstract class, have to implement all the abstract methods of it, else you have to declare that class abstract as well.

## Abstract class vs Concrete class

A class which is not abstract is referred as **Concrete class**. In the above example that we have seen in the beginning of this guide, `Animal` is a abstract class and `Cat`, `Dog` & `Lion` are concrete classes.

**Key Points:**

1. An abstract class has no use until unless it is extended by some other class.

2. If you declare an **abstract method** in a class then you must declare the class abstract as well. you can't have abstract method in a concrete class. It's vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
3. It can have non-abstract method (concrete) as well.

Lets just see some basics and example of abstract method.
1) Abstract method has no body.
2) Always end the declaration with a **semicolon**(;).
3) It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
4) A class has to be declared abstract to have abstract methods.

**Note:** The class which is extending abstract class must override all the abstract methods.

## Example of Abstract class and method

```
abstract class MyClass{
public void disp(){
     System.out.println("Concrete method of parent class");
   }
   abstract public void disp2();
}
class Demo extends MyClass{
   /* Must Override this method while extending
    * MyClas
*/
   public void disp2()
   {
       System.out.println("overriding abstract method");
   }
   public static void main(String args[]){
       Demo obj = new Demo();
obj.disp2();
   }
}
```

Output:

```
overriding abstract method
```

# Abstract method in Java with examples

A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class, if you have not yet checked it out read it here: Abstract class in Java, before reading this guide. This is how an abstract method looks in java:

```
public abstract int myMethod(int n1, int n2);
```

As you see this has no body.

## Rules of Abstract Method

1. Abstract methods don't have body, they just have method signature as shown above.

2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn't need to have an abstract method compulsory.
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

# Example 1: abstract method in an abstract class

```
//abstract class abstract
class Sum{
   /* These two are abstract methods, the child class
    * must implement these methods
    */
   public abstract int sumOfTwo(int n1, int n2);
   public abstract int sumOfThree(int n1, int n2, int n3);

   //Regular method
   public void disp(){
       System.out.println("Method of class Sum");
   }
}
//Regular class extends abstract class class
Demo extends Sum{

   /* If I don't provide the implementation of these two methods, the
    * program will throw compilation error.
    */
   public int sumOfTwo(int num1, int num2){
       return num1+num2;
   }
   public int sumOfThree(int num1, int num2, int num3){
       return num1+num2+num3;
   }
   public static void main(String args[]){
       Sum obj = new Demo();
```

```
      System.out.println(obj.sumOfTwo(3, 7));
      System.out.println(obj.sumOfThree(4, 3, 19));
      obj.disp();
   }
}
```

Output:

```
10
26
Method of class Sum
```

# Example 2: abstract method in interface

All the methods of an interface are public abstract by default. You cannot have concrete (regular methods with body) methods in an interface.

```
//Interface interface
Multiply{
//abstract methods
   public abstract int multiplyTwo(int n1, int n2);

   /* We need not to mention public and abstract in interface
*     as all the methods in interface are
*     public and abstract by default so the compiler will     * treat this as
*     public abstract multiplyThree(int n1, int n2, int n3);
    */
   int multiplyThree(int n1, int n2, int n3);

   /* Regular (or concrete) methods are not allowed in an interface
*     so if I uncomment this method, you will get compilation error     * public
void disp(){
*     System.out.println("I will give error if u uncomment me");
*     }
    */
}
class Demo implements Multiply{
   public int multiplyTwo(int num1, int num2){
return num1*num2;
   }
   public int multiplyThree(int num1, int num2, int num3){
return num1*num2*num3;
   }
   public static void main(String args[]){
      Multiply obj = new Demo();
```

```
        System.out.println(obj.multiplyTwo(3, 7));
        System.out.println(obj.multiplyThree(1, 9, 0));
    }
}
```

Output:

```
21
0
```

# Interface in java with example programs

In the last tutorial we discussed abstract class which is used for achieving partial abstraction. Unlike abstract class an interface is used for full abstraction. Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user(See: Abstraction). In this guide, we will cover **what is an interface in java**, why we use it and what are rules that we must follow while using interfaces in Java Programming.

## What is an interface in Java?

Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body, see: Java abstract method). Also, the variables declared in an interface are public, static & final by default.

## What is the use of interface in Java?

As mentioned above they are used for full abstraction. Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class.

**Syntax:**

Interfaces are declared by specifying a keyword “interface”. E.g.:

```
interface MyInterface
{
   /* All the methods are public abstract by default
    * As you see they have no body
    */
   public void method1();
public void method2(); }
```

# Example of an Interface in Java

This is how a class implements an interface. It has to provide the body of all the methods that are declared in interface or in other words you can say that class has to implement all the methods of interface.

**Do you know?** class `implements` interface but an interface `extends` another interface.

```
interface MyInterface
{
   /* compiler will treat them as:
* public abstract void method1();
* public abstract void method2();
    */
   public void method1();
public void method2();
}
class Demo implements MyInterface
{
   /* This class must have to implement both the abstract methods
* else you will get compilation error
    */
   public void method1()
   {
       System.out.println("implementation of method1");
   }
   public void method2()
   {
       System.out.println("implementation of method2");
   }
   public static void main(String arg[])
   {
       MyInterface obj = new Demo();
       obj.method1();
   }
```

```
}
```

Output:

```
implementation of method1
```

# Interface and Inheritance

As discussed above, an interface can not implement another interface. It has to extend the other interface. See the below example where we have two interfaces Inf1 and Inf2. Inf2 extends Inf1 so If class implements the Inf2 it has to provide implementation of all the methods of interfaces Inf2 as well as Inf1.

```
interface Inf1{
   public void method1();
}
interface Inf2 extends Inf1 {
public void method2();
}
public class Demo implements Inf2{
   /* Even though this class is only implementing the
* interface Inf2, it has to implement all the methods
* of Inf1 as well because the interface Inf2 extends Inf1
    */
    public void method1(){
      System.out.println("method1");
    }
    public void method2(){
      System.out.println("method2");
    }
    public static void main(String args[]){
      Inf2 obj = new Demo();
      obj.method2();
    }
}
```

In this program, the class Demo only implements interface Inf2, however it has to provide the implementation of all the methods of interface Inf1 as well, because interface Inf2 extends Inf1.

# CHAPTER-9

## Tag or Marker interface in Java

An empty interface is known as tag or marker interface. For example Serializable, EventListener, Remote(java.rmi.Remote) are tag interfaces. These interfaces do not have any field and methods in it. Read more about it here.

## Nested interfaces

An interface which is declared inside another interface or class is called nested interface. They are also known as inner interface. For example Entry interface in collections framework is declared inside Map interface, that's why we don' use it directly, rather we use it like this: Map.Entry.

**Key points:** Here are the key points to remember about interfaces:

1) We can't instantiate an interface in java. That means we cannot create the object of an interface

2) Interface provides full abstraction as none of its methods have body. On the other hand abstract class provides partial abstraction as it can have abstract and concrete(methods with body) methods both.

3) `implements` keyword is used by classes to implement an interface.

4) While providing implementation in class of any method of an interface, it needs to be mentioned as public.

5) Class that implements any interface must implement all the methods of that interface, else the class should be declared abstract.

6) Interface cannot be declared as private, protected or transient.

7) All the interface methods are by default **abstract and public**.

8) Variables declared in interface are **public, static and final** by default.

```
interface Try
{
   int a=10;
public int a=10;
   public static final int a=10;
final int a=10;    static int
a=0;
}
```

All of the above statements are identical.

9) Interface variables must be initialized at the time of declaration otherwise compiler will throw an error.

```
interface Try
{
      int x;//Compile-time error
}
```

Above code will throw a compile time error as the value of the variable x is not initialized at the time of declaration.

10) Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface “Try” which has a variable x. When we tried to set the value for variable x we got compilation error as the variable x is public static **final** by default and final variables can not be re-initialized.

```
class Sample implements Try
{
  public static void main(String args[])
  {
     x=20; //compile time error
  }
}
```

11) An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

12) A **class** can implement any **number of interfaces**.

13) If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

```
interface A
{
   public void aaa();
}
interface B
{
   public void aaa();
}
class Central implements A,B
{
   public void aaa()
   {
        //Any Code here
   }
   public static void main(String args[])
   {
```

```
        //Statements
    }
}
```

14) A class cannot implement two interfaces that have methods with same name but different return type.

```
interface A {
   public void aaa();
} interface B {
   public int aaa();
}

class Central implements A,B
{
   public void aaa() // error
   {
}
   public int aaa() // error
   {
}
   public static void main(String args[])
   {
   }
}
```

15) Variable names conflicts can be resolved by interface name.

```
interface A
{
    int x=10;
}
interface B
{
    int x=100;
}
class Hello implements A,B
{
    public static void Main(String args[])
    {
       /* reference to x is ambiguous both variables are x
```

```
* so we are using interface name to resolve the
* variable
        */
       System.out.println(x);
       System.out.println(A.x);
       System.out.println(B.x);
    }
}
```

# Advantages of interface in java:

Advantages of using interfaces are as follows:

1. Without bothering about the implementation part, we can achieve the security of implementation
2. In java, **multiple inheritance** is not allowed, however you can use interface to make use of it as you can implement more than one interface.

# Encapsulation in Java with example

Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation. In this guide we will see how to do encapsulation in java program.For other OOPs topics such as inheritance and polymorphism, refer OOPs concepts

## What is encapsulation?

The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class.

However if we setup public getter and setter methods to update (for example `void setSSN(int ssn)`)and read (for example `int getSSN()`) the private data fields then the outside class can access those private data fields via public methods.

This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That's why encapsulation is known as **data hiding.** Lets see an example to understand this concept better.

## Example of Encapsulation in Java

How to implement encapsulation in java:
1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EncapsulationDemo{
```

```
    private int ssn;
private String empName;
private int empAge;


    //Getter and Setter methods
public int getEmpSSN(){
return ssn;
    }
    public String getEmpName(){
return empName;
    }      public int
getEmpAge(){         return
empAge;
    }
    public void setEmpAge(int newValue){
empAge = newValue;
    }
    public void setEmpName(String newValue){
empName = newValue;
    }
    public void setEmpSSN(int newValue){
ssn = newValue;
    }
}
public class EncapsTest{
    public static void main(String args[]){
         EncapsulationDemo obj = new EncapsulationDemo();
obj.setEmpName("Mario");          obj.setEmpAge(32);
obj.setEmpSSN(112233);
         System.out.println("Employee Name: " + obj.getEmpName());
         System.out.println("Employee SSN: " + obj.getEmpSSN());
System.out.println("Employee Age: " + obj.getEmpAge());
    }
}
```

**Output:**

```
Employee Name: Mario
Employee SSN: 112233 Employee
Age: 32
```

In above example all the three data members (or data fields) are private which cannot be accessed directly. These fields can be accessed via public methods only.

Fields `empName`, `ssn` and `empAge` are made hidden data fields using encapsulation technique of OOPs.

## Advantages of encapsulation

1. It improves maintainability and flexibility and re-usability: for e.g. In the above code the implementation code of `void setEmpName(String name)` and `String getEmpName()` can be changed at any point of time. Since the implementation is purely hidden for outside classes they would still be accessing the private field empName using the same methods (`setEmpName(String name)` and `getEmpName()`). Hence the code can be maintained at any point of time without breaking the classes that uses the code. This improves the reusability of the underlying class.
2. The fields can be made read-only (If we don't define setter methods in the class) or write-only (If we don't define the getter methods in the class). For e.g. If we have a field(or variable) that we don't want to be changed so we simply define the variable as private and instead of set and get both we just need to define the get method for that variable. Since the set method is not present there is no way an outside class can modify the value of that field.
3. User would not be knowing what is going on behind the scene. They would only be knowing that to update a field call set method and to read a field call get method but what these set and get methods are doing is purely hidden from them.

Encapsulation is also known as "**data Hiding**".

# Packages in java and how to use them

A package as the name suggests is a pack(group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two **types of packages in Java**: built-in packages and the packages

we can create (also known as user defined package). In this guide we will learn what are packages, what are user-defined packages in java and how to use them.

In java we have several built-in packages, for example when we need user input,

we import a package like this: `import java.util.Scanner`

Here:
→ **java** is a top level package
→ **util** is a sub package
→ and **Scanner** is a class which is present in the sub package **util**.

Before we see how to create a user-defined package in java, lets see the advantages of using a package.

## Advantages of using a package in Java

These are the reasons why you should use packages in Java:

- **Reusability**: While developing a project in java, we often feel that there are few things that we are writing again and again in our code. Using packages, you can create such things in form of classes inside a package and whenever you need to perform that same task, just import that package and use the class.
- **Better Organization**: Again, in large java projects where we have several hundreds of classes, it is always required to group the similar types of classes in a meaningful package name so that you can organize your project better and when you need something you can quickly locate it and use it, which improves the efficiency.
- **Name Conflicts**: We can define two classes with the same name in different packages so to avoid name collision, we can use packages

## Types of packages in Java

As mentioned in the beginning of this guide that we have two types of packages in java.
1) User defined package: The package we create is called user-defined package.
2) Built-in package: The already defined package like java.io.*, java.lang.* etc are known as built-in packages.

We have already discussed built-in packages, lets discuss user-defined packages with the help of examples.

# Example 1: Java packages

I have created a class `Calculator` inside a package name `letmecalculate`. To create a class inside a package, declare the package name in the first statement in your program. A class can have only one package declaration. Calculator.java file created inside a package `letmecalculate`

```
package letmecalculate;

public class Calculator {
public int add(int a, int b){
      return a+b;
   }    public static void main(String
args[]){     Calculator obj = new
Calculator();
      System.out.println(obj.add(10, 20));
   }
}
```

Now lets see how to use this package in another program.

```
import letmecalculate.Calculator; public
class Demo{
   public static void main(String args[]){
      Calculator obj = new Calculator();
      System.out.println(obj.add(100, 200));
   }
}
```

To use the class Calculator, I have imported the package `letmecalculate`. In the above program I have imported the package as `letmecalculate.Calculator`, this only imports the Calculator class. However if you have several classes inside package `letmecalculate` then you can import the package like this, to use all the

classes of this package.

```
import letmecalculate.*;
```

## Example 2: Creating a class inside package while importing another package

As we have seen that both package declaration and package import should be the first statement in your java program. Lets see what should be the order when we are creating a class inside a package while importing another package.

```
//Declaring a package package
anotherpackage; //importing a
package
import letmecalculate.Calculator; public
class Example{
   public static void main(String args[]){
       Calculator obj = new Calculator();
       System.out.println(obj.add(100, 200));
   }
}
```

So the order in this case should be:

→ package declaration

→ package import

## Example 3: Using fully qualified name while importing a class

You can use fully qualified name to avoid the import statement. Lets see an example to understand this:

**Calculator.java**

```
package letmecalculate;
public class Calculator {
   public int add(int a, int b){
         return a+b;
   }
   public static void main(String args[]){
         Calculator obj = new Calculator();
         System.out.println(obj.add(10, 20));
   }
}
```

**Example.java**

```
//Declaring a package
package anotherpackage;
public class Example{
   public static void main(String args[]){
         //Using fully qualified name instead of import
         letmecalculate.Calculator obj =
                  new letmecalculate.Calculator();
         System.out.println(obj.add(100, 200));
   }
}
```

In the Example class, instead of importing the package, I have used the full qualified name such as package_name.class_name to create the object of it. You may also want to read: static import in Java

## Sub packages in Java

A package inside another package is known as sub package. For example If I create a package inside letmecalculate package then that will be called sub package.

Lets say I have created another package inside letmecalculate and the sub package name is multiply. So if I create a class in this subpackage it should have this package declaration in the beginning:

```
package letmecalculate.multiply;
```

Multiplication.java

```
package letmecalculate.multiply;
public class Multiplication {
        int product(int a, int b){
                return a*b;
        }
}
```

Now if I need to use this Multiplication class I have to either import the package like this:

```
import letmecalculate.multiply;
```

or I can use fully qualified name like this:

```
letmecalculate.multiply.Multiplication     obj     =
new letmecalculate.multiply.Multiplication();
```

# Points to remember:

1. Sometimes class name conflict may occur. For example: Lets say we have two packages **abcpackage** and **xyzpackage** and both the packages have a class with the same name, let it be JavaExample.java. Now suppose a class import both these packages like this:

```
import abcpackage.*; import
xyzpackage.*;
```

This will throw compilation error. To avoid such errors you need to use the fully qualified name method that I have shown above. For example

```
abcpackage.JavaExample obj = new abcpackage.JavaExample(); xyzpackage.JavaExample
obj2 = new xyzpackage.JavaExample();
```

This way you can avoid the import package statements and avoid that name conflict error.

2. I have already discussed this above, let me mention it again here. If we create a class inside a package while importing another package then the package declaration should be the first statement, followed by package import. For example:

```
package abcpackage;
import xyzpackage.*;
```

3. A class can have only one package declaration but it can have more than one package import statements. For example:

```
package abcpackage; //This should be one
import xyzpackage; import anotherpackage;
import anything;
```

4. The wild card import like package.* should be used carefully when working with subpackages. For example: Lets say: we have a package **abc** and inside that package we have another package **foo**, now **foo** is a subpackage.

classes inside abc are: Example1, Example 2, Example 3 classes
inside foo are: Demo1, Demo2

So if I import the package **abc** using wildcard like this:

```
import abc.*;
```

Then it will only import classes Example1, Example2 and Example3 but it will not import the classes of sub package.

To import the classes of subpackage you need to import like this:

```
import abc.foo.*;
```

This will import Demo1 and Demo2 but it will not import the Example1, Example2 and Example3.

So to import all the classes present in package and subpackage, we need to use two import statements like this:

```
import abc.*;
import abc.foo.*;
```

# Java Access Modifiers – Public, Private, Protected & Default

You must have seen public, private and protected keywords while practising java programs, these are called access modifiers. An access modifier restricts the access of a class, constructor, data member and method in another class. In java we have four access modifiers:
1. default
2. private
3. protected
4. public

## 1. Default access modifier

When we do not mention any access modifier, it is called default access modifier. The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class. Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package. Lets see an example to understand this:

### Default Access Modifier Example in Java

To understand this example, you must have the knowledge of packages in java.

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this

program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

**Addition.java**

```
package abcpackage;

public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
       return a+b;
   }
}
```

**Test.java**

```
package xyzpackage;

/* We are importing the abcpackage
 * but still we will get error because the
*      class we are trying to use has default
access  * modifier.
 */ import
abcpackage.*; public
class Test {
   public static void main(String args[]){
       Addition obj = new Addition();
         /* It will throw error because we are trying to access
*      the default method in another package
           */
       obj.addTwoNumbers(10, 21);
   }
}
```

**Output:**

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type Addition is not visible at
xyzpackage.Test.main(Test.java:12)
```

# 2. Private access modifier

The scope of private modifier is limited to the class only.

1. Private Data members and methods are only accessible within the class
2. Class and Interface cannot be declared as private
3. If a class has private constructor then you cannot create the object of that class from outside of the class.

Let's see an example to understand this:

## Private access modifier example in java

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

```
class ABC{
   private double num = 100;
   private int square(int a){
       return a*a;
   }
}
public class Example{
   public static void main(String args[]){
       ABC obj = new ABC();
       System.out.println(obj.num);
       System.out.println(obj.square(10));
}
```

} Output:

```
Compile - time error
```

# 3. Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes.

Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

## Protected access modifier example in Java

In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

**Addition.java**

```
package abcpackage; public
class Addition {


   protected int addTwoNumbers(int a, int b){
       return a+b;
   }
}
```

**Test.java**

```
package xyzpackage; import
abcpackage.*; class Test
extends Addition{
   public static void main(String args[]){
       Test obj = new Test();
       System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

Output:

```
33
```

# 4. Public access modifier

The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

## public access modifier example in java

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.
Addition.java

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){
       return a+b;
   }
}
```

Test.java

```
package xyzpackage; import
abcpackage.*; class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Output:

```
101
```

Lets see the scope of these access modifiers in tabular form:

# The scope of access modifiers in tabular form

| | Class | Package | Subclass (same package) | Subclass (diff package) | Outside Class |
|---|---|---|---|---|---|
| public | Yes | Yes | Yes | Yes | Yes |
| protected | Yes | Yes | Yes | Yes | No |
| default | Yes | Yes | Yes | No | No |
| private | Yes | No | No | No | No |

# Garbage Collection in Java

When JVM starts up, it creates a heap area which is known as runtime data area. This is where all the objects (instances of class) are stored. Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use. The process of removing unused objects from heap memory is known as **Garbage collection** and this is a part of memory management in Java.

Languages like C/C++ **don't** support automatic garbage collection, however in java, the garbage collection is automatic.

Now we know that the garbage collection in java is automatic. Lets see when does java performs garbage collection.

## When does java perform garbage collection?

### 1. When the object is no longer reachable:

```
BeginnersBook obj = new BeginnersBook();
obj = null;
```

Here the reference obj was pointing to the object of class BeginnersBook but since we have assigned a null value to it, this is no longer pointing to that object, which makes the BeginnersBook object unreachable and thus unusable. Such objects are automatically available for garbage collection in Java.

Another example is:

```
char[] sayhello = { 'h', 'e', 'l', 'l', 'o'};
String str = new String(sayhello);
str = null;
```

Here the reference str of String class was pointing to a string "hello" in the heap memory but since we have assigned the null value to str, the object "hello" present in the heap memory is unusable.

**2. When one reference is copied to another reference:**

```
BeginnersBook obj1 = new BeginnersBook();
BeginnersBook obj2 = new BeginnersBook();
obj2 = obj1;
```

Here we have assigned the reference obj1 to obj2, which means the instance (object) pointed by (referenced by) obj2 is not reachable and available for garbage collection.

# How to request JVM for garbage collection

We now know that the unreachable and unusable objects are available for garbage collection but the garbage collection process doesn't happen instantly. Which means once the objects are ready for garbage collection they must to have to wait for JVM to run the memory cleanup program that performs garbage collection. However you can request to JVM for garbage collection by calling **System.gc()** method (see the example below).

## Garbage Collection Example in Java

In this example we are demonstrating the garbage collection by calling System.gc(). In this code we have overridden a finalize() method. This method is invoked just before a object is destroyed by java garbage collection process. This is the reason you would see in the output that this method has been invoked twice.

```
public class JavaExample{
   public static void main(String args[]){
        /* Here we are intentionally assigning a null
* value to a reference so that the object becomes
* non reachable
         */
       JavaExample obj=new JavaExample();
       obj=null;
```

```
        /* Here we are intentionally assigning reference a
* to the another reference b to make the object referenced
* by b unusable.
         */
       JavaExample a = new JavaExample();
       JavaExample b = new JavaExample();
       b = a;
       System.gc();
   }
   protected void finalize() throws Throwable
   {
        System.out.println("Garbage collection is performed by JVM");
   }
}
```

Output:

```
Garbage collection is performed by JVM
```
Garbage collection is performed by JVM

# CHAPTER-10

## Exception handling in java with examples

Exception handling is one of the most important feature of java programming that allows us to handle the runtime errors caused by exceptions. In this guide, we will learn what is an exception, types of it, exception classes and how to handle exceptions in java with examples.

### What is an exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

#### Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

### Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:
**An exception generated by the system is given below**

```
 Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
 ExceptionDemo : The class name
main : The method name
 ExceptionDemo.java : The filename
java:5 : Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

## Advantage of exception handling

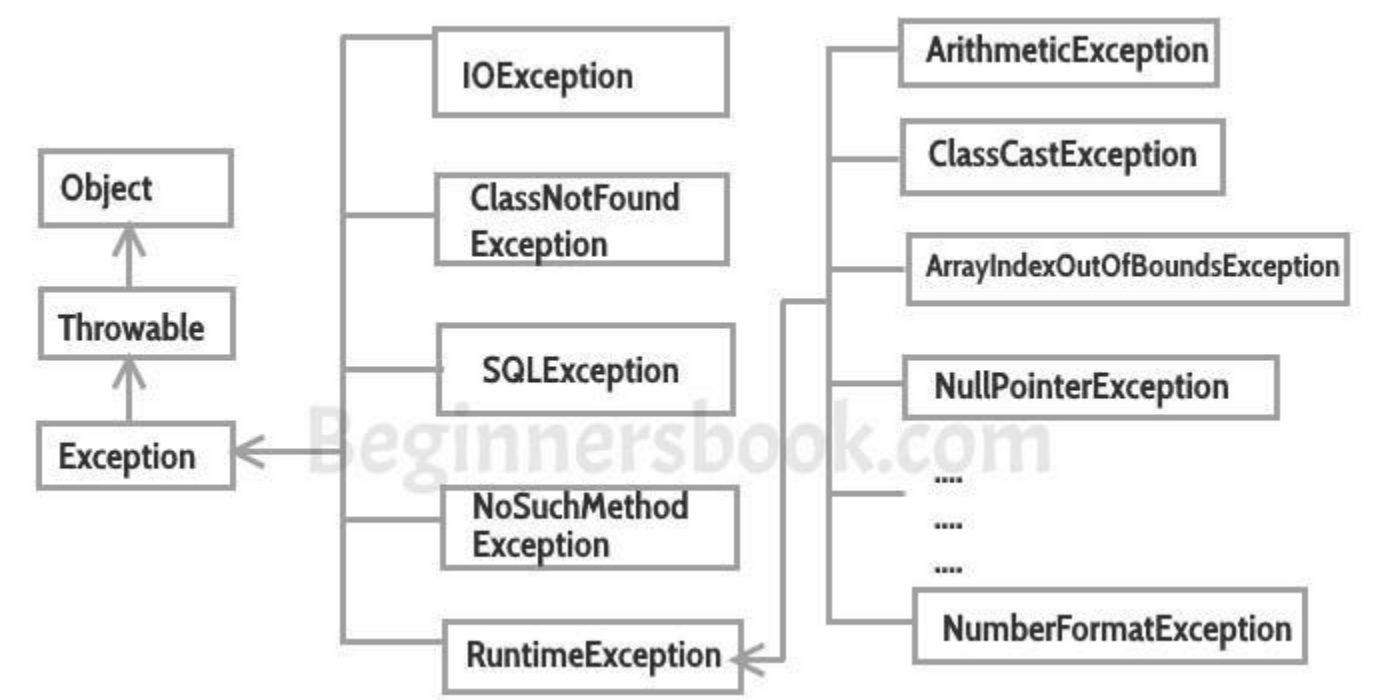
Exception handling ensures that the flow of the program doesn’t break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly. By handling we make sure that all the statements execute and the flow of program doesn’t break.

# Difference between error and exception

**Errors** indicate that something severe enough has gone wrong, the application should crash rather than try to handle the error.

**Exceptions** are events that occurs in the code. A programmer can handle such conditions and take necessary corrective actions. Few examples:
NullPointerException – When you try to use a reference that points to null.
ArithmeticException – When bad data is provided by user, for example, when you try to divide a number by zero this exception occurs because dividing a number by zero is undefined.

ArrayIndexOutOfBoundsException – When you try to access the elements of an array out of its bounds, for example array size is 5 (which means it has five elements) and you are trying to access the 10th element.




# Types of exceptions

There are two types of exceptions in Java:
1)Checked exceptions
2)Unchecked exceptions

## Checked exceptions

All exceptions other than Runtime Exceptions are known as Checked exceptions as the compiler checks them during compilation to see whether the programmer has handled them or not. If these exceptions are not handled/declared in the program, you will get compilation error. For example, SQLException, IOException, ClassNotFoundException etc.

## Unchecked Exceptions

Runtime Exceptions are also known as Unchecked Exceptions. These exceptions are not checked at compile-time so compiler does not check whether the programmer has handled them or not but it’s the responsibility of the programmer to handle these exceptions and provide a safe exit. For example,

ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.

Compiler will never force you to catch such exception or force you to declare it in the method using throws keyword.

# Try Catch in Java – Exception handling

## Try block

The try block contains set of statements where an exception can occur. A try block is always followed by a catch block, which handles the exception that occurs in associated try block. A try block must be followed by catch blocks or finally block or both.

### Syntax of try block

```
try{
   //statements that may cause an exception
}
```

While writing a program, if you think that certain statements in a program can throw a exception, enclosed them in try block and handle that exception

## Catch block

A catch block is where you handle the exceptions, this block must follow the try block. A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks. When an exception occurs in try block, the corresponding catch block that handles that particular exception executes. For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

### Syntax of try catch in java

```
try
{
     //statements that may cause an exception
}
catch (exception(type) e(object))
{
     //error handling code
}
```

# Example: try catch block

If an exception occurs in try block then the control of execution is passed to the corresponding catch block. A single try block can have multiple catch blocks associated with it, you should place the catch blocks in such a way that the generic exception handler catch block is at the last(see in the example below). The generic exception handler can handle all the exceptions but you should place is at the end, if you place it at the before all the catch blocks then it will display the generic message. You always want to give the user a meaningful message for each type of exception rather then a generic message.

```
class Example1 {
   public static void main(String args[]) {
int num1, num2;
      try {
         /* We suspect that this block of statement can throw
* exception so we handled it by placing these statements
* inside try and handled the exception in catch block
          */
num1 = 0;
         num2 = 62 / num1;
         System.out.println(num2);
         System.out.println("Hey I'm at the end of try block");
      }
      catch (ArithmeticException e) {
         /* This block will only execute if any Arithmetic exception
* occurs in try block
          */
         System.out.println("You should not divide a number by zero");
      }
      catch (Exception e) {
         /* This is a generic Exception handler which means it can handle
* all the exceptions. This will execute if the exception is not
* handled by previous catch blocks.
          */
```

```
            System.out.println("Exception occurred");
        }
        System.out.println("I'm out of try-catch block in Java.");
    }
}
```

Output:

```
You should not divide a number by zero
I'm out of try-catch block in Java.
```

## Multiple catch blocks in Java

The example we seen above is having multiple catch blocks, lets see few rules about multiple catch blocks with the help of examples. To read this in detail, see catching multiple exceptions in java.

1. As I mentioned above, a single try block can have any number of catch blocks.
2. A generic catch block can handle all the exceptions. Whether it is ArrayIndexOutOfBoundsException or ArithmeticException or NullPointerException or any other type of exception, this handles all of them. To see the examples of NullPointerException and ArrayIndexOutOfBoundsException, refer this article: Exception Handling example programs.

```
catch(Exception e){
  //This catch block catches all the exceptions
}
```

If you are wondering why we need other catch handlers when we have a generic that can handle all. This is because in generic exception handler you can display a message but you are not sure for which type of exception it may trigger so it will display the same message for all the exceptions and user may not be able to understand which exception occurred. Thats the reason you should place is at the end of all the specific exception catch blocks

3. If no exception occurs in try block then the catch blocks are completely ignored.
4. Corresponding catch blocks execute for that specific type of exception:

catch(ArithmeticException e) is a catch block that can hanlde ArithmeticException
catch(NullPointerException e) is a catch block that can handle

NullPointerException

5. You can also throw exception, which is an advanced topic and I have covered it in separate tutorials: user defined exception, throws keyword, throw vs throws.

## Example of Multiple catch blocks

```
class Example2{
   public static void main(String args[]){
try{
         int a[]=new int[7];
```

```
        a[4]=30/0;
        System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
       System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
       System.out.println("Warning: Some Other exception");
    }
   System.out.println("Out of try-catch block...");
  } }
```

Output:

```
Warning: ArithmeticException Out
of try-catch block...
```

In the above example there are multiple catch blocks and these catch blocks executes sequentially when an exception occurs in try block. Which means if you put the last catch block ( catch(Exception e)) at the first place, just after try block then in case of any exception this block will execute as it can handle all exceptions. This catch block should be placed at the last to avoid such situations.

## Finally block

I have covered this in a separate tutorial here: java finally block. For now you just need to know that this block executes whether an exception occurs or not. You should place those statements in finally blocks, that must execute whether exception occurs or not.

# How to Catch multiple exceptions

## Catching multiple exceptions

Lets take an example to understand how to handle multiple exceptions.

```
class Example{
   public static void main(String args[]){
try{
         int arr[]=new int[7];
         arr[4]=30/0;
         System.out.println("Last Statement of try block");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
}
      System.out.println("Out of the try-catch block");
   }
}
```

**Output:**

```
You should not divide a number by zero Out
of the try-catch block
```

In the above example, the first catch block got executed because the code we have written in try block throws ArithmeticException (because we divided the number by zero).

**Now lets change the code a little bit and see the change in output:**

```
class Example{
   public static void main(String args[]){
try{
         int arr[]=new int[7];
arr[10]=10/5;
         System.out.println("Last Statement of try block");
      }
      catch(ArithmeticException e){
```

```
         System.out.println("You should not divide a number by zero");
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
}
      System.out.println("Out of the try-catch block");
   } }
```

Output:

```
Accessing array elements outside of the limit Out
of the try-catch block
```

In this case, the second catch block got executed because the code throws ArrayIndexOutOfBoundsException. We are trying to access the 11th element of array in above program but the array size is only 7.

**What did we observe from the above two examples?**

1. It is clear that when an exception occurs, the specific catch block (that declares that exception) executes. This is why in first example first block executed and in second example second catch.
2. Although I have not shown you above, but if an exception occurs in above code which is not Arithmetic and ArrayIndexOutOfBounds then the last generic catch handler would execute.

**Lets change the code again and see the output:**

```
class Example{
   public static void main(String args[]){
try{
         int arr[]=new int[7];
         arr[10]=10/5;
         System.out.println("Last Statement of try block");
      }
      catch(Exception e){
         System.out.println("Some Other Exception");
      }
      catch(ArithmeticException e){
         System.out.println("You should not divide a number by zero");
      }
```

```
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Accessing array elements outside of the limit");
}
      System.out.println("Out of the try-catch block");
   }
}
```

**Output:**

```
Compile time error: Exception in thread "main" java.lang.Error:
Unresolved compilation problems: Unreachable catch block for ArithmeticException.
It is already handled by the catch block for Exception Unreachable catch block
for ArrayIndexOutOfBoundsException. It is already handled by the catch block for
Exception at Example.main(Example1.java:11)
```

**Why we got this error?**
This is because we placed the generic exception catch block at the first place which means that none of the catch blocks placed after this block is reachable. You should always place this block at the end of all other specific exception catch blocks.

# Nested try catch block in Java – Exception handling

When a try catch block is present in another try block then it is called the nested try catch block. Each time a try block does not have a catch handler for a particular exception, then the catch blocks of parent try block are inspected for that exception, if match is found that that catch block executes.

If neither catch block nor parent catch block handles exception then the system generated message would be shown for the exception, similar to what we see when we don't handle exception.

Lets see the syntax first then we will discuss this with an example.

## Syntax of Nested try Catch

```
....
//Main try block
try {
statement 1;
statement 2;
   //try-catch block inside another try block
try {       statement 3;       statement 4;
      //try-catch block inside nested try block
try {          statement 5;          statement
6;
```

```
      }
      catch(Exception e2) {
//Exception Message
      }
   }
   catch(Exception e1) {
//Exception Message
   }

}
//Catch of Main(parent) try block
catch(Exception e3) {
//Exception Message }
....
```

# Nested Try Catch Example

Here we have deep (two level) nesting which means we have a try-catch block inside a nested try block. To make you understand better I have given the names to each try block in comments like try-block2, try-block3 etc.

This is how the structure is: try-block3 is inside try-block2 and try-block2 is inside main try-block, you can say that the main try-block is a grand parent of the tryblock3. Refer the explanation which is given at the end of this code.

```
class NestingDemo{
   public static void main(String args[]){
      //main try-block
try{
       //try-block2
try{
//try-block3
try{
             int arr[]= {1,2,3,4};
             /* I'm trying to display the value of
* an element which doesn't exist. The
* code should throw an exception
              */
             System.out.println(arr[10]);
```

```
            }catch(ArithmeticException e){
             System.out.print("Arithmetic Exception");
             System.out.println(" handled in try-block3");
}         }
        catch(ArithmeticException e){
           System.out.print("Arithmetic Exception");
           System.out.println(" handled in try-block2");
        }
}
    catch(ArithmeticException e3){
       System.out.print("Arithmetic Exception");
      System.out.println(" handled in main try-block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
       System.out.print("ArrayIndexOutOfBoundsException");
      System.out.println(" handled in main try-block");
    }
    catch(Exception e5){
       System.out.print("Exception");
       System.out.println(" handled in main try-block");
}
  }
}
```

Output:

```
ArrayIndexOutOfBoundsException handled in main try-block
```

As you can see that the try-block3. Since try-block3 is not handling this exception, the control then gets transferred to the parent try-block2 and looked for the catch handlers in tryblock2. Since the try-block2 is also not handling that exception, the control gets transferred to the main (grand parent) try-block where it found the appropriate catch block for exception. This is how the the nesting structure works.

ArrayIndexOutOfBoundsException occurred in the grand child

## Example 2: Nested try block

```
class Nest{
   public static void main(String args[]){
        //Parent try block
     try{
       //Child try block1
         try{
            System.out.println("Inside block1");
int b =45/0;
            System.out.println(b);
         }
         catch(ArithmeticException e1){
            System.out.println("Exception: e1");
         }
         //Child try block2
         try{
            System.out.println("Inside block2");
int b =45/0;
            System.out.println(b);
         }
         catch(ArrayIndexOutOfBoundsException e2){
            System.out.println("Exception: e2");
}
        System.out.println("Just other statement");
    }
    catch(ArithmeticException e3){
        System.out.println("Arithmetic Exception");
         System.out.println("Inside parent try catch block");
    }
    catch(ArrayIndexOutOfBoundsException e4){
       System.out.println("ArrayIndexOutOfBoundsException");
         System.out.println("Inside parent try catch block");
}
    catch(Exception e5){
       System.out.println("Exception");
         System.out.println("Inside parent try catch block");
    }
    System.out.println("Next statement..");
```

```
  }
}
```

**Output:**

```
Inside block1
Exception: e1
Inside block2
Arithmetic Exception
Inside parent try catch block
Next statement..
```

This is another example that shows how the nested try block works. You can see that there are two try-catch block inside main try block's body. I've marked them as block 1 and block 2 in above example.

**Block1:** I have divided an integer by zero and it caused an ArithmeticException, since the catch of block1 is handling ArithmeticException "Exception: e1" displayed.

**Block2:** In block2, ArithmeticException occurred but block 2 catch is only handling ArrayIndexOutOfBoundsException so in this case control jump to the Main try-catch(parent) body and checks for the ArithmeticException catch handler in parent catch blocks. Since catch of parent try block is handling this exception using generic Exception handler that handles all exceptions, the message "Inside parent try catch block" displayed as output.

**Parent try Catch block:** No exception occurred here so the "Next statement.." displayed.

The important point to note here is that whenever the child catch blocks are not handling any exception, the jumps to the parent catch blocks, if the exception is not handled there as well then the program will terminate abruptly showing system generated message.

# Java Finally block – Exception handling

A **finally block** contains all the crucial statements that must be executed whether exception occurs or not. The statements present in this block will always execute

regardless of whether exception occurs in try block or not such as closing a connection, stream etc.

## Syntax of Finally block

```
try {
    //Statements that may cause an exception
} catch
{
   //Handling exception
}
finally {
   //Statements to be executed }
```

## A Simple Example of finally block

Here you can see that the exception occurred in try block which has been handled in catch block, after that finally block got executed.

```
class Example
{
   public static void main(String args[]) {
try{
        int num=121/0;
        System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("Number should not be divided by zero");
}
      /* Finally block will always execute
       * even if there is no exception in try block
       */
finally{
        System.out.println("This is finally block");
}
      System.out.println("Out of try-catch-finally");
   }
}
```

**Output:**

```
Number should not be divided by zero
This is finally block
Out of try-catch-finally
```

## Few Important points regarding finally block

1. A finally block must be associated with a try block, you cannot use finally without a try block. You should place those statements in this block that must be executed always.
2. Finally block is optional, as we have seen in previous tutorials that a try-catch block is sufficient for exception handling, however if you place a finally block then it will always run after the execution of try block.

3. In normal case when there is no exception in try block then the finally block is executed after try block. However if an exception occurs then the catch block is executed before finally block.

4. An exception in the finally block, behaves exactly like any other exception.

5. The statements present in the **finally block** execute even if the try block contains control transfer statements like return, break or continue.

Lets see an example to see how finally works when return statement is present in try block:

## Another example of finally block and return statement

You can see that even though we have return statement in the method, the finally block still runs.

```
class JavaFinally
{
   public static void main(String args[])
   {
      System.out.println(JavaFinally.myMethod());
   }
   public static int myMethod()
```

```
   {
try {
        return 112;
      }
      finally {
        System.out.println("This is Finally block");
        System.out.println("Finally block ran even after return statement");
      }
   }
}
```

**Output of above program:**

```
This is Finally block
Finally block ran even after return statement 112
```

.

# Cases when the finally block doesn't execute

The circumstances that prevent execution of the code in a finally block are:
– The death of a Thread
– Using of the System. exit() method.
– Due to an exception arising in the finally block.

# Finally and Close()

**close()** statement is used to close all the open streams in a program. Its a good practice to use close() inside finally block. Since finally block executes even if exception occurs so you can be sure that all input and output streams are closed properly regardless of whether the exception occurs or not.

For example:

```
.... try{
    OutputStream osf = new FileOutputStream( "filename" );
    OutputStream osb = new BufferedOutputStream(opf);
ObjectOutput op = new ObjectOutputStream(osb);     try{
       output.writeObject(writableObject);
    }     finally{
op.close();
    }
}
```

```
catch(IOException e1){
System.out.println(e1); }
...
```

## Finally block without catch

A try-finally block is possible without catch block. Which means a try block can be used with finally without having a catch block.

```
...
InputStream input = null;
try {
    input = new FileInputStream("inputfile.txt");
}  finally
{
    if (input != null) {
try {
in.close();
       }catch (IOException exp) {
           System.out.println(exp);
        }
    }
}
...
```

## Finally block and System.exit()

**System.exit()** statement behaves differently than **return statement**. Unlike return statement whenever System.exit() gets called in try block then **Finally block** doesn't execute. Here is a code snippet that demonstrate the same:

```
....
try {
   //try block
   System.out.println("Inside try block");
   System.exit(0)
}
catch (Exception exp) {
System.out.println(exp);
}
finally {
```

```
   System.out.println("Java finally block");
}
....
```

In the above example if the **System.exit(0)** gets called without any exception then finally won't execute. However if any exception occurs while calling **System.exit(0)** then finally block will be executed.

# try-catch-finally block

- Either a try statement should be associated with a catch block or with finally.
- Since catch performs exception handling and finally performs the cleanup, the best approach is to use both of them.

**Syntax:**

```
try {
     //statements that may cause an exception
}
catch (…) {
     //error handling code
}
finally {
    //statements to be executed
}
```

## Examples of Try catch finally blocks

**Example 1:** The following example demonstrate the working of finally block when no exception occurs in try block

```
class Example1{
  public static void main(String args[]){
try{
       System.out.println("First statement of try block");
int num=45/3;
       System.out.println(num);
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("ArrayIndexOutOfBoundsException");
```

```
    }
finally{
       System.out.println("finally block");
    }
    System.out.println("Out of try-catch-finally block");
}
}
```

**Output:**

```
First statement of try block
15
finally block
Out of try-catch-finally block
```

**Example 2:** This example shows the working of finally block when an exception occurs in try block but is not handled in the catch block:

```
class Example2{
   public static void main(String args[]){
try{
        System.out.println("First statement of try block");
int num=45/0;
        System.out.println(num);
     }
     catch(ArrayIndexOutOfBoundsException e){
        System.out.println("ArrayIndexOutOfBoundsException");
     }
finally{
        System.out.println("finally block");
     }
     System.out.println("Out of try-catch-finally block");
   }
}
```

**Output:**

```
First statement of try block finally
block
Exception in thread "main" java.lang.ArithmeticException: / by zero
at beginnersbook.com.Example2.main(Details.java:6)
```

As you can see that the system generated exception message is shown but before that the finally block successfully executed.

**Example 3**: When exception occurs in try block and handled properly in catch block

```
class Example3{
   public static void main(String args[]){
try{
         System.out.println("First statement of try block");
         int num=45/0;
         System.out.println(num);
      }
      catch(ArithmeticException e){
         System.out.println("ArithmeticException");
      }
finally{
         System.out.println("finally block");
      }
      System.out.println("Out of try-catch-finally block");
   }
}
```

**Output:**

```
First statement of try block
ArithmeticException finally
block
Out of try-catch-finally block
```

# How to throw exception in java with example

In Java we have already defined exception classes such as ArithmeticException, NullPointerException, ArrayIndexOutOfBounds exception etc. These exceptions are set to trigger on different-2 conditions. For example when we divide a number by zero, this triggers ArithmeticException, when we try to access the array element out of its bounds then we get ArrayIndexOutOfBoundsException.

We can define our own set of conditions or rules and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException when we divide number by 5, or any other numbers, what we need to do is just set the condition and throw any exception using throw keyword. Throw keyword

can also be used for throwing custom exceptions, I have covered that in a separate tutorial, see Custom Exceptions in Java.

**Syntax of throw keyword:**

```
throw new exception_class("error message");
```

For example:

```
throw new ArithmeticException("dividing a number by 5 is not allowed in this program");
```

# Example of throw keyword

Lets say we have a requirement where we we need to only register the students when their age is less than 12 and weight is less than 40, if any of the condition is not met then the user should get an ArithmeticException with the warning message “Student is not eligible for registration”. We have implemented the logic by placing the code in the method that checks student eligibility if the entered student age and weight doesn’t met the criteria then we throw the exception using throw keyword.

```
/* In this program we are checking the Student age
* if the student age<12 and weight <40 then our program
* should return that the student is not eligible for registration.
 */
public class ThrowExample {
   static    void    checkEligibilty(int    stuage,    int    stuweight){
if(stuage<12 && stuweight<40) {
         throw new ArithmeticException("Student is not eligible for registration");
      }
else {
         System.out.println("Student Entry is Valid!!");
      }
   }
   public static void main(String args[]){
     System.out.println("Welcome to the Registration process!!");
checkEligibilty(10, 39);
     System.out.println("Have a nice day..");
 }
```

```
}
```

Output:

```
Welcome to the Registration process!!Exception in thread "main"
java.lang.ArithmeticException: Student is not eligible for registration at
beginnersbook.com.ThrowExample.checkEligibilty(ThrowExample.java:9)
at beginnersbook.com.ThrowExample.main(ThrowExample.java:18)
```

In the above example we have throw an unchecked exception, same way we can throw uncheckedand user-defined exception as well.

# Throws clause in java – Exception handling

As we know that there are two types of exception checked and unchecked. Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.

On the other hand unchecked exception (Runtime) doesn't get checked during compilation. **Throws keyword** is used for handling checked exceptions . By using throws we can declare multiple exceptions in one go.

## What is the need of having throws keyword when you can handle exception using try-catch?

Well, thats a valid question. We already know we can handle exceptions using try-catch block.

The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch. For example:

Lets say we have a method myMethod() that has statements that can throw either ArithmeticException or NullPointerException, in this case you can use try-catch as shown below:

```
public void myMethod()
```

```
{   try
{
    // Statements that might throw an exception
  }   catch (ArithmeticException e)
{     // Exception handling
statements
  }
  catch (NullPointerException e) {
// Exception handling statements
  }
}
```

But suppose you have several such methods that can cause exceptions, in that case it would be tedious to write these try-catch for each method. The code will become unnecessary long and will be less-readable.

One way to overcome this problem is by using throws like this: declare the exceptions in the method signature using throws and handle the exceptions where you are calling this method by using try-catch.
Another advantage of using this approach is that you will be forced to handle the exception when you call this method, all the exceptions that are declared using throws, must be handled where you are calling this method else you will get compilation error.

```
public void myMethod() throws ArithmeticException, NullPointerException {
  // Statements that might throw an exception
}
public static void main(String args[]) {
try {
    myMethod();
  }   catch (ArithmeticException e)
{     // Exception handling
statements
  }   catch (NullPointerException e)
{     // Exception handling
statements
  }
}
```

## Example of throws Keyword

In this example the method myMethod() is throwing two **checked exceptions** so we have declared these exceptions in the method signature using **throws** Keyword. If we do not declare these exceptions then the program will throw a compilation error.

```
import java.io.*; class
ThrowExample {
  void myMethod(int num)throws IOException, ClassNotFoundException{
if(num==1)
        throw new IOException("IOException Occurred");
else
        throw new ClassNotFoundException("ClassNotFoundException");
}
}
public class Example1{
  public static void main(String args[]){
try{
```

```
     ThrowExample obj=new ThrowExample();
obj.myMethod(1);     }catch(Exception ex){
     System.out.println(ex);
    }
  }
}
```

Output:

```
java.io.IOException: IOException Occurred
```

# User defined exception in java

In java we have already defined, exception classes such as ArithmeticException, NullPointerException etc. These exceptions are already set to trigger on predefined conditions such as when you divide a number by zero it triggers ArithmeticException, In the last tutorial we learnt how to throw these exceptions explicitly based on your conditions using throw keyword.

In java we can create our own exception class and throw that exception using throw keyword. These exceptions are known as **userdefined** or **custom** exceptions. In this tutorial we will see how to create your own custom exception and throw it on a particular condition.

## Example of User defined exception in Java

```
/* This is my Exception class, I have named it MyException
* you can give any name, just remember that it should
* extend Exception class
 */
class MyException extends Exception{
   String str1;
```

```
   /* Constructor of custom exception class
* here I am copying the message that we are passing while
* throwing the exception to a string and then displaying      * that string along
  with the message.
    */
   MyException(String str2) {
       str1=str2;
   }    public String
toString(){
       return ("MyException Occurred: "+str1) ;
   }
}
class Example1{
   public static void main(String args[]){
       try{
              System.out.println("Starting of try block");
       // I'm throwing the custom exception using throw
       throw new MyException("This is My error Message");
       }
       catch(MyException exp){
              System.out.println("Catch Block") ;
              System.out.println(exp) ;
       }
   }
}
```

**Output:**

```
Starting of try block
Catch Block
MyException Occurred: This is My error Message
```

**Explanation:**

You can see that while throwing custom exception I gave a string in parenthesis ( throw new MyException("This is My error Message");). That's why we have a parameterized constructor (with a String parameter) in my custom exception class.

**Notes:**

1. User-defined exception must extend Exception class.
2. The exception is thrown using throw keyword.

## Another Example of Custom Exception

In this example we are throwing an exception from a method. In this case we should use throws clause in the method signature otherwise you will get compilation error saying that “unhandled exception in method”. To understand how throws clause works, refer this guide: throws keyword in java.

```
class InvalidProductException extends Exception
{
    public InvalidProductException(String s)
{
        // Call constructor of parent Exception
super(s);
    }
}

public class Example1
{
   void productCheck(int weight) throws InvalidProductException{
      if(weight<100){
            throw new InvalidProductException("Product Invalid");
      }
   }
    public static void main(String args[])
    {
      Example1 obj = new Example1();
try          {
             obj.productCheck(60);
        }
        catch (InvalidProductException ex)
        {
             System.out.println("Caught the exception");
```

```
            System.out.println(ex.getMessage());
        }
    }
}
```

**Output:**

```
Caught the exception Product
Invalid
```

# Java Exception Handling examples

BY CHAITANYA SINGH | FILED UNDER: EXCEPTION HANDLING

In this tutorial, we will see examples of few frequently used exceptions. If you looking for exception handling tutorial refer this complete guide: Exception handling in Java.

# Example 1: Arithmetic exception

Class: Java.lang.ArithmeticException
This is a built-in-class present in java.lang package. This exception occurs when an integer is divided by zero.

```
class Example1
{
   public static void main(String args[])
   {
      try{
         int num1=30, num2=0;
         int output=num1/num2;
         System.out.println ("Result: "+output);
      }
      catch(ArithmeticException e){
         System.out.println ("You Shouldn't divide a number by zero");
      }
   }
}
```

**Output of above program:**

```
You Shouldn't divide a number by zero
```

**Explanation:** In the above example I've divided an integer by a zero and because of this ArithmeticException is thrown.

# Example 2: ArrayIndexOutOfBounds Exception

Class: Java.lang.ArrayIndexOutOfBoundsException
This exception occurs when you try to access the array index which does not exist. For example, If array is having only 5 elements and we are trying to display 7th element then it would throw this exception.

```
class ExceptionDemo2
{
   public static void main(String args[])
   {
try{
        int a[]=new int[10];
        //Array has only 10 elements
a[11] = 9;
```

```
      }
      catch(ArrayIndexOutOfBoundsException e){
         System.out.println ("ArrayIndexOutOfBounds");
      }
   }
}
```

**Output:**

```
ArrayIndexOutOfBounds
```

In the above example the array is initialized to store only 10 elements indexes 0 to 9. Since we are try to access element of index 11, the program is throwing this exception.

## Example 3: NumberFormat Exception

Class: Java.lang.NumberFormatException

This exception occurs when a string is parsed to any numeric variable.

For example, the statement `int num=Integer.parseInt ("XYZ");` would throw `NumberFormatException` because `String` "XYZ" cannot be parsed to int.

```
class ExceptionDemo3
{
   public static void main(String args[])
   {
      try{
         int num=Integer.parseInt ("XYZ") ;
         System.out.println(num);
      }catch(NumberFormatException e){
          System.out.println("Number format exception occurred");
       }
   }
}
```

Output:

```
Number format exception occurred
```

# Example 4: StringIndexOutOfBound Exception

Class: `Java.lang.StringIndexOutOfBoundsException`

- An object of this class gets created whenever an index is invoked of a string, which is not in the range.
- Each character of a string object is stored in a particular index starting from 0.
- To get a character present in a particular index of a string we can use a method charAt(int) of java.lang.String where int argument is the index. E.g.

```
class ExceptionDemo4
{
   public static void main(String args[])
   {
try{
        String str="beginnersbook";
System.out.println(str.length());;
        char c = str.charAt(0);
c = str.charAt(40);
System.out.println(c);
      }catch(StringIndexOutOfBoundsException e){
         System.out.println("StringIndexOutOfBoundsException!!");
}
   }
```

```
}
```

Output:

```
13
StringIndexOutOfBoundsException!!
```

Exception occurred because the referenced index was not present in the String.

# Example 5: NullPointer Exception

Class: Java.lang.NullPointer Exception

An object of this class gets created whenever a member is invoked with a “null” object.

```
class Exception2
{
   public static void main(String args[])
   {
      try{
            String str=null;
            System.out.println (str.length());
      }
        catch(NullPointerException e){
            System.out.println("NullPointerException..");
      }
   }
}
```

Output:

```
NullPointerException..
```

Here, length() is the function, which should be used on an object. However in the above example String object str is null so it is not an object due to which NullPointerException occurred.

# CHAPTER-11

# Multithreading in java with examples

Before we talk about **multithreading**, let's discuss threads. A thread is a lightweight smallest part of a process that can run concurrently with the other parts(other threads) of the same process. Threads are independent because they all have separate path of execution that's the reason if an exception occurs in one thread, it doesn't affect the execution of other threads. All threads of a process share the common memory. **The process of executing multiple threads simultaneously is known as multithreading.**

Let's summarize the discussion in points:

1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.

2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.

3. A thread can be in one of the following states:

NEW – A thread that has not yet started is in this state.
RUNNABLE – A thread executing in the Java virtual machine is in this state.
BLOCKED – A thread that is blocked waiting for a monitor lock is in this state.
WAITING – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
TIMED_WAITING – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
TERMINATED – A thread that has exited is in this state.
A thread can be in only one state at a given point in time.

## Multitasking vs Multithreading vs Multiprocessing vs parallel processing

If you are new to java you may get confused among these terms as they are used quite frequently when we discuss multithreading. Let's talk about them in brief.
**Multitasking:** Ability to execute more than one task at the same time is known as multitasking.

**Multithreading:** We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

**Multiprocessing:** It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

**Parallel Processing:** It refers to the utilization of multiple CPUs in a single computer system.

## Creating a thread in Java

There are two ways to create a thread in Java:
1) By extending Thread class.
2) By implementing Runnable interface.

Before we begin with the programs(code) of creating threads, let's have a look at these methods of Thread class. We have used few of these methods in the example below.

- `getName()`: It is used for Obtaining a thread's name
- `getPriority()`: Obtain a thread's priority
- `isAlive()`: Determine if a thread is still running
- `join()`: Wait for a thread to terminate
- `run()`: Entry point for the thread
- `sleep()`: suspend a thread for a period of time

- start(): start a thread by calling its run() method

## Method 1: Thread creation by extending Thread class **Example 1:**

```
class MultithreadingDemo extends Thread{
  public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
```

```
     obj.start();
  }
}
```

## Output:

```
My thread is in running state.
```

## Example 2:

```
class Count extends Thread
{
   Count()
{
     super("my extending thread");
     System.out.println("my thread created" + this);
start();
   }
   public void run()
   {
try
{
        for (int i=0 ;i<10;i++)
        {
           System.out.println("Printing the count " + i);
           Thread.sleep(1000);
        }
}
     catch(InterruptedException e)
     {
        System.out.println("my thread interrupted");
}
     System.out.println("My thread run is over" );
   }
}
class ExtendingExample
{
   public static void main(String args[])
   {
      Count cnt = new Count();
try       {
         while(cnt.isAlive())
         {
```

```
            System.out.println("Main thread will be alive till the child thread is
live");
            Thread.sleep(1500);
          }
}
      catch(InterruptedException e)
      {
        System.out.println("Main thread interrupted");
}
      System.out.println("Main thread's run is over" );
   }
}
```

**Output:**

```
my thread createdThread[my runnable thread,5,main] Main
thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live
Printing the count 2
Main thread will be alive till the child thread is live
Printing the count 3
Printing the count 4
Main thread will be alive till the child thread is live Printing
the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live Printing
the count 8
Main thread will be alive till the child thread is live
Printing the count 9 mythread run is over Main thread
run is over
```

## Method 2: Thread creation by implementing Runnable Interface A

**Simple Example**

```
class MultithreadingDemo implements Runnable{
public void run(){
    System.out.println("My thread is in running state.");
  }
  public static void main(String args[]){
     MultithreadingDemo obj=new MultithreadingDemo();
     Thread tobj =new Thread(obj);
tobj.start();
 }
}
```

**Output:**

```
My thread is in running state.
```

## Example Program 2:

Observe the output of this program and try to understand what is happening in this program. If you have understood the usage of each thread method then you should not face any issue, understanding this example.

```
class Count implements Runnable
{
   Thread mythread ;
   Count()
   {
      mythread = new Thread(this, "my runnable thread");
System.out.println("my thread created" + mythread);
mythread.start();
   }
   public void run()
   {
try
{
        for (int i=0 ;i<10;i++)
        {
          System.out.println("Printing the count " + i);
          Thread.sleep(1000);
        }
}
     catch(InterruptedException e)
```

```
     {
        System.out.println("my thread interrupted");
}
     System.out.println("mythread run is over" );
   }
}
class RunnableExample
{
    public static void main(String args[])
    {
       Count cnt = new Count();
try         {
          while(cnt.mythread.isAlive())
          {
            System.out.println("Main thread will be alive till the child thread is
live");
            Thread.sleep(1500);
          }
}
       catch(InterruptedException e)
       {
          System.out.println("Main thread interrupted");
}
       System.out.println("Main thread run is over" );
    }
}
```

## Output:

```
my thread createdThread[my runnable thread,5,main] Main
thread will be alive till the child thread is live
Printing the count 0
Printing the count 1
Main thread will be alive till the child thread is live Printing
the count 2
Main thread will be alive till the child thread is live
Printing the count 3
```

```
Printing the count 4
Main thread will be alive till the child thread is live Printing
the count 5
Main thread will be alive till the child thread is live
Printing the count 6
Printing the count 7
Main thread will be alive till the child thread is live Printing
the count 8
Main thread will be alive till the child thread is live
Printing the count 9 mythread run is over Main thread
run is over
```

# Thread priorities

- Thread priorities are the integers which decide how one thread should be treated with respect to the others.
- Thread priority decides when to switch from one running thread to another, process is called context switching
- A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
- A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
- To set the priority of the thread `setPriority()` method is used which is a method of the class `Thread` Class.
- In place of defining the priority in integers, we can use `MIN_PRIORITY`, `NORM_PRIORITY` or `MAX_PRIORITY`.

# Methods: isAlive() and join()

- In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
- To know whether the thread has finished we can call `isAlive()` on the thread which returns true if the thread is not finished.
- Another way to achieve this by using `join()` method, this method when called from the parent thread makes parent thread wait till child thread terminates.
- These methods are defined in the `Thread` class.
- We have used isAlive() method in the above examples too.

# Synchronization

- Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
- When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
- To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
- When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

```
Synchronized(object)
{
    // statement to be synchronized }
```

# Inter-thread Communication

We have few methods through which java threads can communicate with each other. These methods are wait(), notify(), notifyAll(). All these methods can only be called from within a synchronized method.
1) To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor. 2) wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
3) notify() wakes up the first thread that called wait() on the same object. notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

# Thread join() method in Java with example

The join() method is used to hold the execution of currently running thread until the specified thread is dead(finished execution). In this tutorial we will discuss the purpose and use of join() method with examples.

## Why we use join() method?

In normal circumstances we generally have more than one thread, thread scheduler schedules the threads, which does not guarantee the order of execution of threads.
For example lets have a look at the following code:

## Without using join()

Here we have three threads th1, th2 and th3. Even though we have started the threads in a sequential manner the thread scheduler does not start and end them in the specified order. Everytime you run this code, you may get a different result each time. **So the question is: How can we make sure that the threads executes in a particular order. The Answer is: By using join() method appropriately.**

```
public class JoinExample2 {
   public static void main(String[] args) {
      Thread th1 = new Thread(new MyClass2(), "th1");
      Thread th2 = new Thread(new MyClass2(), "th2");
      Thread th3 = new Thread(new MyClass2(), "th3");

th1.start();
th2.start();
th3.start();
   }
}
class MyClass2 implements Runnable{


    @Override
    public void run() {
       Thread t = Thread.currentThread();
        System.out.println("Thread started: "+t.getName());
try {
```

```
            Thread.sleep(4000);
        } catch (InterruptedException ie) {
ie.printStackTrace();
        }
        System.out.println("Thread ended: "+t.getName());
    }
}
```

Output:

```
Thread started: th1
Thread started: th3
Thread started: th2
Thread ended: th1
Thread ended: th3
Thread ended: th2
```

Lets have a look at the another code where we are using the join() method.

# The same example with join()

Lets say our requirement is to execute them in the order of first, second and third. We can do so by using join() method appropriately.

```
public class JoinExample {
   public static void main(String[] args) {
      Thread th1 = new Thread(new MyClass(), "th1");
      Thread th2 = new Thread(new MyClass(), "th2");
      Thread th3 = new Thread(new MyClass(), "th3");

      // Start first thread immediately
      th1.start();

      /* Start second thread(th2) once first thread(th1)
*      is dead        */
try {
          th1.join();
      } catch (InterruptedException ie) {           ie.printStackTrace();
        }
      th2.start();

      /* Start third thread(th3) once second thread(th2)
*      is dead        */
try {
          th2.join();
      } catch (InterruptedException ie) {            ie.printStackTrace();
```

```
        }
      th3.start();

      // Displaying a message once third thread is dead       try {
          th3.join();
      } catch (InterruptedException ie) {             ie.printStackTrace();
        }
      System.out.println("All three threads have finished execution");
   }
}
class MyClass implements Runnable{

    @Override     public void run() {
       Thread t = Thread.currentThread();
        System.out.println("Thread started: "+t.getName());
try {
            Thread.sleep(4000);
        } catch (InterruptedException ie) {
ie.printStackTrace();
        }
        System.out.println("Thread ended: "+t.getName());

    }
}
```

Output:

```
Thread started: th1 Thread
ended: th1
Thread started: th2 Thread
ended: th2
Thread started: th3
Thread ended: th3
All three threads have finished execution
```

In this example we have used the join() method in such a way that our threads execute in the specified order.