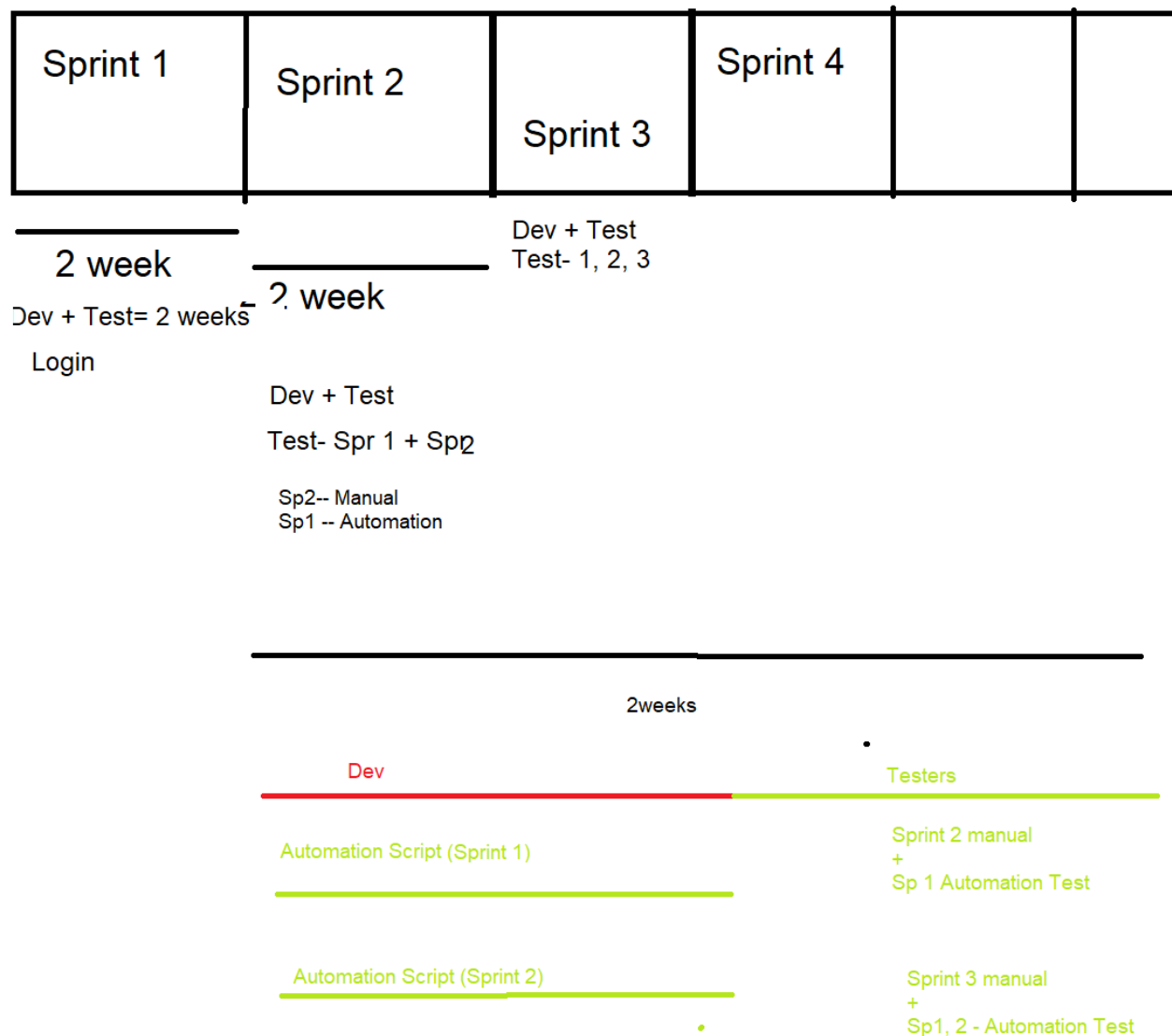


Automation Testing: A testing in which after the triggering (execution of code) the content over the application acts on the basis of the code written by automation tester.

Creation of a set of code which helps to reduce the time required for testing is called **Automation testing**.



Need of Automation:

1. Time required to complete the task is fixed hence we have to complete the whole task in the same time of sprint.
2. To avoid the human errors.
3. Whenever we want to execute a set of requirements multiple times we require automation.
4. A proper report generation after the execution.
5. End to End coverage can be achieved through automation in very less interval of time.
6. Overall performance to get improves then we should have automation.

How to achieve automation:

To perform the automation we require a set of instruction to be provided to the machine so that a particular action get performs.

Set of actions to be written in a language that is Java.

Java is the one because:

Principle of java- *"Write once and run anywhere"*

Qualities of java:

- a. Portable language.
- b. Simple.
- c. Secure.
- d. Multithreaded language.
- e. Robust.
- f. Garbage collector
- g. OOP (Object oriented programming language)

Risk involved in Automation testing:

1. Tester should have good programming knowledge.
2. 100% automation is not possible. – We always prioritize the scenarios based on their functionality and then automate them.

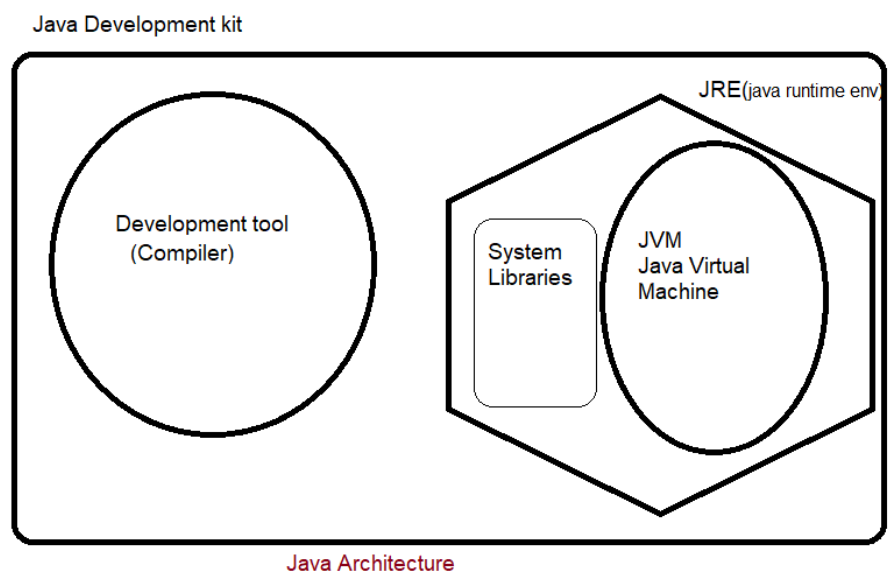
There are 3 flavor of java:

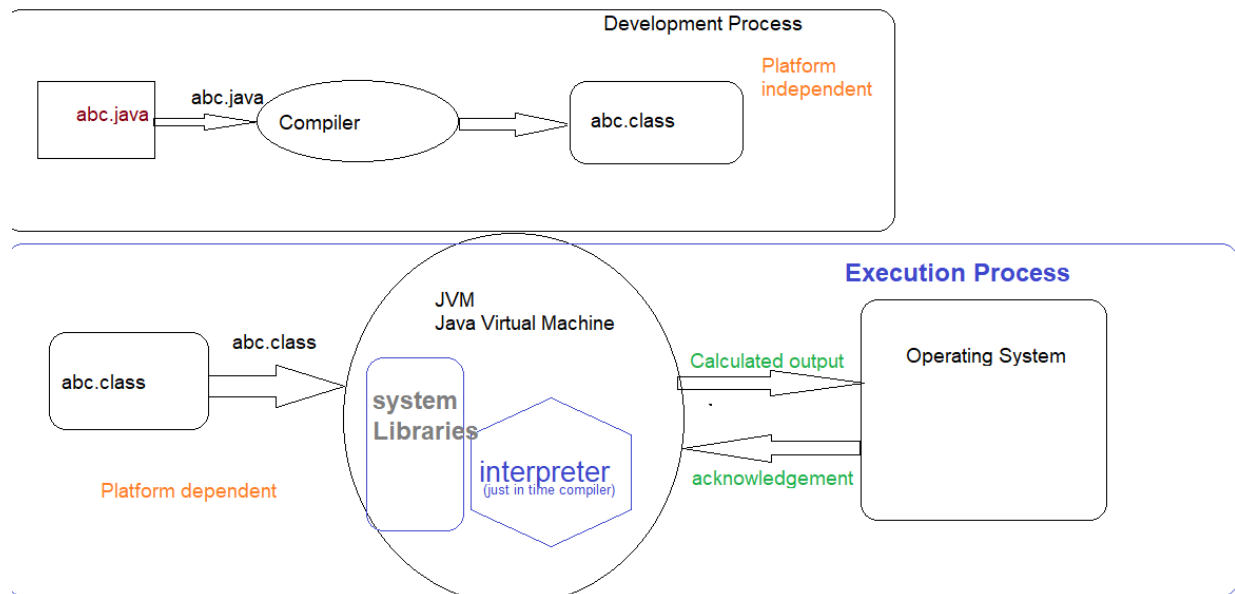
1. Java SE (Standard Edition) - **Core java**.
2. Java EE (Enterprise Edition) – Advance Java
3. Java ME (Micro Edition)

Error caused while compiling is called **compilation error**.

JDK = JRE+ Dev tool

JRE = JVM + System libraries





Compiler: It is software which is used to convert the .java file to .class file. Its major role is to check whether the java statement is syntactically correct or not according to the java protocol.

JVM: It is a Java virtual machine which basically responsible to execute the program line by line and provide the analytical output to OS. JVM is responsible to execute the program in which it takes the encoded or converted format from interpreter and then perform the action which makes the performance to get improve.

In order to execute a java program we are using an integrated development environment which is known as eclipse. Eclipse–IDE

To install the eclipse in the system :

Step 1 : go to <https://www.eclipse.org/downloads> and download the eclipse software.

Step 2: Click on download button available in the image:

Step 3: After downloading double click on the file and install it.

Step 4: We have to select eclipse IDE for java developers.

Class, Objects, Variable and methods

Class: It is a blue print which defines the complete plan of action. It is basically collection of objects.

Inside a class we define number of variables, methods and objects in a planned way.

Object: Object is an entity which comprises of State (Characteristic) and behavior (Actions).

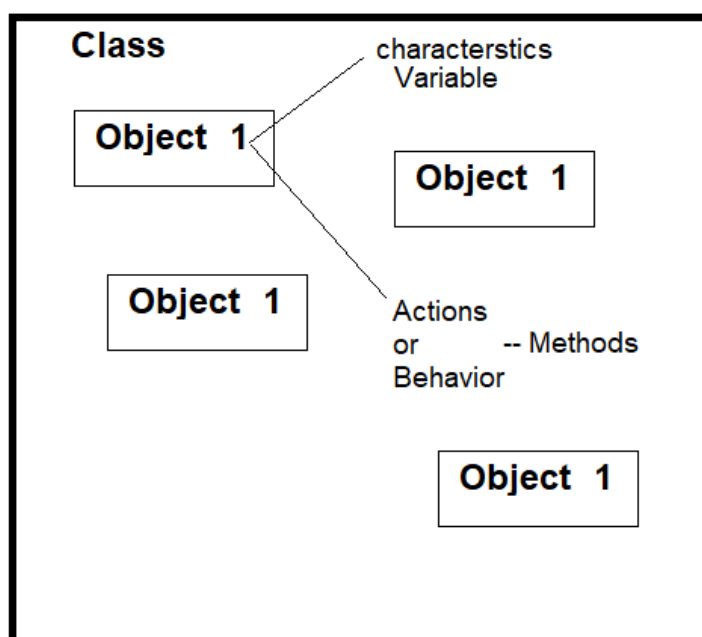
In a java program whenever we are creating an object which means it consume some part of memory which helps to store variables.

State refers to variable and behavior refers to methods.

Variable: These are the one which describes the characterstic or state.

Method: These are the one which describes the action which are going to perform on object.

Package: In java we can define a specific category with the help of a package which contains numerous numbers of classes in it depends on number of plan.



Data type

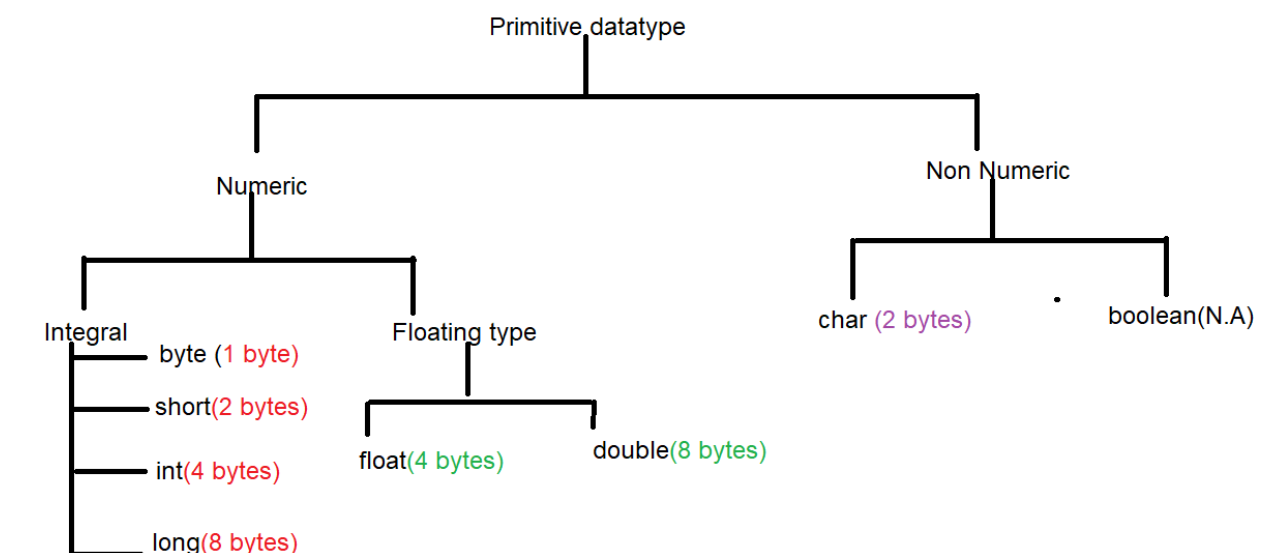
In Java every single variable has some type and that type should from different category i.e numeric, alphabets etc. so in order to define a variable we should provide the type of variable that is data type.

Java is strongly typed language.

We have different types of data type those are:

a. Primitive data type.

b. Non Primitive data type.



Numeric data type :

a. Integral data type:

i. byte: size of byte is 1 byte.

The maximum number will be 127.

The minimum number is -128.

Range : -128 to +127 or Range: -2^7 to $2^7 - 1$

ii. **short**: size of short is 2 bytes

The maximum number will be 32767.

The minimum number is -32768.

Range : -32768to +32767

Range: -2^{15} to $2^{15} - 1$

- iii. `int` : size of `int` is 4 bytes

The maximum number will be 2147483647.

The minimum number is -2147483648.

Range : -2147483648 to +2147483647

Range: -2^{31} to $2^{31} - 1$

iv. long : size of long is 8 bytes

Range: -2^{63} to $2^{63} - 1$

Note: In java for any numeric value it accepts it as a integer (int).

b) floating type :

v. float: size of float is 4 bytes.

Range : 1.4×10^{-45} to 3.4×10^{38}

Ex.

```
float f = 44555.565f;
```

```
float g = 0.6565656f;
```

[illegible]

vi. double : size of double is 8 bytes

Range: 4.9×10^{-324} to 1.79×10^{308}

example:

```
double d =  
6644464423265556565565655556555655565556565.565655656;
```

```
double e = 5656556.656;
```

Float Vs double:

If we want to represent a value with the accuracy of 5 to 6 decimal places then we should use float and if we want to go beyond 14-15 then we should prefer double.

Size of float is 4 bytes and size of double is 8 bytes.

F suffix is required to represent float value but no suffix is required for double value.

Note : By default any decimal value will be considered as double in java where as any numeric value will be considered as integer(int).

Non Numeric data type

i. char : Size of char is 2 bytes.

The number of words we are having in char data type are 65536.

Example: char c = 'a';

```
char d = '1';
```

```
char e = '#';
```


ii.boolean: A boolean can have only two values 'true' and 'false'. Since it contains only two possible values we don't have any memory allocated to it.

Example:

```
boolean b = true;  
boolean f = false;
```

Non Primitive data type:

A data type which represents all the classes which actually doesn't have any specific memory allocation they get adjust themselves dynamically based on the data available in it.

All the variables of classes are non primitive data type.

1. String : It is a class.

Example:

```
String ss = "Value of String";  
System.out.println(ss);
```

Output:

Value of String

Note: If we write '+' between any data type(Primitive /String) then the resultant value will be the merging operation of String with data type.

Here '+' is an operator which is used between String and any other data type hence it is known as Concatenation operator. But if it is used between numbers then it is addition operator.

Example: int k = 10;

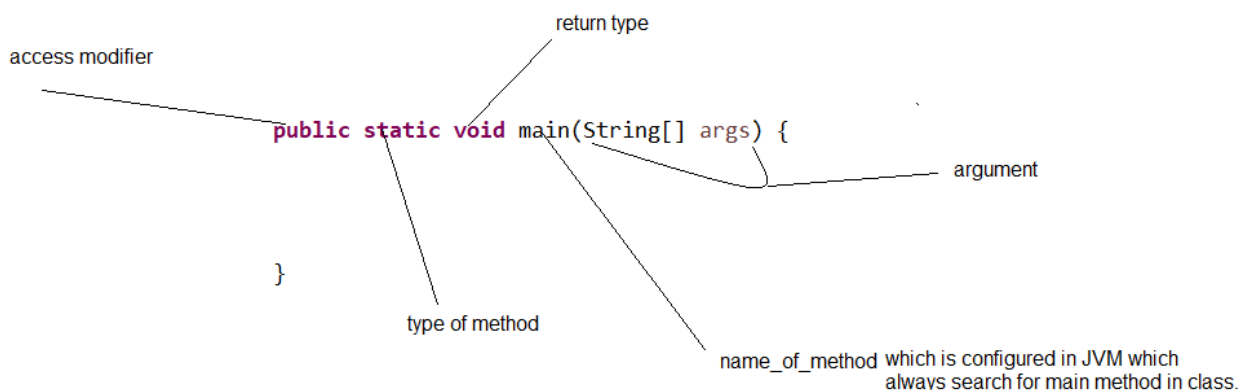
```
int l = 20;  
String m= "60";  
System.out.println(k+l+m);//3060  
  
System.out.println(k+m+l);//106020
```

Methods in Java

Method is the one which has set of code available in it and those get execute once we call it.

There are two types of methods:

a. main method.



b. Regular method.

i. static method

ii. non static method

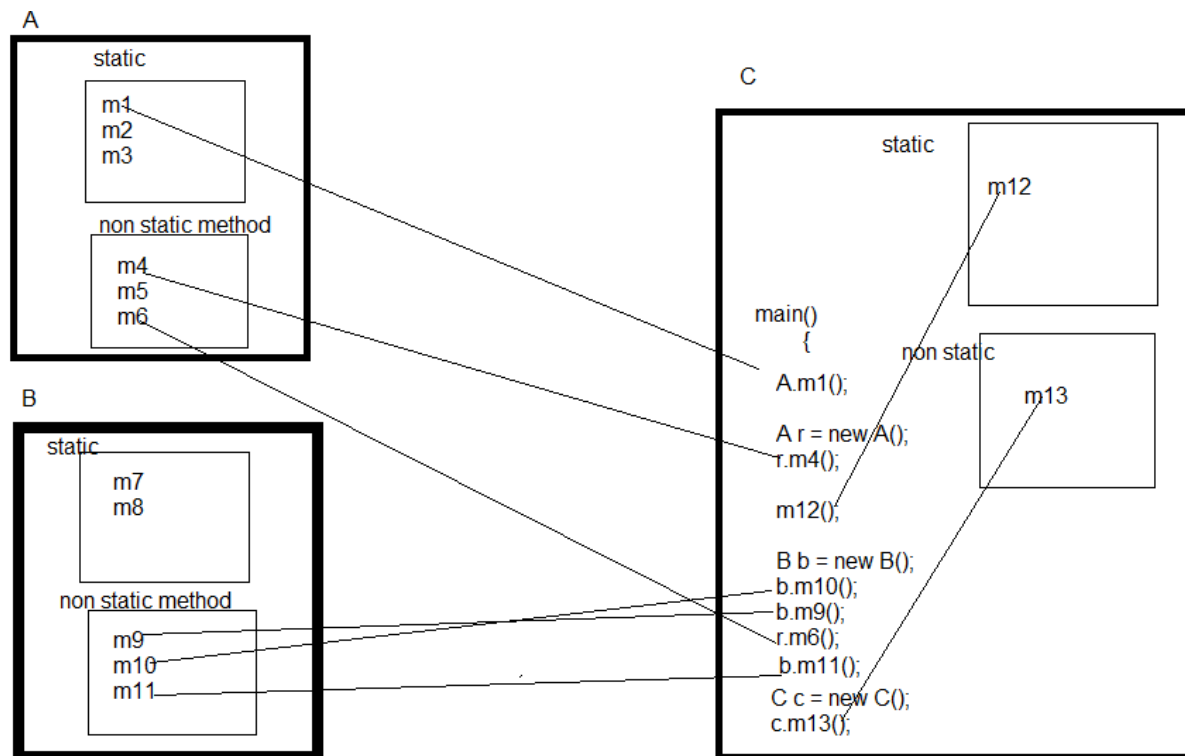
i. static method : It is a type of regular method which can be called through directly method name(if it is in same class where we are calling it) and If it is in another class then we have to call it by `classname.method name`.

ii. non static method: A method in which we don't write any static or non static keyword to define. To call a non static method we have to first create an object of the class which has that method and then we can call it by `object_reference_variable.method name`.

syntax:

`accessmodifier returntype nameofmethod()`

`{ }`



Process of execution:

1. JVM gets start.
2. Locate .class file.
3. Load .class file. – Static content memory allocation
4. Execute the main method - non static content memory allocation and de-allocate.
5. Unload .class file. – Static content memory de-allocation
6. JVM gets shutdown.

Java is nearly 100% object oriented programming language because we have some static content is also present in java which does not make it 100% OOP language.

Difference between Static and non static methods:

Sr.no.	Static methods	Non – Static methods
1	' static ' keyword is used while defining the method.	No keyword is used while defining the method.
2	We don't have to create an object to call static method.	We have to create an object for calling non static method.
3	Memory allocation gets done at the time of loading of .class file and de-allocation gets done when file gets unload.	Memory allocation gets done at the time of execution of main method and de-allocation gets done instantly after the completion of execution.
4.	Performance wise we should not prefer this.	Performance wise we should prefer this.

Categories of methods:

If we have static or non-static we have 4 categories applicable to them.

- Method without return type and without argument.
- Method without return type and with argument.
- Method with return type and without argument.
- Method with return type and with argument.

a. Method without return type and without argument.

```
public static void m1()  
{  
    System.out.println("without return type and  
without arguments");  
}
```

b. Method without return type and with argument.

```
public void m2(String a)
{
    System.out.println("without return type and with argument");

    System.out.println("The value of a is "+a);
}

public static void add(int a, int b, boolean d)
{
    System.out.println("mehthod without return and with three
arguments");

    int c = a+b;

    System.out.println("sum is :"+c);
}

public void m3(boolean b)
{
    System.out.println("before updation b value is :"+b);

    b = true;

    System.out.println("after updation b value is :"+b);
}

public static void main(String[] args) {

    CategoriesOfMethods com = new CategoriesOfMethods();

    com.m2("Velocity");

    add(50, 20, true);

    add(10, 12, false);

    com.m3(false);

}
```

Output:

```
without return type and with argument
The value of a is Velocity
mehthod without return and with three arguments
sum is :70
mehthod without return and with three arguments
sum is :22
before updation b value is :false
after updation b value is :true
```

c. Method with return and without arguments

```
public String m4()
{
    System.out.println("method with return and without
argument");

    return "Returning String Value";
}
```

Here in this method execution takes place in 2 steps:

Step1 : Logic which has been written inside the method get execute.

Step 2: wherever we call that method it replace itself by the value which we had returned.

```
public static void main(String[] args) {
    CategoriesOfMethods com = new CategoriesOfMethods();
    String retvalue = com.m4();
    System.out.println("m4 method is returning "+retvalue);
}
```

Output:

method with return and without argument

m4 method is returning Returning String Value.

d. method with return type and with arguments:

example:

```
public static int areaCalculator(int l, int b) {
    int area = l * b;
    return area;
}

public static void main(String[] args) {
    int land1area = areaCalculator(10, 15);
}
```

```

        int land2area = areaCalculator(20, 10);

        int land3area = areaCalculator(40, 50);

        int total = land1area + land2area + land3area;

        System.out.println("Total area is :" + total);
    }

```

Output:

Total area is :2350

Example 2

```

public static char m6(char c, int b)
{
    char x = 'a';
    char v = 'b';
    String g = "d";

    String h = g+x+v+c;

    System.out.println("h value is :"+h);

    return c;
}

public int m7(char f)
{
    System.out.println(f);

    return 10;
}

public static void main(String[] args) {

    CategoriesOfMethods com = new CategoriesOfMethods();

    System.out.println("*****");

    char l      = m6('q', 100);

    System.out.println("l value is :"+l);

    int k= com.m7(l);

```

Output:

```

*****
h value is :dabq
l value is :q
q

```

Calling of one method into another method:

Calling of static inside static and non static method:

For calling static method inside another static and nonstatic method directly we can call by methodname(If it is in the same class) or Classname.method if it is from the another class . Therefore it is same as we call any static method.

Calling of non static in Static and non static method:

Inside static method:

To call non static method inside a static method we need to create an object and call it by reference variable name.method name.

Inside non static method:

For calling a non static method inside a non static method we don't have to create an object because altogether at the end we have to create an object for the method in which we have called another non static method.

Note : If both non static methods are in the same class then to call a non static method inside another non static method we don't have to create an object here to call m6 inside m7 we don't have to create object because to m7 we need an object which is to be used for m6 also. But if both non static methods are in different classes then we have to create an object to call.

Variables in Java

To represent any quantity in a class we require a reference to represent it which is called as Variables.

There are 3 types of variable:

1. Static variable.
2. Non static variable.
3. Local variable.

1. Static variable:

Note : Static variable is a class level variable which gets define only at class level and it will be easily accessible to all the methods direct name of variable (If it is in the same class) or Classname.variable name if it is in different class.

2. non static variable or instance variable:

A variable which defines at class level without using static keyword is called non static variable.

It can be accessible to any method by creating an object for static and directly if the method is non static and the variable in same class. If non static variable is not in the same class then we have to create an object to access it like we have in method.

Note:

1. Non static variable value gets vary from object to object.
2. Static variable's value doesn't get change from object to object it get updated to every object whenever we update it it means static variable share its memory with every object.
3. Static and non static variables are also known as global variables because they are define at class level and can be accessible to all methods.

Note: Static method and variable can be accessible through object but it is NOT recommended.

If we don't assign any value to the static or non static variable then JVM automatically assign the default value in it. Default value concept is only applicable to static and non static it is not applicable for local variable. The default values are:

Int --- 0

Double – 0.0

String – null

Boolean - false


Char – ‘ ‘

3. Local variable : A variable which get define only inside the two curly braces (Except class curly braces). Local variable's scope is limited to the curly braces. Default value concept is not applicable to local variable.

this keyword in java: This keyword is used to access the global variable(static / non static) inside the non static area. This can be used by

this.variable name.

```
public class LocalVariables {  
    static int j = 20;  
    int z= 50;  
    public void m1()  
    {  
        int i =40;  
        int z = 80;  
        System.out.println(i);//40  
        System.out.println("Z local variable value is "+z);//80  
        System.out.println("Z value from this keyword is "+this.z);// non static z  
        variable 50  
        System.out.println("j value from this keyword is : "+this.j);  
        System.out.println(j);//20}  
}
```



Sr no	Static variable	Non static variable
1	It can be access through direct vaiable name/ class name.variable name.	It can be access through object only.
2	Its value doesn't get vary from object to object.	Its value get varies from object to object.
3	Memory allocation gets done at the time of class loading	Memory allocation gets done at the time of execution.
4.	Static keyword is used to define	No keyword is used to define.

Similarity between static and non static:

1. Both the variables are defined at class level hence these are also known as global variable.
2. Default value concept is applicable to both of them.

Conditional Statements

Operators in Java:

1. Arithmetic operator:

//1. Arithmetic operator

// +, -, *, /
// % - Modulus operator

Example:

```
public static void main(String[] args) {  
  
    int i = 10;  
    int j = 3;  
  
    int k = i/j;  
  
    int l = i%j;  
  
    System.out.println("k value is "+k);  
  
    System.out.println("l value is "+l);  
}
```

Output:

```
k value is 3  
l value is 1
```

if and else statement:

syntax:

```
if (Boolean condition)  
  
    {  
        //Action items when if condition get true  
    }  
else  
    {  
        //Action item when if condition get false  
    }
```

Example:

```
int i = 10;
int j = 3;

if (i<j)
{
    System.out.println("i is smaller than j");
}
else
{
    System.out.println("i is greater than j");
}
```

Output:

i is greater than j

2. Comparator operator:

```
// >- Greater than
// >= Greater than or equal to
// < -- Less than
// <= -- Less than or equal to
// == equal operator
```

3. Logical operator:

a. && (Logical AND operator): Logical AND operator gives true only when both the conditions are true.

```
int i =10;

int l = 20;

boolean m = (i>5) && (l<50);

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

Output: true

b. Logical OR operator: OR operator gives false only when both the condition becomes false.

// Logical OR operator

Example:

```
int i =10;
```

```
int l = 20;
```

```
boolean k = (i==50) || (i==10);
```

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

```
boolean n = (i==50) || (i==60);// false
```

c. Logical NOT operator: It gives the exact opposite Boolean value when it is applied to a condition.

example

```
int i =10;
```

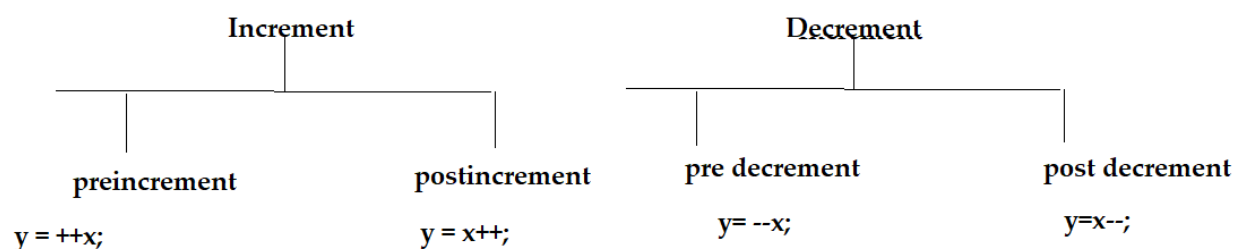
```
boolean o = !(i==10); // false
```

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

```
boolean p = (i!=10);// false
```

```
System.out.println("p value is :"+p);// false
```

d. Incremental and decremental operator:



	int x;			
	int y;			
Sr. no.	Expression	initial value of x	value of y	final value of x
1	y= ++x;	10	11	11
2	y=x++;	10	10	11
3	y= --x;	10	9	9
4	y= x--;	10	10	9

Use of else if:

Use the **else if** statement to specify a new condition if the first condition is **false**.

```
public static void main(String[] args) {  
    int i = 100;  
  
    if((0<=i) && (i<=20))  
    {  
        System.out.println("number is between 0 to 20");  
    }  
  
    else if((21<=i) && (i<=40))  
    {  
        System.out.println("number is between 21 to 40");  
    }  
  
    else if ((41<=i) && (i<=60))  
    {  
        System.out.println("number is between 41 to 60");  
    }  
    else  
    {  
        System.out.println("number is not in any range");  
    }  
  
}
```

Loops in Java

Loops: Whenever we want to execute a set of statements in a repeated manner then we require a loop. Using a loop, the complexity of the program gets reduced.

1. while loop:

the code in the while loop will run, over and over again,
up to the specified condition is true

Syntax:

while(Boolean condition)

 $\{$

Actions to be perform

}

Example:

```
while(i<10)
{
    System.out.println("Hello");

    i++;
}

System.out.println("Outside the loop");
```

Output:

```
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Outside the loop
```



```
int number = 12345;
while(number>0)
{
    number = number/10;
    System.out.println(number);
}
```

Output:

```
1234
123
12
1
0
```

2. do-while loop: If we want to execute the loop at least one time irrespective of any condition then we should use do-while loop.

Example:

```
public static void dowhileloop()
{
    int i = 0;
    do
    {
        System.out.println("hello");
    }
    while(i>2);
}

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

Output:

```
Hello
```

3. for loop: When you know exactly how many times you want to loop through a block of code, use the **for** loop instead of a **while** loop:

Syntax:

```
for (statement 1; statement 2; statement 3) {  
    // code block to be executed  
}
```

Statement 1 is executed (one time) before the execution of the code block.

Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

```
// syntax of for loop  
//          1          2, 5, 8          4, 7  
//          // for(any_java_statement; boolean_condition;  
any_java_statement)  
// {    3, 6, 9  
//      Actions  
//  
// }
```

Example:

```
public static void main(String[] args) {  
    int i=10;  
    for(System.out.println("hello"); i<15; i++ )  
    {  
        System.out.println("inside for loop");  
    }  
}
```

Output:

```
hello  
inside for loop  
inside for loop  
inside for loop  
inside for loop  
inside for loop
```

Pattern printing using for loop:

i/j	1	2	3	4	5
1	*				
2	*	*			
3	*	*	*		
4	*	*	*	*	
5	*	*	*	*	*

step 2	
i	j
1	1
2	1, 2
3	1, 2, 3
4	1, 2, 3, 4
5	1, 2, 3, 4, 5

Step 3	
i	j
1	j<=1
2	j<=2
3	j<=3;
4	j<=4;
5	j<=5;

Step 4

i	j
	j<=i;

Program:

```
public static void main(String[] args) {  
  
    for (int i = 0; i < 5; i++) {  
        for (int j = 0; j < 5; j++) {  
            if(j<=i)  
            {  
                System.out.print("*");  
            }  
            else  
            {  
                System.out.print(" ");  
            }  
            System.out.println();  
        }  
    }  
}
```

Output:

```
*  
**  
***  
****  
*****
```

i/j	1	2	3	4	5	6	7	8	9
1					*				
2				*	*	*			
3			*	*	*	*	*		
4		*	*	*	*	*	*	*	
5	*	*	*	*	*	*	*	*	*

Step 2		Step 3		Step 4	
i	j	i	j	i	j
1	5	1	$j \geq 5 \ \&\& \ j \leq 5$		
2	4, 5, 6	2	$j \geq 4 \ \&\& \ j \leq 6$		
3	3, 4, 5, 6, 7	3	$j \geq 3 \ \&\& \ j \leq 7$		
4	2, 3, 4, 5, 6, 7, 8	4	$j \geq 2 \ \&\& \ j \leq 8$		
5	1, 2, 3, 4, 5, 6, 7, 8, 9	5	$j \geq 1 \ \&\& \ j \leq 9$		$j \geq (6-i) \ \&\& \ j \leq 4+i$

```

for (int i = 1; i <= 5; i++) {
//column
    for (int j = 1; j <= 9; j++) {
        if(j>=(6-i) && j<=(4+i))
        {
            System.out.print("*");
        }
        else
        {
            System.out.print(" ");
        }
    }
    System.out.println();
}
}

```

Constructor in java

A specialize method which gets execute whenever we create an object is called constructor.

are There 2 types of constructors:

a. Default constructor: A constructor which gets define by JVM whenever we don't have any constructor available in the class and that constructor don't have any logic in it.

b. User define constructor: A constructor which got defined by user it can be any type (one arg, two arg, three arg etc) and we can have multiple constructor inside a class.

Rules of constructor:

1. Name of constructor and class name must be same.
2. Constructor cannot have return type if we try to have return type then it becomes a non static method.
3. Whenever we create an object automatically constructor get call of that class.
4. Constructor can accept arguments and we can have multiple constructor with different arguments inside a class.
5. Constructor can call another constructor inside a constructor but that statement must at the first line inside a constructor.

Use of constructor:

1. Whenever we want to execute logic on creation of object then we require constructor.
2. Constructor is used to initialize the data members or it is used to initialize the variables of the class.

Example:

```
public class CollegeData {  
  
    static String name = "COEP";  
  
    String studentname;  
    int m1marks;  
    int phymarks;  
  
    public CollegeData(String stuname, int maths, int physics)  
    {  
        studentname=stuname;  
        m1marks= maths;  
        phymarks=physics;  
    }  
  
    public static void main(String[] args) {  
        CollegeData s1 = new CollegeData("Johny", 50, 60);  
  
        CollegeData s2 = new CollegeData("Cesar", 30, 80);  
  
        CollegeData s3 = new CollegeData("Peter", 90, 80);  
  
        System.out.println(s3.studentname);  
    }  
}
```

Note:

this vs this():

this keyword is used to access the global variable(non static/ static) but this() is used to call a constructor inside another constructor.

Constructor with respect to Inheritance:

1. If we create an object inside the child class then by default the child class constructor calls parent's class constructor first then it execute child class constructor.

Example:

```
public class GrandCons {  
    public GrandCons()  
    {  
        System.out.println("Constructor of GandCons");  
    }  
}  
  
} public class ParentCons extends GrandCons {  
  
    public ParentCons()  
    {  
        System.out.println("Constructor from parent class");  
    }  
}  
  
}  
  
public class ChildCons extends ParentCons {  
  
    public ChildCons()  
    {  
        System.out.println("Constructor from Child class");  
    }  
  
    public static void main(String[] args) {  
        ChildCons cc = new ChildCons();  
  
    }  
}
```

Output:

```
Constructor of GandCons  
Constructor from parent class  
Constructor from Child class
```

2. Constructor doesn't follow inheritance.

Example:

```
public class A {  
    public A()  
    {  
        System.out.println("Constructor from A class");  
    }  
}  
  
} public class B extends A{  
  
    public B(int i)  
    {  
        System.out.println("Constructor from B class");  
    }  
}
```

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

```
    B b = new B();
```

```
// here we are getting an error which says provide an argument (int) to call the  
// constructor but we have no argument constructor in parent class which is not  
// accessible here through B's object hence constructor doesn't follow  
// inheritance.  
}
```

super() keyword:

3. To call a constructor of **parent** class we have to call it by **super()**. Constructor call inside another constructor should be at the first line and we can call only one constructor inside another constructor.

this(); vs **super();**

this is used to call the constructor from the same class whereas **super()** is used to call the constructor from parent class.

Note:

If there is no argument constructor available in parent class then that constructor get call directly or by default but if we have any argument constructor in parent class then child class constructor has to call that parent class constructor in it.

```
public class ParentCons extends GrandCons {
```

```
    public ParentCons(int i)
```

```
    {
```

```
        System.out.println("Constructor from parent class");
```

```
    }
```

```
}
```

```
public class ChildCons extends ParentCons {
```

```
    public ChildCons(){
```

```
        super(80);
```

```
        System.out.println("Constructor from Child class");
```

```
    }
```

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

```
    ChildCons cc = new ChildCons();}
```


Inheritance:

It is an oops concept

in which there is a relationship between the two classes and relationship establish by the help of **extends** keyword is called Inheritance.

With the help of inheritance we can access all the properties (Variables, methods etc) from parent class to the child class.

If we want to access the properties of parent class then we have to write keyword 'extends' while defining child class.

So In this child class(subclass) **extends** parent class(superclass).

- . With the help of inheritance we can achieve reusability property.
- . A class cannot extends another class in a cyclic manner hence **cyclic inheritance is not possible.**

Example:

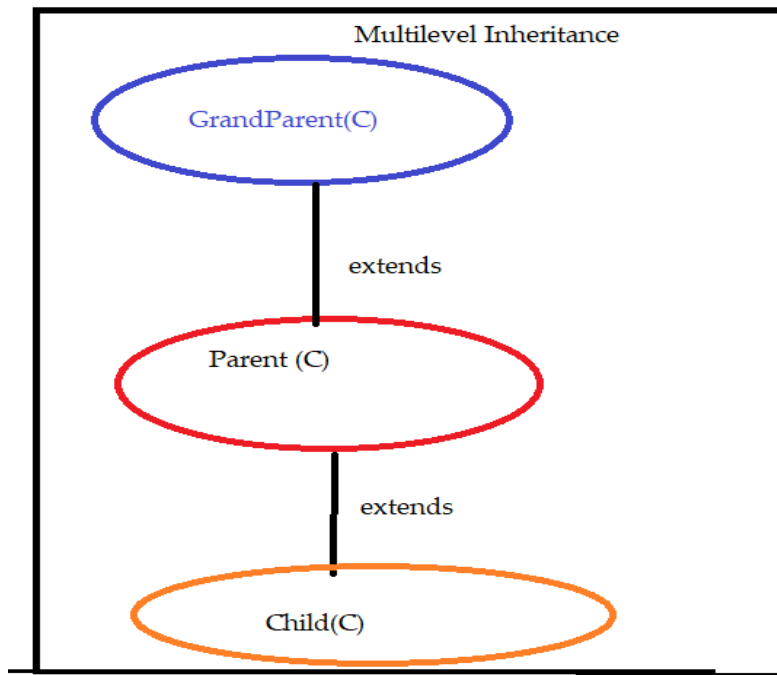
```
public class Child extends Parent {  
}
```

```
public class Parent extends Child {  
}
```

Above scenario is **invalid** as cyclic inheritance is not possible.

5. Multilevel inheritance:

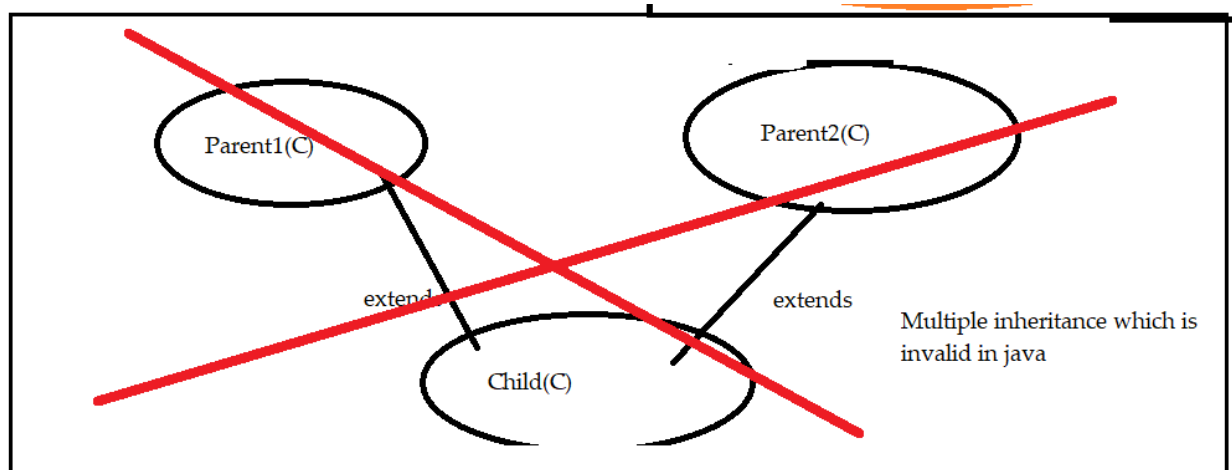
Whenever a child extends another class which is actually a child class of another class this scenarios depicts **multilevel inheritance** which is valid.



6. Multiple Inheritance is not possible :

Whenever a class is trying to extends multiple classes at a same time which is **not possible** is called **Multiple Inheritance**.

Example:



Access of variables in child class of parent's class using inheritance:

a. Variables are not having same name:

If we want to access the variable from parent class then we can call it by child object's reference variable.

Example:

```
public class Parent {  
    int i = 50;  
    static String s= "abc";  
}  
public class Child extends Parent  
{  
  
    public void bike()  
    {  
  
        System.out.println(this.i); // Parent's class i variable  
        System.out.println(super.i); // parent's class i variable  
  
        System.out.println(super.s); // parent's class s static  
variable  
    }  
  
    public static void main(String[] args) {  
  
        Child c = new Child();  
  
        System.out.println(c.i); // parent class non static variable  
  
        System.out.println(s); // parent's class static variable.  
    }  
}
```

Note: Super keyword:

super is a keyword which is used to access global variable from parent's class or super class. It can be used only in non static area as we have **this** keyword.

b. Variables are having same name:

```
public class Parent{  
    int i = 50;  
    static String s= "abc";  
  
public class Child extends Parent  
    {  
        int i = 20;  
  
public void bike()  
    {  
System.out.println(super.i);// parent's class i variable  
System.out.println(this.i);// child class variable  
  
public static void main(String[] args) {  
  
        Child c = new Child();  
  
System.out.println(c.i);// child class non static variable= 20  
  
        Parent p = new Parent();  
        System.out.println(p.i);// Parent's class non static variable
```

Philosophy inside inheritance:

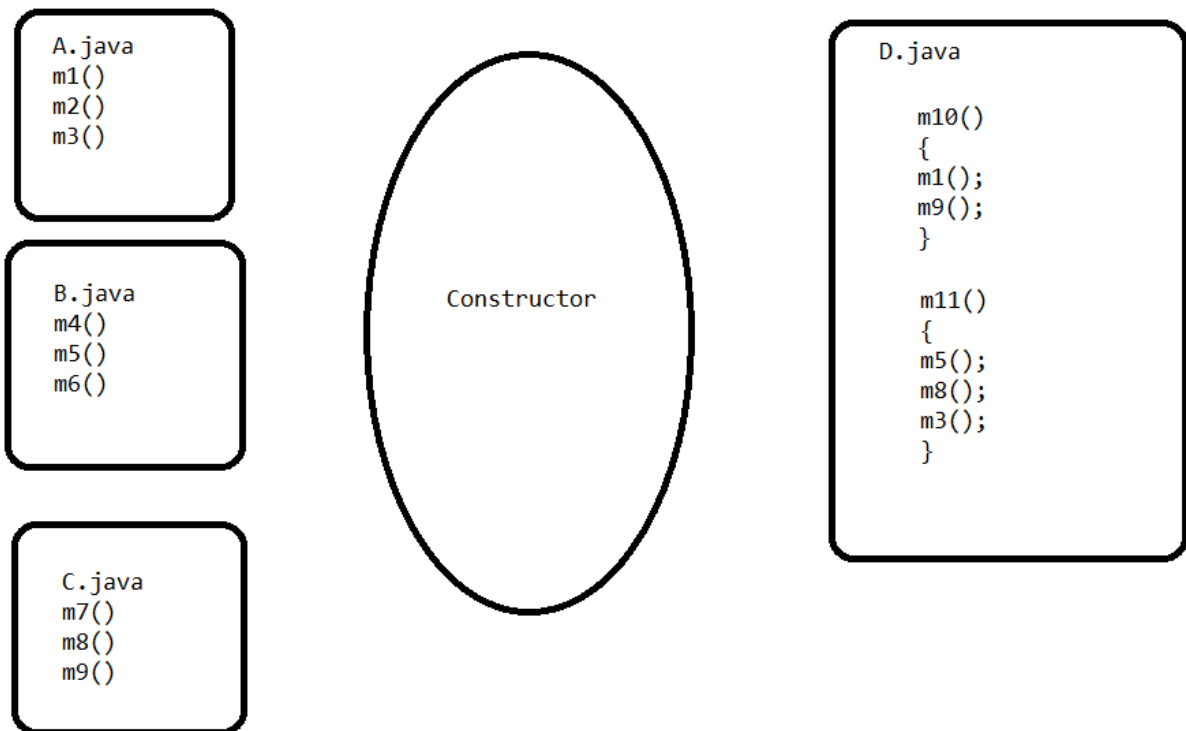
a. IS A Relationship: This represents there is a parent child relationship between the two classes. It means we have to use extends keyword between the two classes.

Example:

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

```
    Here A and B has IS A Relationship  
}
```

b. Has A Relationship: This represents if there is any set of functionality that is to be used in another class then we have to use Has A Relationship. This means we should create an object of the class whose functionality we are going to use and call the method by reference variable.



Access modifiers

Access modifiers: Through which we define the accessibility of any entity (Class, variable, method) whether they are accessible to a particular location or not.

Class level access modifier:

a. public

b. <default>

c. final

d. abstract:

a. public: If we declare a class as public then that class will be accessible throughout the project(with in the package and outside the package).

```
public class AccessTest {
```

b. <default>: If we declare a class as <default> then that class will only be accessible within the package but not outside the package.

```
class DefaultTest {  
}
```

c. final: It is a modifier which actually used with public / <default>.

If we declare a class as final :

a. public final: If we declare a class as public final then we can access the class throughout the project but we cannot create a child of that class i.e we cannot extends it.

Example:

```
public final class FinalClass {
```

or

```
final public class FinalClass {
```

b. <default> final: If we declare a class as <default> final then we can access the class within the package only but we cannot create a child of that class i.e we cannot extends it.

```
final class FinalClass {
```

Note: We cannot create the child class of final class but we can make a final class as parent of another class.

d. abstract modifier:

Whenever we don't have complete information of the implementation of some methods then we declare the class as abstract and the incomplete methods in it as also abstract methods.

- It is a modifier which is applicable to classes as well as methods.
- We can declare a class as abstract with no method as abstract in it but if a class has abstract method then compulsorily that class should also be declared as abstract.

```
public abstract class AbstractTest {  
  
    public abstract void m4();  
  
    public abstract void m5();  
  
    public abstract class AbstractTest2 {
```

- We cannot create an object of abstract class because it contains complete as well as incomplete methods and those incomplete methods cannot be called as they don't have body.

```
public abstract class AbstractTest {  
  
    public static void main(String[] args) {  
  
        AbstractTest at = new AbstractTest();// Error-  
        cannot create object  
  
    }  
}
```

- The method which are incomplete in abstract class has to be complete in its child class.

- If a child class is not able to provide the implementation of incomplete methods then we should declare that class also as abstract and then provide the implementation in its child class.

- If we write public abstract or abstract public then both meaning are same.

Example:

```
public abstract class AbstractTest2 {  
or  
abstract public class AbstractTest2 {
```

- We cannot declare a static method as abstract as we cannot override them.

Example:

```
public abstract class AbstractTest {
```

```
// public abstract static void m8();- We cannot declare  
a static method as abstract
```

We cannot declare a variable as abstract because a variable cannot have body.

- We can have a constructor inside an abstract class but we can call that constructor from child class object.
- We **cannot** have final and abstract together for a method or class.

Method level access modifier:

a. public

b. <default>

c. private

d. protected

e. final: to be done after overriding concept

a. public : If a method or variable is declare as public then it can be accessible throughout the project provided the class which contains it must have visibility or accessibility in that location.

```
public int var1 = 50;
public void m1()
{
    System.out.println("m1 method from Default Test class");
}
```

b. <default>: If a method or variable is declare as <default> then it can be accessible within the package only.

```
String s = "abc";// default variable
```

```
void m2()
{
    System.out.println("m2 method from default Test class");
}
```

c. private: If a method or variable is declared as private then it can be accessible only within the class it cannot be accessible outside the class. We cannot access the private method or variable even if we extends that class and try to access the private from that class.

```
private char c = '%';

private static void m3()
{
    System.out.println("m3 method from default class");
}
```

c. protected:

protected = <default> + through child class reference variable for outside the package.

If we declare a method or variable as protected then we will be able to access it within the package and through the child class reference variable for outside the package.

```

package accessmodifiers;
public class DefaultTest {

    protected double d = 99.99;
    protected void m4()
    {
        System.out.println("Protected method");
    }
}

package variablesinjava;

import accessmodifiers.DefaultTest;

public class B extends DefaultTest {

    public static void main(String[] args) {

        DefaultTest dt = new DefaultTest();

        // dt.m4();// here it is not accessible
        System.out.println(dt.d);// this is not accessible because it
        // can be done through child class
        B b = new B();
        b.m4();// here it is accessible
        System.out.println(b.d);// protected variable

    }

package variablesinjava;
public class C extends B{

    public static void main(String[] args) {

        B b = new B();
        // b.m4();// here it is not accessible

        C c = new C();
        c.m4();// here it is accessible}

}

```

d. Final method: We can declare a method as final which cannot be overridden by its child class.

```

public class FinalParent {

    public void home()
    {
        System.out.println("Home");
    }

    public void property()
    {
        System.out.println("property");}
}

```

```

public final void marry()
{
    System.out.println("marry method from Parent class");
}
}
public class ChildOfFinalParent extends FinalParent {

    public void marry()// error- because final method cannot be
    overridden.
    {
        System.out.println("child marry method");
    }

    public static void main(String[] args) {
        ChildOfFinalParent c = new ChildOfFinalParent();
        c.home();
        c.property();
        c.marry();
    }
}

```

Summary of all kind of access modifiers

	Scenario	private	default	protected	public
1	With in the same class	Y	Y	Y	Y
2	From the child class of same package	N	Y	Y	Y
3	From non child class of same package	N	Y	Y	Y
4	From the child class of outside package	N	N	Y(from child reference only)	Y
5	From non child class of outside package	N	N	N	Y

private<default<protected<public

Polymorphism:

This is an OOPs concept in which we have one name but many forms.

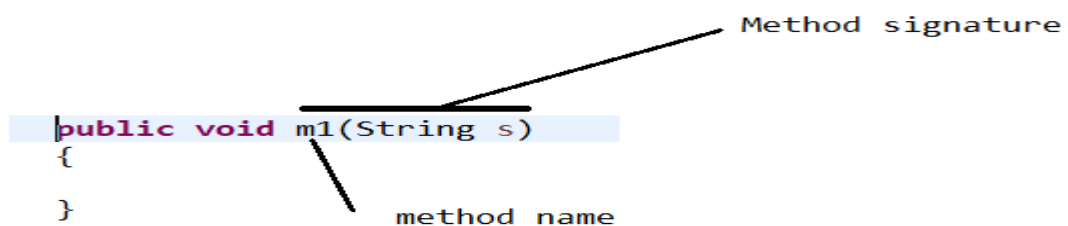
We have two types of polymorphism:

a. Overloading.

b. Overriding.

Through polymorphism we have achieved protability.

a. Method Overloading: Methods can be called as overloaded if the name of those methods are same but the ~~signature(name + arguments)~~ signature (name + arguments) are different.



The diagram shows a code snippet: `public void m1(String s)`. A horizontal line is drawn above the text `m1(String s)`, with an arrow pointing from the label "Method signature" to this line. Another arrow points from the label "method name" to the text `m1`.

Example:

```
public void m1()  
{  
    System.out.println("m1 method with no argument");  
}  
  
public void m1(int i)  
{  
    System.out.println("m1 method with one argument");  
    System.out.println(10);  
}  
  
public void m1(String s, int i)  
{  
    System.out.println("method with two arguments");  
}
```

- Method name must be same but their arguments must be different atleast their order.
- Method type doesn't matter in the case of overloading but name should be same and argument should be different.
- Return type of method doesn't make any difference to call a method as overloaded.
- Access modifier doesn't make any difference to call a method as overloaded.
- In Overloading method gets execute based on reference variable that's why overloading is also known as Compile time polymorphism.
- Overloading is also known as Early binding.

b. Overriding: A method can be called as overridden if:

1. There should be a relationship between the two classes i.e inheritance.
2. Name and signature of method in parent class and name and signature in child class must be same.
3. Return type of both the methods must be same.

```
classA{
    public int m1()
    {
        System.out.println("m 1 method from B class");
        return 50;
    }
}
Class B{
    public int m1()
    {
        System.out.println("m1 method from A class");
        return 10;
    }
}
```

4. Access modifier of overridden method should be increase in Child class but not get decrease.

```
Class A{
    void m2()
    {
        System.out.println("m2 method from A class");}
}
Class B{
    public void m2()
    {
        System.out.println("m2 method from B class");}
}
```

5. In overriding method gets execute based on the run time object that's why it is known as Runtime polymorphism.

6. Overriding is also known as Late binding.

Rule in java:

A parent reference variable can be used to hold child class object. And with that object we can call all the overridden methods based on object and Overloaded method method based on reference variable.

7. We cannot have a method as static in one class and the identical method definition in another class but the type is non static which is invalid in java.

8. Constructor can be overloaded but cannot be overridden.

Example:

```
Class A{
    public void m5() {
        System.out.println("m5 method from B class");
    }
}
Class B{
    public static void m5(){
        System.out.println("m4 static method from A class");}
}
```

Note: Child class reference variable cannot be used to hold parent's object.
i.e

B bb = new A();---- This is invalid in java

Static method with respect to overriding:

- We cannot override a static method because static method doesn't follow overriding but it follows Method hiding in which always reference variable decides the execution of method if both methods in both the class are having same return type, static, name and signature are also same.
- Main method cannot be overridden it can be overloaded.

Example:

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

Overriding with respect to variables:

Variable does not follow overriding principle.

Example:

```
package polymorphism;  
  
class A {  
    int i =10;  
  
    public void home()  
    {  
        System.out.println("home method of A class");  
    }  
  
    public void property()  
    {  
        System.out.println("Property method of A class");  
    }  
}
```

```

public void car(){
    System.out.println("Car from A class");
}

public void marry()
{
    System.out.println("Marry method from A class");
}

public int m1()
{
    System.out.println("m1 method from A class");
    return 10;
}

public static void m4()
{
    System.out.println("m4 static method from A class");
}

// public static void m5()
// {
//     System.out.println("m4 static method from A class");
// }

Protected void m2()
{
    System.out.println("m2 method from A class");
}
}

package polymorphism;

public class B extends A {

int i =50;

    public void bike()
    {
        System.out.println("Bike from B class");
    }

    public void marry()
    {
        System.out.println("marry method from B class");
    }

    public static void m4()
    {
        System.out.println("M4 static method from B class");
    }
}

```



```

    public int m1()
    {
        System.out.println("m 1 method from B class");

        return 50;
    }

    public void m2()
    {
        System.out.println("m2 method from B class");
    }

//    public void m5()
//    {
//        System.out.println("m5 method from B class");
//    }

    public static void main(String[] args) {
        B b = new B();
        b.home();// A class home method
        b.car(); // A class car method
        b.bike();// B class bike method
        b.marry();// B class marry method
        b.m4();// B class static method

        System.out.println(b.i);//50

        A a = new A();
        a.marry();// A class marry method
        a.m4();// A class Static method

        System.out.println(a.i);//10
        A aa = new B();

        aa.m2();// child class

        aa.marry();// child class marry method

        aa.car();//Parent class

        aa.m1(); // Child class m1 method

        aa.m4();// A class Static method
        System.out.println(aa.i);//10
//        B bb = new A();- invalid in java

    }
}

```

Output from B class
 home method of A class
 Car from A class
 Bike from B class
 marry method from B class
 M4 static method from B class
 50
 Marry method from A class
 m4 static method from A class
 10
 m2 method from B class
 marry method from B class
 Car from A class
 m 1 method from B class
 m4 static method from A class
 10

Sr. no	Overloading	Overriding
1	Name of method must be same	Name must be same.
2	Argument must be different.	Argument must be same.
3	Method signature must be different.	Method signature must be same.
4	There is no restrictions for return type.	Return type must be same.
5	There is no restrictions for Access modifiers.	Scope of method must be same or can be improve but not reduce.
6	Method get execute(resolution) on the basis of reference variable.	Method get execute (Resolution) is based on runtime object.
7	It is also known as Early binding or compile time polymorphism or static polymorphism.	It is also known as late binding or Runtime polymorphism or dynamic polymorphism.
8	Static methods can be overloaded.	Static method doesn't follow overriding but it follows method hiding.
9	Inheritance is not mandatory.	Inheritance is mandatory.
10	Constructor can be overloaded.	Constructor cannot be overridden.

Through polymorphism we have achieved protability.

Encapsulation

Encapsulation: A process of binding the data and corresponding methods into a single unit is called Encapsulation.

Encapsulation can be achieved by: **Data hiding and Abstraction**

Through encapsulation we have achieved Security.

Encapsulation = Data hiding+ Abstraction.

a. Data hiding: Declaring a variable as private and use that variable inside a public method so that we cannot access that variable directly outside the class.

Example:

```
public class ServerOfBank {  
  
    private double cust1balance = 50000;  
  
    public void getBalance(int pin){  
  
        if(pin ==1234){  
  
            System.out.println("Your account balance is "+cust1balance);  
        }  
        else {  
            System.out.println("Wrong pin please check and try again");  
        }  
    }  
}
```

Note: here we have hide the cust1balance from other class.

b. Abstraction: Hiding the internal functionality or implementation and provide only the output or services that we are offering is called Abstraction.

Example:

```
public static void main(String[] args) {  
  
    ServerOfBank sb = new ServerOfBank();  
  
    sb.getBalance(1234);  
  
}
```

Note: here we are only showing the name of service (method name) but not the implementation of method which is Abstraction.

Complete data hiding and Abstraction example:

```
package encapsulation;
```

```
public class ServerOfBank {
```

```
    private double cust1balance = 50000;
```

```
    public double getBalance(int pin) {
```

```
        if (pin == 1234) {
            System.out.println("Your account balance is " +
cust1balance);
```

```
        } else {
            System.out.println("Wrong pin please check and try
again");
        }
    }
```

```
    return cust1balance;
```

```
}
```

```
    public void withdrawlAmount(int amount) {
```

```
        setBalance(amount);
```

```
        System.out.println("Amount withdrawl is :"+amount);
```

```
}
```

```
    public void setBalance(int amount) {
```

```
        if (cust1balance > amount) {
            cust1balance = cust1balance - amount;
```

```
        }
```

```
}
```

```
}
```

```
package encapsulation;
```

```
public class ATMMachine {
```

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

```
        ServerOfBank sb = new ServerOfBank();
```

```
        sb.getBalance(1234);
```

```
        sb.withdrawlAmount(5000);
```

```
        sb.getBalance(1234);
```

```
    }}
```

Output:

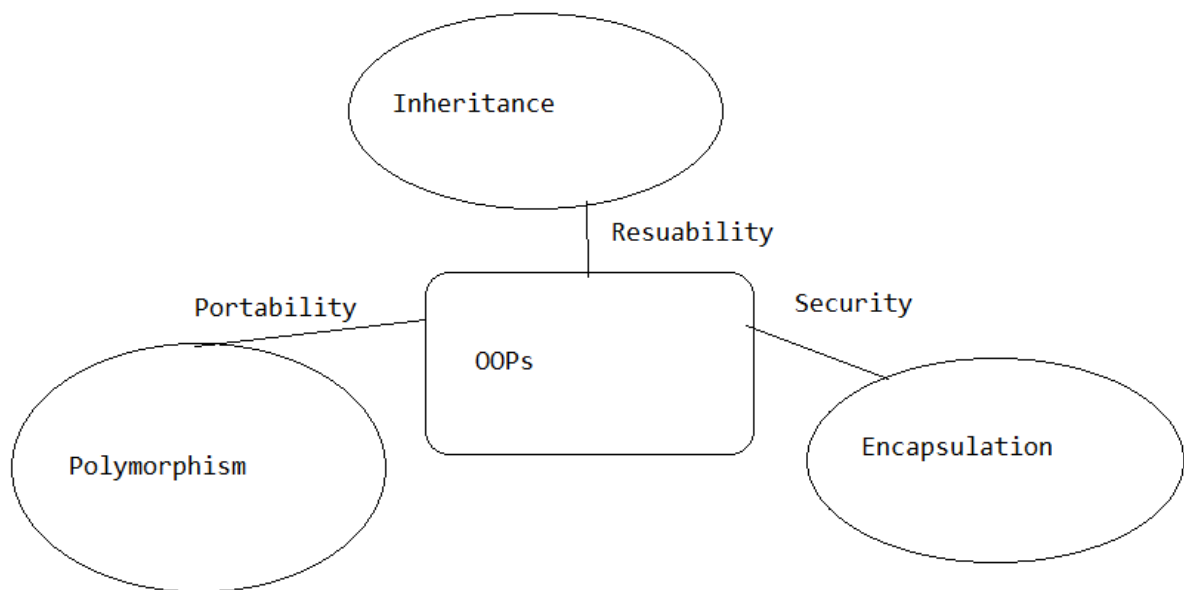
Your account balance is 50000.0

Amount withdrawl is :5000

Your account balance is 45000.0

The above yellow highlighted part refers to Abstraction and the one which got highlighted in blue color is referring Data hiding.

Through encapsulation we have achieved Security.



Pillars of OOPs concept

Interface in Java:

Interface is an entity which has all incomplete methods inside it i.e it has all the rules which is to be implemented by the class which implements it. To provide the implementation of the methods inside an interface a class has to use **implements** keyword.

- Inside an interface methods are by default public and abstract whether we declare them or not.

Example:

```
public interface InterfaceOne {  
  
    public abstract void m1(); // public void m1(); // void m1();  
  
    void m2(); // public void m2();}
```

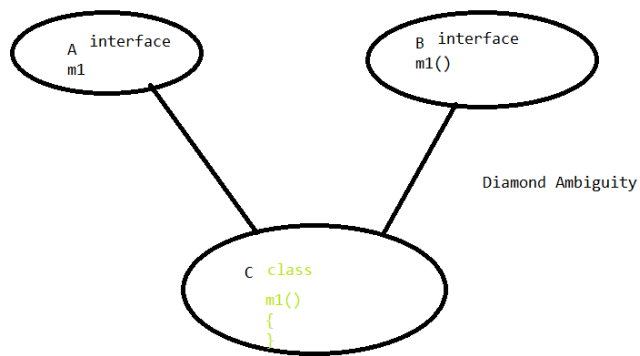
- The class which implements an interface has to provide the implementation of all the methods of interface but if that class is not able to provide the implementation then we have to declare that class as abstract and its child class responsibility to provide the implementation.
- An interface can have static method inside it but it should be complete in nature.

```
public interface InterfaceOne {  
  
    public static void m6(){  
        System.out.println("Static method in interface");  
    }  
  
    public static void main(String[] args) {  
  
        m6();  
        InterfaceOne.m6();  
    }  
}
```

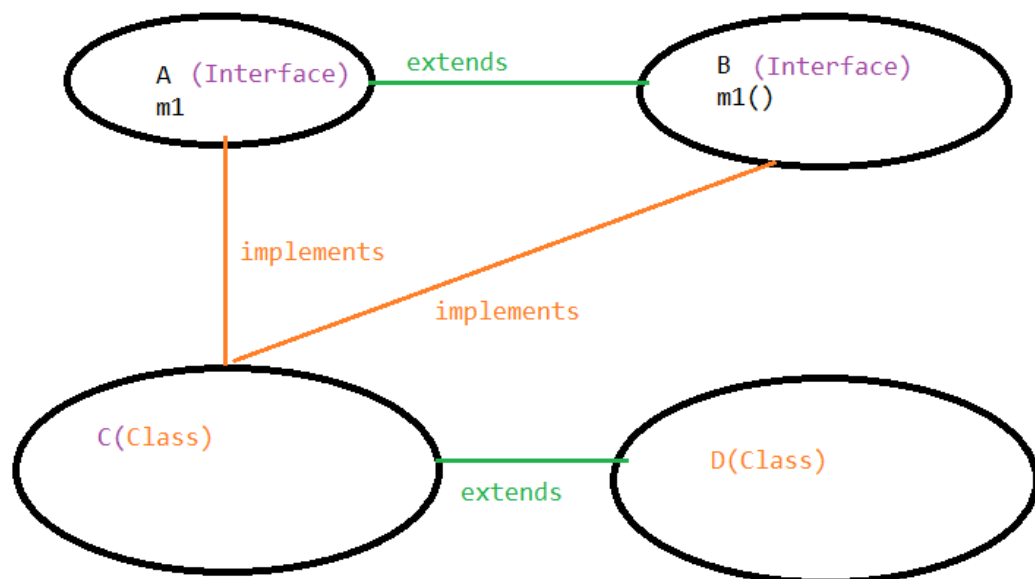
Output:

```
Static method in interface  
Static method in interface
```

- With the help of interface we can achieve multiple inheritance which is not possible in classes.
- Diamond ambiguity problem can be resolved by using an interface but it is there in terms of classes.



- We can implements any number of interfaces at a time but that class has to provide the implementations of all the methods available in all the interfaces.
- We can extends a class and implements the interfaces at the same time.



Example:

```
public class Test3 extends Test4 implements InterfaceOne, InterfaceTwo
```

- We can extend any number of interfaces with another interface example:

```
public interface InterfaceFour extends InterfaceThree, InterfaceOne
```

- All the variables defined inside an interface are by default public static and final whether we defined them or not

Example:

```
static int i = 50; // public static int i = 50 or int i = 50 -- All are same
```

Sr. no	Interface	Abstract class
1	If we don't know anything about implementation then we should go for interface.	If we know partial implementation then we should go for Abstract class.
2	Inside interface all the methods are public and abstract whether we declare them or not. Hence it is called as 100% abstract.	2. Every method present inside the abstract class need not be public and abstract.
3	Every variable inside the interface is by default public static and final.	3. Every variable inside an abstract class has no restrictions.
4	Constructor concept is not applicable to interface.	4. Constructor concept is applicable to abstract class.

Exception Handling:

Any unexpected event which cause program or code to get terminate abnormally is called Exception and the process through which we can handle that abnormal termination to normal termination is called exception handling.

It is the mechanism to handle the runtime error so that we can avoid abnormal termination .

In java we can handle the exception in 2 ways:

a. try-catch-finally combination or try-catch.

b. throws keyword.

a. try-catch-finally combination:

1. try-catch:

- We have to write the risky code inside the try block and the possible type of exception which can arrive inside the catch block.

```
public static void main(String[] args) {  
    int i=0;  
    System.out.println("first line");  
    try {  
        i = 10/0;    }  
    catch (ArithmeticException ){  
  
        System.out.println("Arithmetic exception arrived");  
        i =20;  
    }  
}
```

- We can have multiple catch blocks for one try block.

Example:

```
public static void main(String[] args) {  
  
    int i=0;  
  
    System.out.println("first line");  
  
    try {  
  
        i = 10/0;  
    }  
  
    catch(NullPointerException e)  
    {  
        System.out.println("Nullpointer Exception ocured");  
  
    }  
  
    catch (ArrayIndexOutOfBoundsException e)  
    {  
        System.out.println("Arithmetic exception arrived");  
        i =20;  
    }  
}
```

- If the corresponding catch block matches with the type of exception comes under try block then that particular catch block will get execute but if no catch block get match with the type of exception then program get terminate abnormally.

Example:

```
public static void main(String[] args) {  
  
    int i=0;  
  
    System.out.println("first line");  
  
    try {  
  
        i = 10/0;  
    }  
  
    catch(NullPointerException e)  
    {  
        System.out.println("Nullpointer Exception ocured");  
    }  
}
```

```

catch (ArrayIndexOutOfBoundsException e)
{
    System.out.println("Arithmetic exception arrived");
    i =20;
}

System.out.println("Final value of i is :"+i);

int j= i+2;

System.out.println(j);
System.out.println("last line");
}

```

Output:

first line

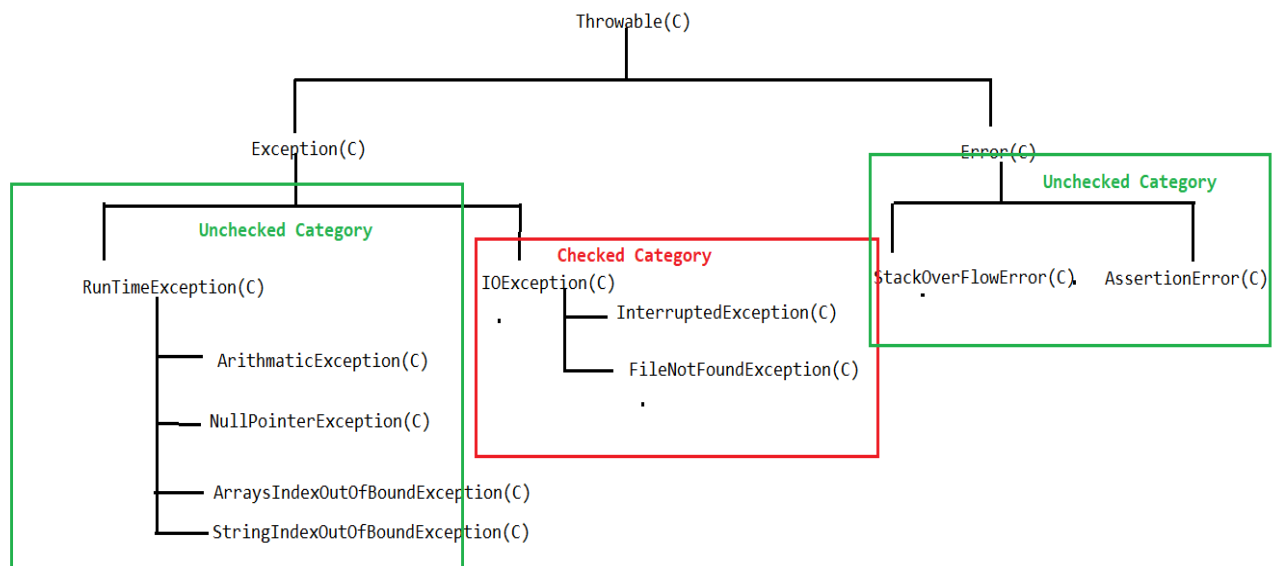
```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at exceptionhandling.Test.main(Test.java:14)

```

Null pointer exception example:

If we have null value in any



variable, performing any operation on the variable throws a `NullPointerException`.

```

package exceptionhandling;

public class NullPointerTest {

    Test2 t;
}

```

```

public void m2(){
    try {
        t.m1();
        System.out.println("after exception");
    }

    catch(NullPointerException e)
    {
        System.out.println("Null pointer exception comes out
        please check");
    }
}

public static void main(String[] args) {

    NullPointerException npt = new NullPointerException();
    npt.m2();
}
}
package exceptionhandling;

public class Test2 {

    public void m1()
    {
        System.out.println("m1 method is running");
    }
}

```

Output:

Null pointer exception comes out please check

Note:

1. We should always keep the risky code in try block and that will be handle with the help of its corresponding catch block.
2. We should always keep Exception catch block at last of all the catch block.

Example:

```
public static void main(String[] args) {  
  
    int i = 10;  
    int j = 0;  
    int k = 0;  
  
    try {  
        k = i/j;  
    }  
  
    catch(StringIndexOutOfBoundsException e)  
    {  
        System.out.println("Arithmetic exception arrived");  
        String d = e.getMessage();  
  
        System.out.println("the message is :"+d);  
        k=30;  
    }  
  
    catch(NullPointerException e)  
    {  
        System.out.println("Nullpointer exception arrived");  
    }  
  
    catch(ArrayIndexOutOfBoundsException a)  
    {  
        System.out.println("Array index Exception arrived");  
    }  
  
    catch(Exception e)  
    {  
        System.out.println("Exception arrived please check");  
    }  
}
```

2. try-catch-finally:

Finally is a block which always get executes whether we get an exception or not.

Generally we use finally block for keeping the code which is related to clean up activities.

Example:

```
package exceptionhandling;

public class Test4 {
    // try catch finally
    public static void main(String[] args) {
        try
        {
            String s = null;
            s.equals("Velocity");

        }

        catch(ArithmeticException e)
        {
            System.out.println("Arithmethic exception");
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException");
        }

        catch (NullPointerException e) {
            System.out.println("Null pointer exception");
        }

        finally {
            System.out.println("finally block got executed");
        }
    }
}
Null pointer exception
finally block got executed
```

example 2 in which exception arrived:

```
package exceptionhandling;

public class Test4 {

    // try catch finally
    public static void main(String[] args) {
        try
        {
            String s = null;
            s.equals("Velocity");

        }

    }
}
```

```

        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic exception");
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException");
        }

        finally {
            System.out.println("finally block got executed");
        }
    }
}

```

Output:

```

finally block got executed
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.equals(Object)" because "s" is null
    at exceptionhandling.Test4.main(Test4.java:10)

```

finalize(): It is a method which is called by garbage collector to destroy unused object which allocates the memory.

Combination of try, catch and finally block:

- We can have finally or catch as a immediate block after try block.

```

try {
    int i = 10/0;
}

finally {
    System.out.println("Finally block");
}

```

In the above example if exception arrived there is no way to handle it hence code will terminate abnormally.

- We can have try-catch / try-catch-finally / try-finally at any block example:

```
try {  
    int i = 10/0;  
  
    try {  
  
    }  
  
    finally {  
  
    }  
}  
  
finally {  
    System.out.println("Finally block");  
}
```

Example:

```
public static void main(String[] args) {  
  
    try {  
        int i = 10/0;  
  
        try {  
  
        }  
  
        finally {  
  
        }  
    }  
  
    catch(ArithmeticException e)  
    {  
        try {  
  
        }  
        catch(Exception r)  
        {  
  
        }  
    }  
  
    finally {  
  
    }
```



```

        try {
            }
        catch(Exception e)
        {

        }
        System.out.println("Finally block");
    }
}

}

```

Throws keyword: Throws keyword is used to handle the exception which are of checked category just to resolve the error that we have at compile time. But for unchecked category there is no such use of it.

Whenever any exception comes in the program then we will not be able to protect it by abnormal termination.

Example:

```

public static void main(String[] args) throws InterruptedException,
ArithmeticException{
    System.out.println("first line");

    Thread.sleep(5000);

    System.out.println("second line after sleep time");

    int i = 10/0;

}

public static void m1() throws FileNotFoundException{
    String path = "E:\\desktop\\Katraj\\Oct Batch\\Git.docx";

    FileInputStream fis = new FileInputStream(path);

    System.out.println("file path is correct");

}

```

Note :

if we want the program to get terminate normally then we should handle the exception through try –catch.

Throw keyword :

It is a keyword which is used to throw the exception in a program deliberately. Throw is used with the method or block

Example:

```
public static void main(String[] args) {
    int i = 5;

    System.out.println("first line");

    if (i>5)
    {
        throw new NullPointerException();
    }
    else {
        throw new ArithmeticException();
    }
}
```

Throw, throws and throwable:

Throw is a keyword which is used to throw the exception, throws is a keyword which is used to handle the exception but it cannot protect the program from abnormal termination and Throwable is a class in java which is a parent class of Error and Exception.

Error :

Due to some system's inefficiency or deficiency cause the program to terminate in an abnormal manner then that is known as Error.

Example: StackOverflowError and AssertionError (TestNG)

```
public class ErrorDiscussion {

    public static void m1()
    {
        System.out.println("m1 method");

        m2();
    }
    public static void m2()
    {
        System.out.println("m2 method");
        m1();}
}
```

```

public static void main(String[] args) {

    m1();

}
}

```

Output:

```

m1 method
m2 method
m1 method
m2 method

```

Exception in thread "main" java.lang.StackOverflowError

Note: To overcome error we have to improve the performance of overall system.

Difference between Final, Finally and Finalize.

Final: final is access modifier which is applicable for class, methods and variables.

- 1) If a class declare as a final then we can not extend that class.i.e we cannot make child of that class.
- 2) if a method declare as a final then we cannot override that method in child class.
- 3) If a variable declare as final then it will become constant. And we can't perform re-assignment for that variable.

Finally: finally keyword is used in exception handling .so finally block always associate with try catch to maintain cleanup code.

Finalize(): finalize() is a method which is always worked as garbage collector just before destroying an object to perform cleanup activities.

So this method destroy unused object which allocates the memory .

Array in java

Collection of same type or Homogenous type of data and represent them as a single entity is called an Array.

Index position of an array always starts with 0.

Length or total number of values can be calculated for an array through `a.length`;

If any value is not been stored inside an array then it takes its default value in it.

Example for the first way to define the array:

```
public static void main(String[] args) {  
    //          first way to define the array  
  
    int a [] = new int [8];  
  
    a[0] = 10;  
    a[1] = 20;  
    a[2] = 80;  
    a[3] = 5;  
    a[4] = 50;  
  
    System.out.println(a[2]);  
  
    int size = a.length;  
  
    System.out.println("Size of array is "+size);  
  
    //          for(int i=0; i<5; i++) {  
    //              System.out.println(a[i]);  
    //          }  
  
    for(int i=0; i<size; i++)  
    {  
        System.out.println(a[i]);  
    }  
}
```

Output:

```
80  
Size of array is 8  
10  
20
```

```
80
5
50
0
0
0
```

```
String ss [] = new String [2];

for(int j=0; j<ss.length; j++)
{
    System.out.println(ss[j]);
}
```

Output:

Null

Null

For each loop:

```
// for each loop i.e advance for loop
int a [] = new int [8];

a[0] = 10;
a[1] = 20;
a[2] = 80;
a[3] = 5;
a[4] = 50;

for(int g:a)
{
    System.out.println(g);
}
```

Output:

```
10
20
80
5
50
0
0
0
```

variable which we wants to iterate

```
for(int g:a)
```

variable takes the values one by one from a.

```
{  
    System.out.println(g);  
}
```

Example 2:

```
for(int g:a)  
{  
    if(g>0)  
    {  
        System.out.println(g);  
    }  
}
```

Output:

```
10  
20  
80  
5  
50
```

```
public class Test2 {  
    public static void main(String[] args) {  
  
        int a [] = new int [5];  
  
        a[0] = 1;  
        a[1] = 20;  
        a[2] = 80;  
        a[3] = 5;  
        a[4] = 50;  
  
        // To print the even index position's element value  
        for (int i=0; i<a.length; i++)  
        {  
            if(i%2==0)  
            {  
                System.out.println("The value at "+i+ " position is  
"+a[i]);  
            }  
        }  
    }  
}
```

```

// to print the even element in the output
for(int i=0; i<a.length; i++)
{
    if(a[i]%2==0)
    {
        System.out.println(a[i]);
    }
}

}

```

Output:

```

The value at 0 position is 1
The value at 2 position is 80
The value at 4 position is 50
20
80
50

```

Another way to define an array:

```

public static void main(String[] args) {

```

```

    String s[] = {"abc", "def", "ghi"}; //alternate way

```

```

    int i[] = {10, 45, 1, 6, 8};

```

```

    int g =0;
    for(String h:s)
    {
        if(g%2 ==0)
        {
            System.out.println(h);
        }

        g++;
    }

```

String class in java:

String is a class which is present under java.lang package. Which is used to store sequence of character in it. It is immutable in nature, i.e if we perform any manipulation operation on the existing string then it will not affect to the original string till we don't assign the same variable to it.

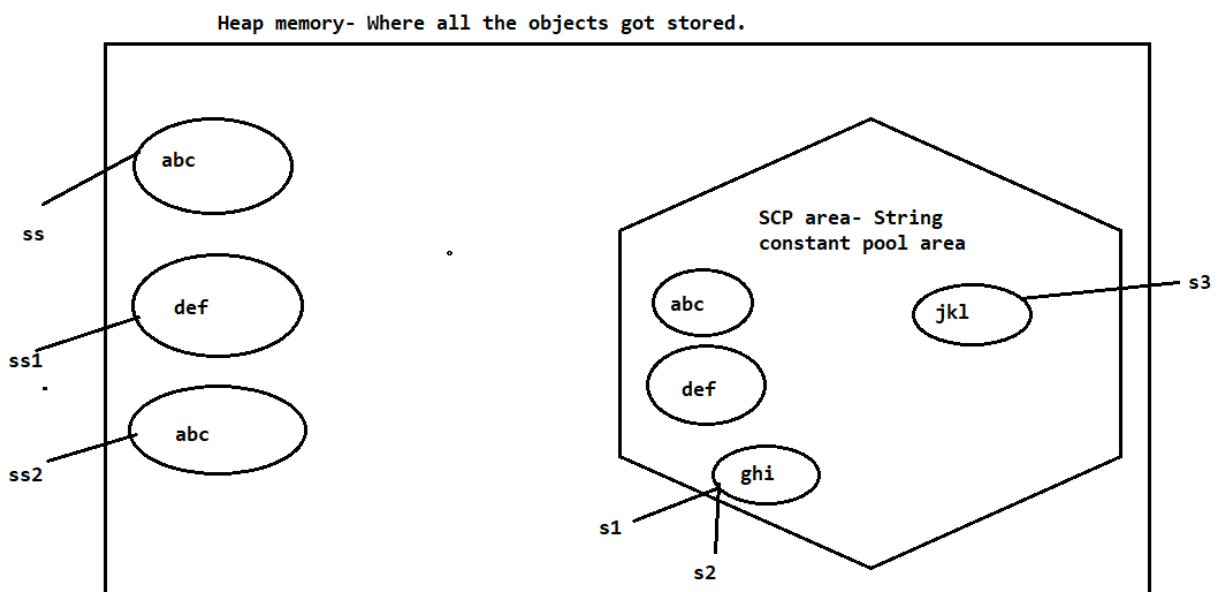
We can represent string in 2 ways:

a. `String s = new String("abc");`

If we represent string with new keyword then two objects will get created first one in Heap memory and second is in SCP area (depends on the content of string is already present there or not).

b. `String ss = "abc"`

If we represent string without new keyword then object of string will be created inside only SCP area which depends on if any object is already available in it. If it is already available then it will not create any new object and point that variable to the same. But if object is not present then a new object will get created.



`==` Vs equals method:

`==` it is used to compare the object reference variable whether they are pointing to the same object or not. i.e it is used for reference comparison.

Equals is used to compare the content of String whether it is equal or not. i.e it is used for content comparison.

1) `==`: Example:

```
String s1= "ghi";  
String ss3= new String("ghi");  
String s2= "ghi";  
  
boolean ispointingsameobject = s1==s2;  
System.out.println(ispointingsameobject);// true  
boolean ispointingsameobject1 = s1==ss3;  
System.out.println(ispointingsameobject1);//false
```

2) `equals`: `boolean` `isequal` = `s1.equals(ss3)`;
System.out.println(`isequal`);`//true`

Note: String is an immutable class and String buffer is a mutable class.

```
String s4 = "Velocity";  
  
System.out.println(s4+" corporate training  
institute");//Velocity corporate training institute  
  
System.out.println(s4.concat(" corporate training  
institute"));//Velocity corporate training institute  
  
System.out.println(s4);// Velocity-This represents immutable  
  
StringBuffer sb= new StringBuffer("Velocity");  
  
sb.append(" corporate training institute");  
  
System.out.println(sb);//Velocity corporate training institute  
- This represents mutable.
```

Methods in String class

1.equals(String s):

This returns true only if all the content inside a string is exactly identical to another one otherwise false .

```
String s1 = "abc";  
String s2 = "aBc";  
  
boolean isequal = s1.equals(s2);  
  
System.out.println(isequal); //false
```

2.equalsIgnoreCase(String s):

This method returns true if the content of both the strings are equals without considering their case.

```
boolean iscontentequal = s1.equalsIgnoreCase(s2);  
  
System.out.println(iscontentequal); // true
```

3. Length():

This method returns the count of number of character present inside a string.

Example:

```
String s3 = "abcdefghi";  
  
int size = s3.length();  
  
System.out.println(size); //9
```

4. substring(int beginIndex):

This method returns a string which start from the provided index value in it as an argument.

```
String s4 = "Velocity";  
  
String s5 = s4.substring(4);  
  
System.out.println(s5); //city
```

5. substring(int beginindex , int endindex):

This method returns a string which starts with the provided begin index and ends with the end index -1 value.

```
String s4 = "Velocity";  
String s6= s4.substring(0, 4);  
System.out.println(s6);//Velo
```

6. charAt(int index):

This method returns a char value which is present at that index value.

```
String s7 = "Happy";  
char charvalue = s7.charAt(2);  
System.out.println(charvalue);//p
```

7. replace(char old, char new):

This method is used to replace the old char with the new one which is provided in the argument.

```
String s8 = "ababab";  
String s9 = s8.replace('b', 'a');  
System.out.println(s9);// aaaaaa
```

8. replace(String charseq, Stringcharseqnew):

This method is used to replace the string with the new string provided in the argument.

```
String s10 = "Constructor";  
String s11 = s10.replace("tructor", "tant");  
System.out.println(s11);//constant
```

9. Contains(String value):

This method returns true if the value provided as an argument has that content available in the String.

```
String s14 = "Velocity";  
  
boolean s15 = s14.contains("elo");  
  
System.out.println(s15); // true
```

10.toLowerCase():

This method is used to convert the given string in lowercase or small letter.

```
String s16 = "ABCDEF";  
  
String s17 = s16.toLowerCase();  
  
System.out.println(s17); // abcdef
```

11.toUpperCase():

This method is used to convert the given string in uppercase or capital letter.

```
String s18 = "ABcdef";  
  
String s19 = s18.toUpperCase();  
  
System.out.println(s19); // ABCDEF
```

12. trim():

This method removes the starting and trailing spaces from the string.

```
String s20= "   Velocity   ";  
  
System.out.println(s20);  
  
String s21 = s20.trim();  
  
System.out.println(s21); // Velocity
```

13. indexOf(char ch):

This method returns the index position of a character in a given string.

```
String s22 = "Velocity";  
  
int indexposition = s22.indexOf('o');  
  
System.out.println(indexposition);//3
```

Note: Through this method we can have the character index value which occurred at the first position and if there is no character found inside the string then it returns -1.

13. indexOf(char ch)

```
String s22 = "ancabc";  
  
int indexposition = s22.indexOf('z');  
  
System.out.println(indexposition);//-1
```

14. valueOf(any datatype):

This method converts any data type into String.

```
int a = 10;  
  
String s23 = String.valueOf(a);  
  
System.out.println(s23+8);//108
```

15.startsWith():

This method returns true if the given string starts with the string provided in the argument.

```
String s24 = "This is String";  
  
boolean s25 = s24.startsWith("This");  
  
System.out.println(s25);//true
```

16. endsWith():

This method returns true if the given string ends with String provided in the argument.

```
boolean s26 = s24.endsWith("ing");
```

```
System.out.println(s26); // true
```

17. split():

This method returns a String array on the basis of the string which is provided as an argument.

```
String s27 = "this is String";
```

```
String[] s28 = s27.split("i");
```

```
for(String w:s28)
{
    System.out.println(w);
}
```

Output:

```
th
s
s Str
ng
```

18.toCharArray():

This method returns a char array on the basis of provided string.

```
String s29 = "Velocity";
```

```
char[] s30 = s29.toCharArray();
```

```
for(char k:s30) {
    System.out.println(k);
}
```

```
V
e
l
o
c
i
t
y
```

19. isDigit(char c):

This method returns true if the provided character is a number.

```
char c = '1';
```

```
boolean isnumber = Character.isDigit(c);  
System.out.println(isnumber);// true
```

20. isAlphabetic(char c):

This method returns true if the provided character is a alphabet.

```
boolean isalphabet = Character.isAlphabetic(c);
```

```
System.out.println(isalphabet);// false
```

To convert from any data type to String we use valueOf method but if we want to convert String into other primitive data type then we have to use its Wrapper class for example Integer is a wrapper class of int data type.

Syntax to convert the String to a primitive data type :

```
WrapperClass.parseXxx();
```

Example:

```
String s32 = "100";
```

```
int s33 = Integer.parseInt(s32);  
System.out.println(s33);
```

```
String s34 = "false";
```

```
boolean s35 = Boolean.parseBoolean(s34);
```

```
System.out.println(s35);// false
```

```
String s36 = "1.56";
```

```
double s37 = Double.parseDouble(s36);  
System.out.println(s37+1);//2.56
```

Note : For the above cases if we provide an invalid value for that particular data type then we get exception `NumberFormatException` .

Regular Expression :

// Regex - Regular expression

```
String s38 = "@Ve$45546l*ocity";
```

```
String s39 = s38.replaceAll("[a-z]", "0");//  
System.out.println(s39);//@V0$455460*00000
```

```
String s40 = s38.replaceAll("[A-Za-z]", "");
```

```
System.out.println(s40);//@$45546*
```

```
String s41 = s38.replaceAll("[^a-z]", "%");
```

```
System.out.println(s41);//%%e%%%%%%l%ocity
```

```
String s42 = s38.replaceAll("[a-zA-Z0-9]", "0");
```

```
System.out.println(s42);//@00$000000*00000
```

```
String s43 = "@ac#$c%en%^45ture";
```

```
String s44 = s43.replaceAll("[^a-zA-Z]", "");
```

```
System.out.println(s44);//accenture
```


Casting:

To convert a type of a variable to another form is called as casting.

There are two types of casting based on primitive and nonprimitive.

a. primitive based casting:

1. implicit casting: To convert a lower order data type to higher order data type is called implicit casting. This is also known as **widening of data**.

```
byte b = 10;

int c = (int)b;

System.out.println(c);//10

int d = 100;

long e = (long)d;

System.out.println(e);//100
```

2. Explicit casting: To convert the higher order data type to lower order data type is called explicit casting. It is also known as **narrowing of data**.

In explicit casting there can be loss of data that may happen while casting.

```
int i = 10;

byte j = (byte)i;

System.out.println(j);//10

int k = 129;

byte l = (byte)k;

System.out.println(l);//-127

float m = 56.89f;

int n = (int)m;

System.out.println(n);//56
```

```
//          byte --> short ---> int ---> long ---> float ---> double

//          char

//          boolean

//          int + byte = int
//          byte + byte = int
//          int +long= long

//          int +int = int
```

Non Primitive based Casting:

There are two types of non primitive based casting:

a. Upcasting: Whenever we cast a sub class or child class to parent class or super class then it is known as UP casting.

```
Parent p = new Parent();

p.m1();// parent class

Child c = new Child();

c.m1();// child class

Parent d = (Parent)c;

d.m1();//child class
```

in the above example Parent is a super class and Child is a subclass.

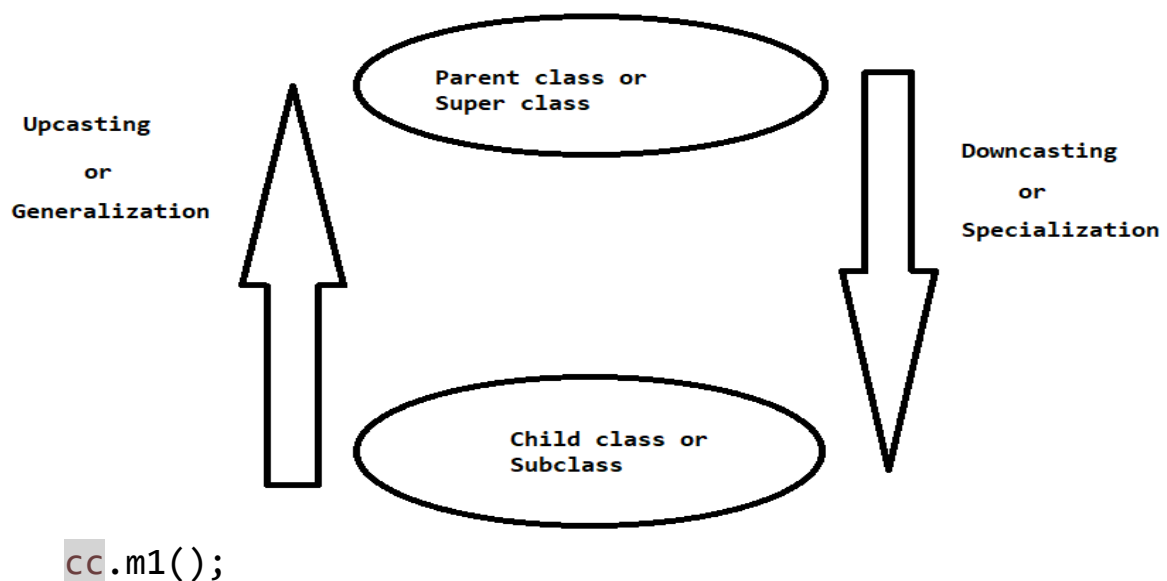
b. Downcasting: Whenever we try to cast a higher order or super class to sub class then it is known as Downcasting.

```
Parent e = new Child();  
  
Child f = (Child)e; // Down casting  
  
f.m2(); // parent class
```

in the above example Parent is a super class and Child is a subclass.

```
Parent pp = new Parent();
```

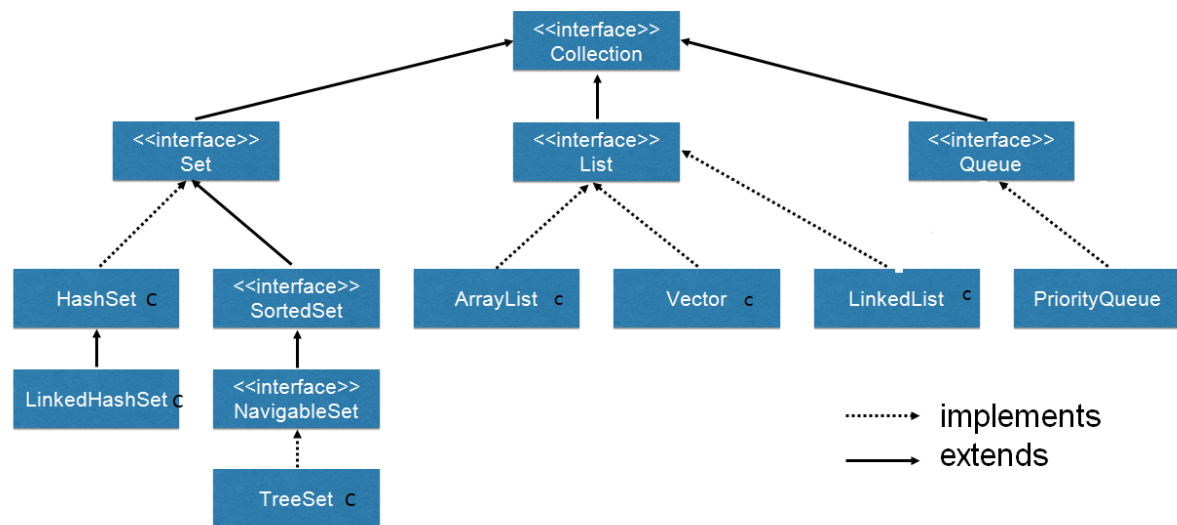
Child cc = (Child)pp; // invalid in java as only parent reference is allowed to hold child object but not vice versa.
here we get an exception - ClassCastException



Example:

When we come from Exception to any specific exception (Arithmetic or NullPointerException etc) then it is Specialization where as when we come from specific Exception to general exception (Exception) then it is Generalization.

Collection Interface or Framework



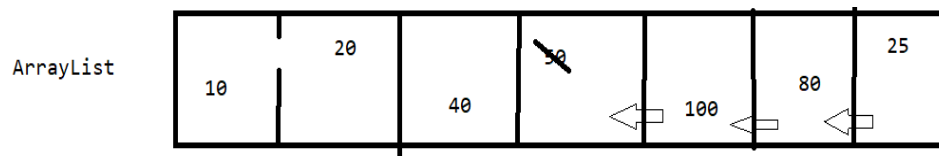
Collection Framework:

Collection: A group of object (same data type or different data type) represented as a single entity is called collection.

- Array contains homogenous elements but in collection it can have heterogeneous elements as well.
- Array doesn't contain any predefined method but in collection we have the support of predefined method.
- Size of the array is fixed whereas collection size is dynamic in nature it can adjust himself by using formula $\text{initial size} \times \frac{3}{2} + 1$ where initial size of the collection is 10.

ArrayList@:

1. Insertion order is preserved and duplicates are allowed.
2. For insertion and deletion operation we don't prefer ArrayList because there is shifting of data and due to which performance of the execution gets hamper.
3. for searching operation ArrayList is preferred.



Example 1:

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

```
    ArrayList al = new ArrayList();
```

```
    al.add("abc");
```

```
    al.add(false);
```

```
    al.add(50);
```

```
    System.out.println(al); // abc false 50
```

```
}
```

Output:

```
[abc, false, 50]
```

Example 2:

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

```
    ArrayList al = new ArrayList();
```

```
    al.add("abc");
```

```
    al.add(false);
```

```
    al.add(50);
```

```
    System.out.println(al); // abc false 50
```

```
    ArrayList<String> al1 = new ArrayList<String>();
```

```
    al1.add("abc");
```

```
    al1.add("def");
```

```
    al1.add("ghi");
```

```
    al1.add("jkl");
```

to check whether the value is present into the collection or not.

```
boolean iscontain = al1.contains("def");  
System.out.println(iscontain);
```

to remove the value from the collection

```
al1.remove("def");  
  
System.out.println(al1);//[abc, ghi, jkl]
```

to update the value inside the collection

```
al1.set(1, "mno");//  
  
System.out.println(al1);  
}
```

Example 3:

```
ArrayList<String> al2 = new ArrayList<String>();
```

```
al2.add("Pune");  
al2.add("Mumbai");  
al2.addAll(al1);  
al2.add("Bengaluru");  
al2.add("Delhi");
```

```
System.out.println(al2);//[Pune, Mumbai, xyz, mno, jkl, Bengaluru, Delhi]
```

```
boolean isallpresent = al2.containsAll(al1);  
System.out.println("isallpresent returns: "+isallpresent);
```

```
al2.remove("mno");
```

```
boolean isallpresentaferwards = al2.containsAll(al1);
```

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

```
}
```

```
}
```

Output:

```
[Pune, Mumbai, xyz, mno, jkl, Bengaluru, Delhi]  
isallpresent returns: true  
false
```

Conversion from Array to collection and vice versa:

```
package collectiondiscussion;
```

```
import java.util.ArrayList;
```

```
import java.util.Arrays;
```

```
public class ConversionFromArrayToCollection {
```

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

conversion from collection to array

```
        ArrayList<String> al = new ArrayList<String>();
```

```
        al.add("Pune");
```

```
        al.add("Mumbai");
```

```
        al.add("Bengaluru");
```

```
        al.add("Delhi");
```

```
        int size = al.size();
```

```
        String [] s = new String[size];
```

```
        al.toArray(s);
```

```
        for(String ss:s)
```

```
        {
```

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

```
        }
```

conversion from array to collection

```
        String[] sss = {"abc", "def", "ghi"};
```

```
        ArrayList<String> al2 = new ArrayList<String>(Arrays.asList(sss));
```

```
        System.out.println(al2);
```

```
    }
```

```
}
```

Output:

Pune

Mumbai

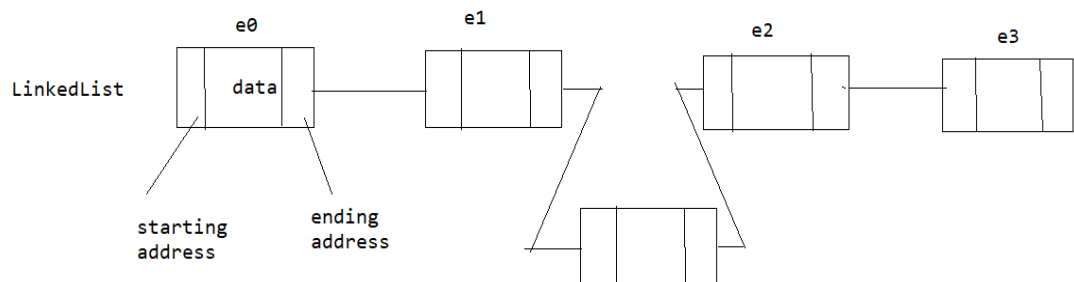
Bengaluru

Delhi

[abc, def, ghi]

LinkedList@:

- It is a class in which every structure contains address as well as data hence searching operation is not preferred.
- In linkedlist insertion and deletion can be done without shifting of data hence it is preferred over ArrayList.
- Memory wise Linked list is not prefer as it contains data as well as address.



Vector class@: It is a class inside the list interface which is the only legacy class available as it was available since 1.0v.

Set interface:

It doesn't allow the duplicate values in it i.e duplicate values are not allowed inside set.

HashSet@: It is a class which removes the duplicate values but it couldn't maintain the order of insertion.

LinkedHashSet@: It is a class in which we maintain the order aslo we can remove the duplicate values as well.

Example:

```
public static void main(String[] args) {  
  
    ArrayList<String> al2 = new ArrayList<String>();  
    al2.add("Pune");  
    al2.add("Mumbai");  
    al2.add("Bengaluru");  
    al2.add("Delhi");  
    al2.add("Delhi");  
    al2.add("Mumbai");  
    System.out.println(al2);  
}
```



```
HashSet<String> hs = new HashSet<String>();
```

```
    al2.addAll(hs);
```

```
    hs.add("Pune");  
    hs.add("Mumbai");
```

```
    hs.add("Bengaluru");  
    hs.add("Delhi");  
    hs.add("Mumbai");  
    hs.add("Delhi");  
    System.out.println(hs);
```

```
LinkedHashSet<String> lhs = new LinkedHashSet<String>();
```

```
    lhs.add("Pune");  
    lhs.add("Mumbai");
```

```
    lhs.add("Bengaluru");  
    lhs.add("Delhi");  
    lhs.add("Mumbai");  
    lhs.add("Delhi");  
    System.out.println(lhs);
```

```
    }
```

```
}
```

Example:

```
[Pune, Mumbai, Bengaluru, Delhi, Delhi, Mumbai]  
[Delhi, Bengaluru, Pune, Mumbai]  
[Pune, Mumbai, Bengaluru, Delhi]
```

TreeSet: It is a class which by default maintain a sorting order (ascending) and also it doesn't consider duplicate objects.

Example:

```
TreeSet<String> ts = new TreeSet<String>(new MyComp());
```

```
    ts.add("Pune");  
    ts.add("Mumbai");
```

```
    ts.add("Bengaluru");  
    ts.add("Delhi");  
    ts.add("Mumbai");  
    ts.add("Delhi");
```

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

```
TreeSet<Integer> ts1 = new TreeSet<Integer>();
```

```
    ts1.add(50);  
    ts1.add(12);  
    ts1.add(4);  
    ts1.add(99);
```

```
    System.out.println(ts1);
```

```
package collectiondiscussion;
```

```
import java.util.Comparator;
```

```
public class MyComp implements Comparator<String>{
```

```
    @Override
```

```
    public int compare(String o1, String o2) {
```

```
        return -o1.compareTo(o2);
```

```
    }
```

```
}
```

Output:

[Pune, Mumbai, Delhi, Bengaluru]

[4, 12, 50, 99]

Map Interface

Map is an interface which has the provision with its implementation class to represent the data in the form of key-value pair.

Inside the map key cannot be duplicate as it follows the principle of set interface.

Value can be duplicated inside a map.

HashMap©: A class which implements Map interface in which the data represents in the form of key and value but the order of the insertion is not maintained.

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
```

```
    hm.put(101, "Delhi");  
    hm.put(506, "Pune");  
    hm.put(802, "Mumbai");  
    hm.put(900, "Bengaluru");
```

to get all the values of the hashmap

```
        System.out.println(hm);  
    }  
}
```

Output:

{900=Mumbai, 101=Mumbai, 506=Pune}

Example 2:

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();  
hm.put(101, "Delhi");  
hm.put(506, "Pune");  
hm.put(101, "Mumbai");  
hm.put(900, "Bengaluru");
```

to get all the values of the hashmap

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

Output:

{900=Bengaluru, 101=Mumbai, 506=Pune}

Example 2:

```
HashMap<Integer, String> hm = new HashMap<Integer, String>();
```

```
hm.put(101, "Delhi");  
hm.put(506, "Pune");  
hm.put(802, "Mumbai");  
hm.put(900, "Bengaluru");
```

to get all the values of the hashmap

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

to get all the values :

```
Collection<String> allvalues = hm.values();  
System.out.println(allvalues);
```

to get the value of corresponding key

```
String keyvalue = hm.get(506);  
System.out.println(keyvalue);
```

to check whether a hashmap contains a particular value or not

```
boolean isvaluepresent = hm.containsValue("Bengaluru");
```

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

to check whether a hashmap contains a particular key or not

```
boolean iskeypresent = hm.containsKey(506);
```

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

to get all the keys available:

```
Set<Integer> keyvalues = hm.keySet();  
  
System.out.println(keyvalues);  
}  
}
```

Output:

```
{802=Mumbai, 900=Bengaluru, 101=Delhi, 506=Pune}  
[Mumbai, Bengaluru, Delhi, Pune]  
Pune  
true  
true  
[802, 900, 101, 506]
```

LinkedHashMap: This is a class inside the map interface in which the order of insertion is maintained.

```
LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();  
lhm.put(101, "Delhi");  
lhm.put(506, "Pune");  
lhm.put(802, "Mumbai");  
lhm.put(900, "Bengaluru");  
System.out.println(lhm);
```

Output:

```
{101=Delhi, 506=Pune, 802=Mumbai, 900=Bengaluru}
```