Java OOPs Concepts

OOPs (Object-Oriented Programming System)

- **Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects.
- It simplifies software development and maintenance by providing some concepts:

  - Object

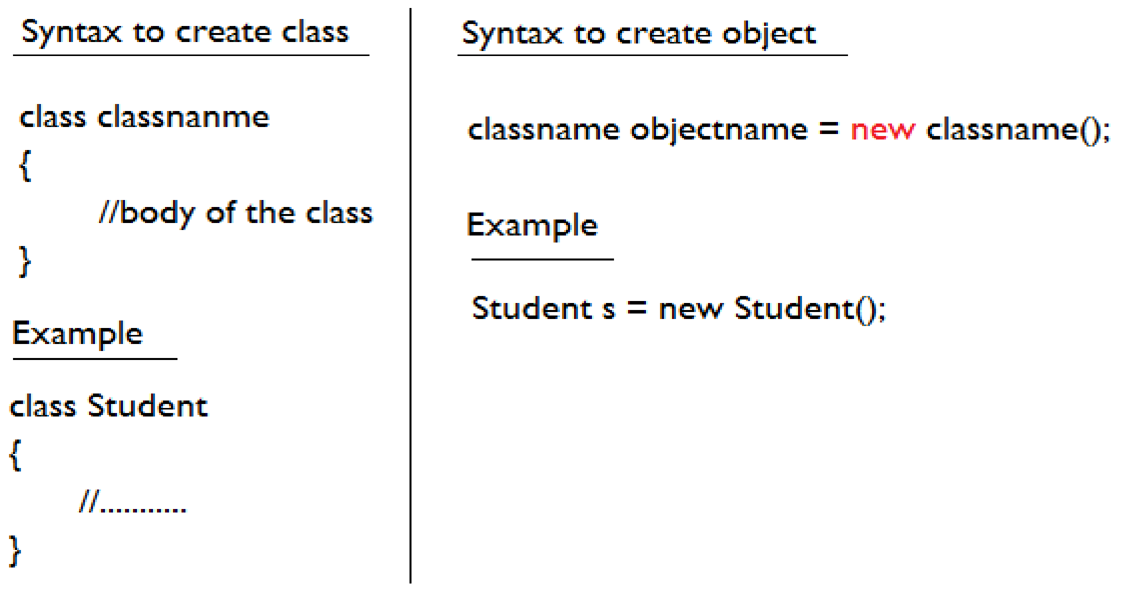
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

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

Class

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



Syntax to create class

```
class classnanme
{
        //body of the class
}
```

Example

```
class Student
{
      //...........
}
```

Syntax to create object

```
classname objectname = new classname();
```

Example

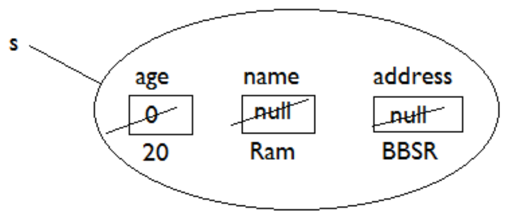
```
Student s = new Student();
```

Example#1

```
class Student{
        int age;
        String name;
        String address;
}
class Test{
        public static void main(String args[]){
                Student s = new Student();
                System.out.println("Age = "+s.age);
                System.out.println("Name = "+s.name);
                System.out.println("Address = "+s.address);
        }
}
```

Example-2

```
class Student{
        int age;
        String name;
        String address;
}
class Test{
        public static void main(String args[]){
                Student s = new Student();
                s.age = 20;
                s.name = "Ram";
                s.address = "BBSR";

                System.out.println("Age = "+s.age);
                System.out.println("Name = "+s.name);
                System.out.println("Address = "+s.address);
        }
}
```



Inheritance

- *When one object acquires all the properties and behaviors of a parent object*, it is known as inheritance.
- It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

- If *one task is performed in different ways*, it is known as polymorphism.
- For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.
- Another example can be to speak something; for example, a cat speaks meow, dog barks woof, etc.

Abstraction

- *Hiding internal details and showing functionality* is known as abstraction.
- For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

- *Binding (or wrapping) code and data together into a single unit are known as encapsulation*.
- For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation.
- Java bean is the fully encapsulated class because all the data members are private here.

Java Naming Convention

- Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.
- But, it is not forced to follow.
- So, it is known as convention not rule.
- These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.
- All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention.
- If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of Naming Conventions in Java

- By using standard Java naming conventions, you make your code easier to read for yourself and other programmers.
- Readability of Java program is very important.
- It indicates that less time is spent to figure out what the code does.

Naming Conventions of the Different Identifiers

The following table shows the popular conventions used for the different identifiers.

Identifiers Type	**Naming Rules**	**Examples**

Class	It should start with the uppercase letter.
	It should be a noun such as Color, Button, System, Thread, etc.
	Use appropriate words, instead of acronyms.
		public class **Employee**
		{
		//code snippet
		}

Interface	It should start with the uppercase letter.
	It should be an adjective such as Runnable, Remote, ActionListener.
	Use appropriate words, instead of acronyms.
		interface **Printable**
		{
		//code snippet
		}

Method	It should start with lowercase letter.
	It should be a verb such as main(), print(), println().
	If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().
		class Employee
		{
		// method
		void **draw()**
		{
		//code snippet
		}
		}

Variable	It should start with a lowercase letter such as id, name.
	It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
	If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
	Avoid using one-character variables such as x, y, z.
		class Employee
		{
		// variable
		int **id**;
		//code snippet
		}

Package	It should be a lowercase letter such as java, lang.
	If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.
		//package
		package **com.javatpoint;**
		class Employee
		{
		//code snippet
		}

Constant	It should be in uppercase letters such as RED, YELLOW.
	If the name contains multiple words, it should be separated by an underscore(_) such as MAX_PRIORITY.
	It may contain digits but not as the first letter.
		class Employee
		{
		//constant
		static final int **MIN_AGE** = 18;
		//code snippet
		}

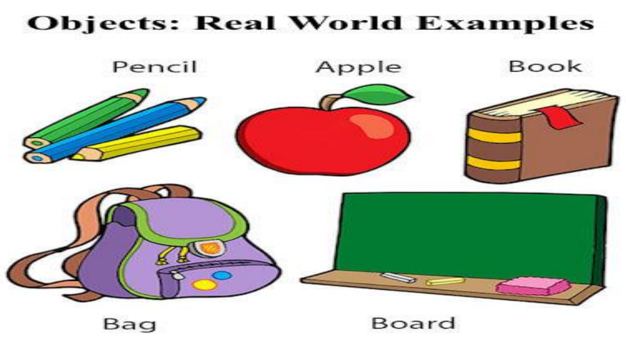
CamelCase in Java naming conventions

- Java follows camel-case syntax for naming the class, interface, method, and variable.
- If the name is combined with two words, the second word will start with uppercase letter always such as actionPerformed(), firstName, ActionEvent, ActionListener, etc.

Objects and Classes in Java

- An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

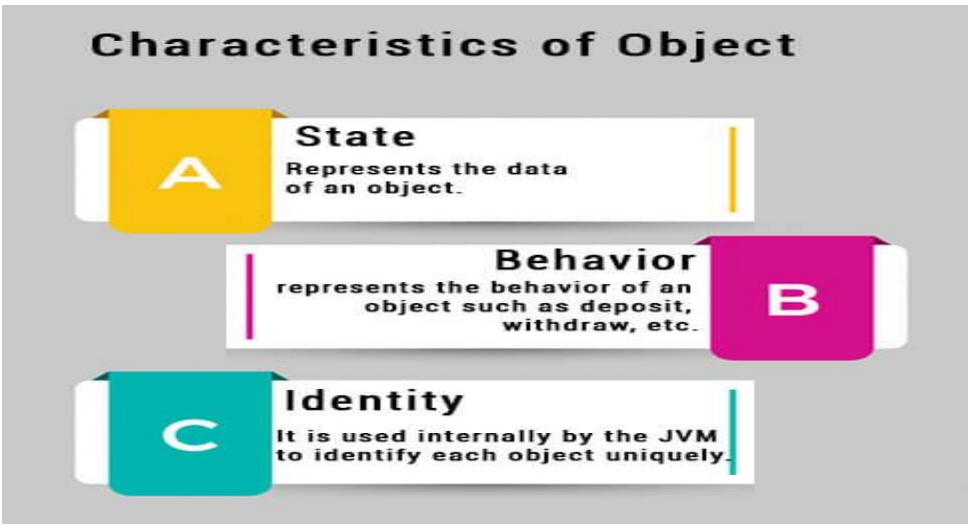
What is an object in Java



An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

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


Characteristics of Object
A
State
Represents the data of an object.
Behavior
represents the behavior of an object such as deposit, withdraw, etc.
B
C
Identity
It is used internally by the JVM to identify each object uniquely.

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

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

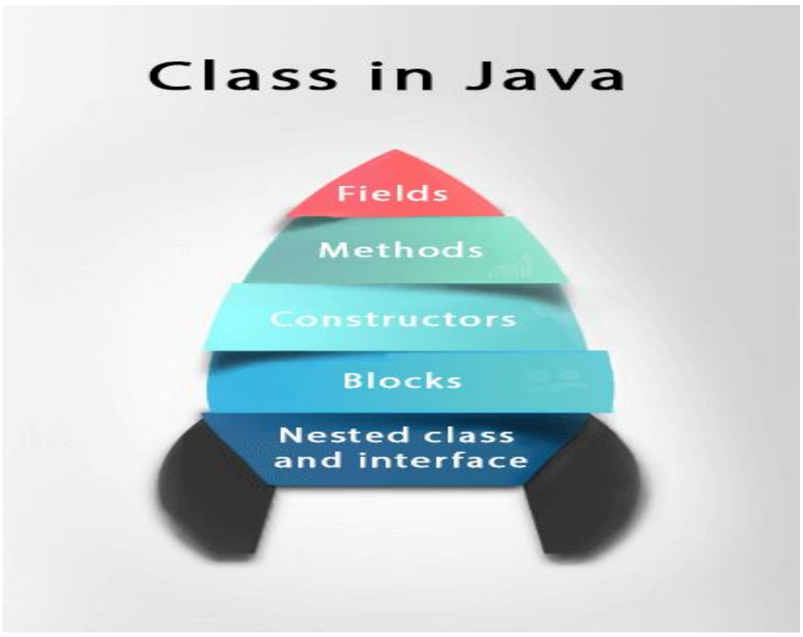
- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

What is a class in Java

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

A class in Java can contain:

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



Syntax to declare a class:

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

Instance variable in Java

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

Method in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

---

new keyword in Java

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

Object and Class Example: main within the class

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

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

File: Student.java

```java
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
```

```
//Creating an object or instance
Student s1=new Student();//creating an object of Student
//Printing values of the object
System.out.println(s1.id);//accessing member through reference variable
System.out.println(s1.name);
 }
}
```

Test it Now

Output:

```
0
null
```

Object and Class Example: main outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main() method in another class.

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

File: TestStudent1.java

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

Test it Now

Output:

```
0
null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

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

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

File: TestStudent2.java

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

Test it Now

Output:

```
101 Sonoo
```

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

File: TestStudent3.java

```java
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Test it Now

Output:

```
101 Sonoo
102 Amit
```

2) Object and Class Example: Initialization through method

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

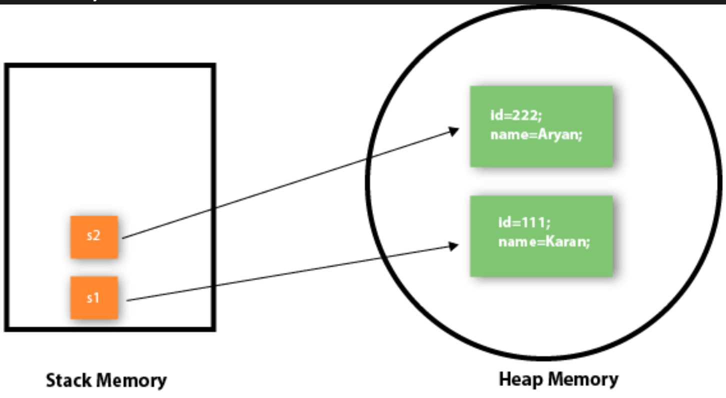
File: TestStudent4.java

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

Test it Now

Output:

```
111 Karan
222 Aryan
```



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

3) Object and Class Example: Initialization through a constructor

We will learn about constructors in Java later.

Object and Class Example: Employee

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

File: TestEmployee.java

```java
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

Test it Now

Output:

```
101 ajeet 45000.0
102 irfan 25000.0
103 nakul 55000.0
```

Object and Class Example: Rectangle

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

File: TestRectangle1.java

```
class Rectangle{
 int length;
 int width;
 void insert(int l, int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle1{
 public static void main(String args[]){
  Rectangle r1=new Rectangle();
  Rectangle r2=new Rectangle();
  r1.insert(11,5);
  r2.insert(3,15);
  r1.calculateArea();
  r2.calculateArea();
}
}
```

Test it Now

Output:

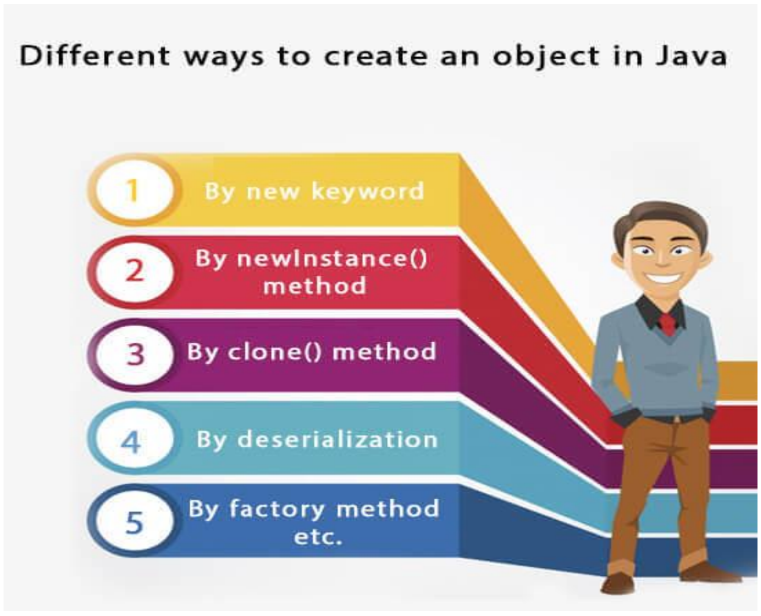
```
55
45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

We will learn these ways to create object later.



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```java
new Calculation();//anonymous object
```

Calling method through a reference:

```java
Calculation c=new Calculation();
c.fact(5);
```

Calling method through an anonymous object

```java
new Calculation().fact(5);
```

Let's see the full example of an anonymous object in Java.

```java
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Output:

```
Factorial is 120
```

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

Initialization of primitive variables:

```java
int a=10, b=20;
```

Initialization of refernce variables:

```java
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Let's see the example:

```java
//Java Program to illustrate the use of Rectangle class which
//has length and width data members
class Rectangle{
 int length;
 int width;
 void insert(int l,int w){
  length=l;
  width=w;
 }
 void calculateArea(){System.out.println(length*width);}
}
class TestRectangle2{
 public static void main(String args[]){
  Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
  r1.insert(11,5);
  r2.insert(3,15);
r1.calculateArea();
r2.calculateArea();
}
}
```

Test it Now

Output:

```
55
45
```

Real World Example: Account

File: TestAccount.java

```
//Java Program to demonstrate the working of a banking-system
//where we deposit and withdraw amount from our account.
//Creating an Account class which has deposit() and withdraw() methods
class Account{
int acc_no;
String name;
float amount;
//Method to initialize object
void insert(int a,String n,float amt){
acc_no=a;
name=n;
amount=amt;
}
//deposit method
void deposit(float amt){
amount=amount+amt;
System.out.println(amt+" deposited");
}
//withdraw method
void withdraw(float amt){
if(amount<amt){
System.out.println("Insufficient Balance");
}else{
amount=amount-amt;
System.out.println(amt+" withdrawn");
}
}
//method to check the balance of the account
void checkBalance(){System.out.println("Balance is: "+amount);}
//method to display the values of an object
void display(){System.out.println(acc_no+" "+name+" "+amount);}
}
//Creating a test class to deposit and withdraw amount
class TestAccount{
public static void main(String[] args){
Account a1=new Account();
a1.insert(832345,"Ankit",1000);
a1.display();
a1.checkBalance();
a1.deposit(40000);
a1.checkBalance();
a1.withdraw(15000);
a1.checkBalance();
}}
```

Test it Now

Output:

```
832345 Ankit 1000.0
Balance is: 1000.0
40000.0 deposited
Balance is: 41000.0
15000.0 withdrawn
Balance is: 26000.0
```

Java static keyword

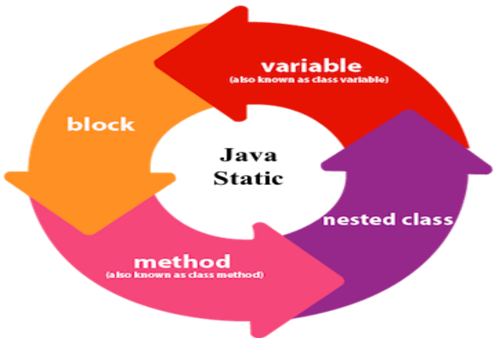
The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

The static can be:

car example

```
class Car{
 String name;
 String modelno;
 String color;
void start(){
 System.out.println("
car started");
 }
void run(){
 System.out.println("
car running");
 }
void stop(){
 System.out.println("
car stopped");
 }
void insert(String
name,String
modelno,String color){
 this.name=name;

this.modelno=modelno
;
 this.color=color;
}
void display(){
 System.out.println
(name+" "
+modelno+" "+color);
}

public class TestDrive{
 public static void main
(String[]args){
 Car c= new Car();
 c.insert("verna","
MH36AG0250","grey");
 c.display();
 c.start();
 c.run();
 c.stop();
 }
}
}
```

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class



1) Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="ITS";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Test it Now

Output:

```
111 Karan ITS
222 Aryan ITS
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.

- A static method can access static data member and can change the value of it.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
```

3) Java static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Example of static block

```
class A2{
  static{System.out.println("static block is invoked");}
  public static void main(String args[]){
   System.out.println("Hello main");
  }
}
```

Test it Now

```
Output:static block is invoked
       Hello main
```

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

```
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

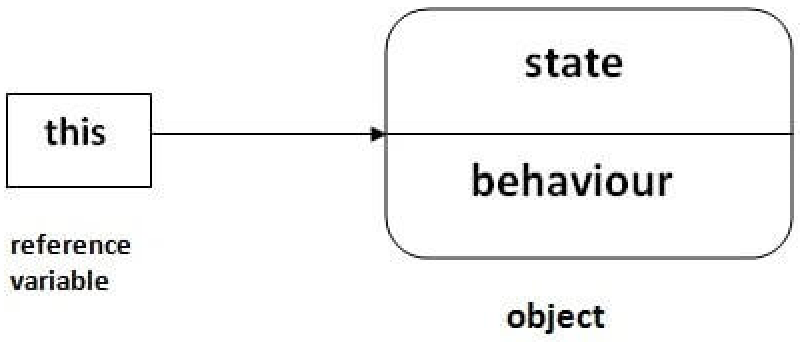
Test it Now

Output:

```
static block is invoked
```

this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

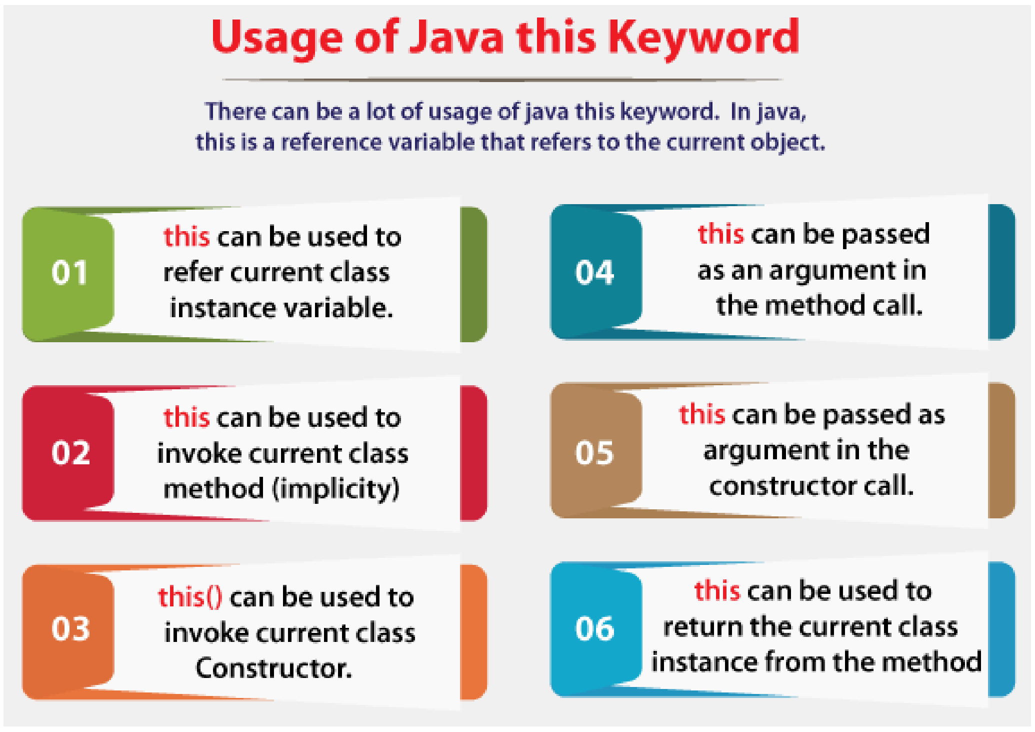


Usage of Java this keyword

Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

Suggestion: If you are beginner to java, lookup only three usages of this keyword.


Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01 this can be used to refer current class instance variable.

02 this can be used to invoke current class method (implicity)

03 this() can be used to invoke current class Constructor.

04 this can be passed as an argument in the method call.

05 this can be passed as argument in the constructor call.

06 this can be used to return the current class instance from the method

1) this: to refer current class instance variable

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

Test it Now

Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- It is an important part of OOPs (Object Oriented programming system).
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of the parent class.
- Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

Why use inheritance in java

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

- For Code Reusability.

Terms used in Inheritance

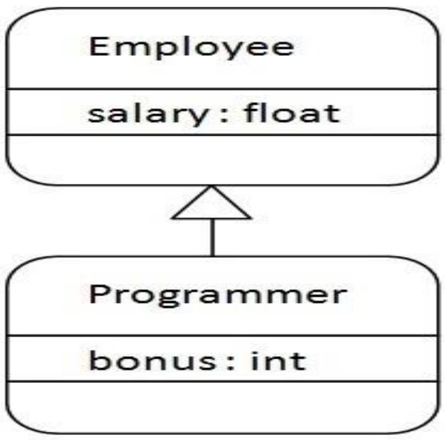
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent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

The syntax of Java Inheritance

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

- The **extends keyword** indicates that you are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Java Inheritance Example



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

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

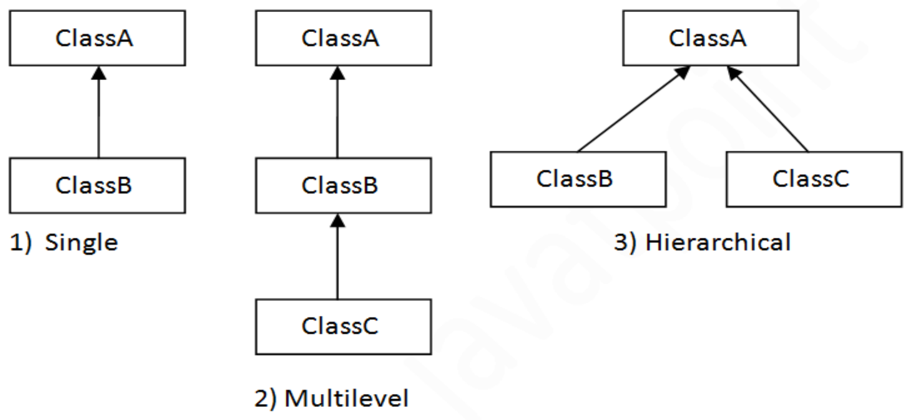
Test it Now

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

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

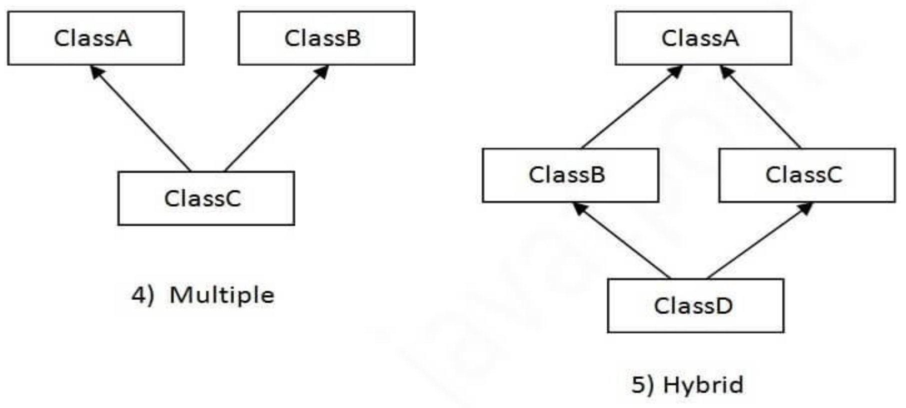
Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



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

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



.

Single Inheritance Example

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

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
```

```java
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

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

Multilevel Inheritance Example

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

File: TestInheritance2.java

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

Output:

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

Hierarchical Inheritance Example

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

File: TestInheritance3.java

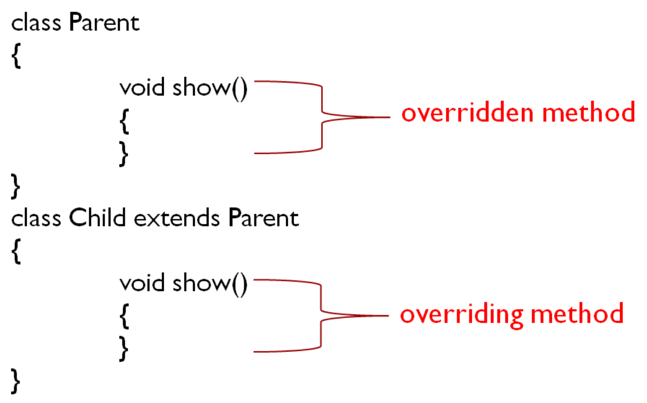
```java
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();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

Output:

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

Method Overriding in Java

- Redefining or Redesigning parent class method in child class is known as method overriding.
- In method overriding, parent class method signature and child class method signature must be same.



```
class Parent
{
            void show()
            {
            }
}
class Child extends Parent
{
            void show()
            {
            }
}
```

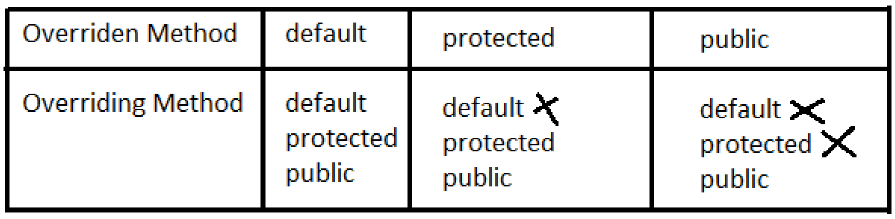
Example

```
class Parent{
        void show(){
                int a = 10;
                int b = 20;
                int c = a+b;
                System.out.println("sum1 = "+c);
        }
}
class Child extends Parent{
        void show(){
                int a = 10;
                int b = 20;
                System.out.println("sum2 = "+ (a+b));
        }
}
class MethodOverridingDemo{
        public static void main(String args[]){
                Child ob = new Child();
                ob.show();
        }
}
```

Rule#1

While overriding the methods it is possible to maintain same level permission or increasing order but not decreasing order.

private <   default < protected < public



Overriden Method	default	protected	public
Overriding Method	default protected public	default ✗ protected public	default ✗ protected ✗ public

```
class Parent{
        public void show(){
                System.out.println("Parent - show method");
        }
}
class Child extends Parent{
        void show(){
                System.out.println("Child - show method");
        }
}
class MethodOverridingDemo{
        public static void main(String args[]){
                Child ob = new Child();
                ob.show();
        }
}
```

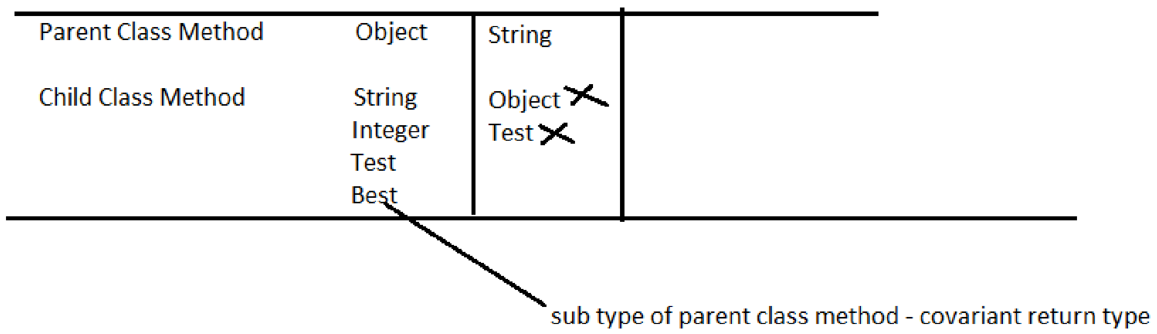
Rule#2

The return types of overridden method & overriding method must be same.

```
class Parent{
        int show(){
                return 11;
        }
}
class Child extends Parent{
        int show(){
                return 10;
        }
}
class MethodOverridingDemo{
        public static void main(String args[]){
                Child ob = new Child();
                ob.show();
        }
}
```

Rule#3

We can use covariant-return types. The return type of overriding method is must be sub-type of overridden method return type this is called covariant return types.



```
class Parent{
        Object show(){
                return null;
        }
}
class Child extends Parent{
        String show(){
                return null;
        }
}
class MethodOverridingDemo{
        public static void main(String args[]){
                Child ob = new Child();
                ob.show();
        }
}
```

Note

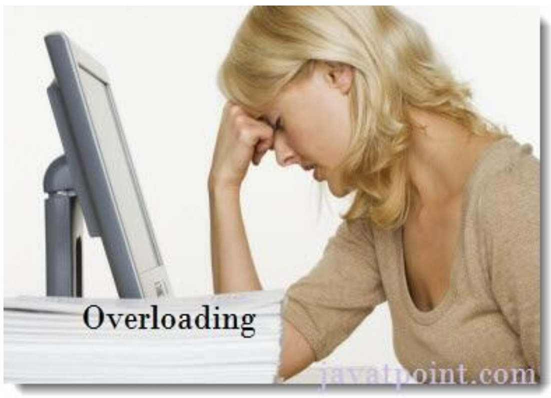
Parent class reference variable is able to hold child class object but Child class reference variable is unable to hold parent class object.

```
class Test{
}

class Best extends Test{

}

class Main{
        public static void main(String args[]){
                Test ob1 = new Test();
                Best ob2 = new Best();
                Test ob3 = new Best();
                Best ob4 = new Test();          // not allowed
        }
}
```

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.



Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```java
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Test it Now

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

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

Test it Now

Output:

```
22
24.9
```

Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```java
class Adder{
static int add(int a,int b){return a+b;}
static double add(int a,int b){return a+b;}
}
class TestOverloading3{
public static void main(String[] args){
System.out.println(Adder.add(11,11));//ambiguity
}}
```

Test it Now

Output:

```
Compile Time Error: method add(int,int) is already defined in class Adder
```

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```java
class TestOverloading4{
public static void main(String[] args){System.out.println("main with String[]");}
public static void main(String args){System.out.println("main with String");}
public static void main(){System.out.println("main without args");}
}
```

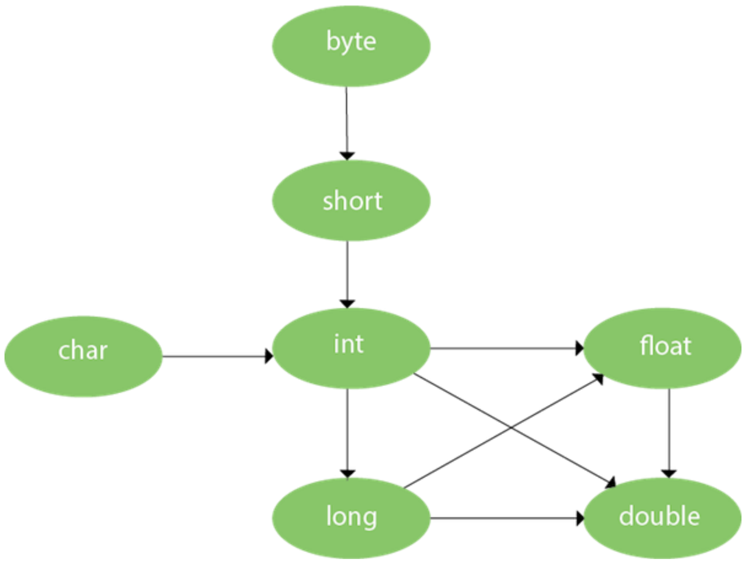
Test it Now

Output:

```
main with String[]
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
class OverloadingCalculation1{
  void sum(int a,long b){System.out.println(a+b);}
  void sum(int a,int b,int c){System.out.println(a+b+c);}

  public static void main(String args[]){
  OverloadingCalculation1 obj=new OverloadingCalculation1();
  obj.sum(20,20);//now second int literal will be promoted to long
  obj.sum(20,20,20);

  }
}
```

Test it Now

```
Output:40
    60
```

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
class OverloadingCalculation2{
  void sum(int a,int b){System.out.println("int arg method invoked");}
  void sum(long a,long b){System.out.println("long arg method invoked");}

  public static void main(String args[]){
  OverloadingCalculation2 obj=new OverloadingCalculation2();
  obj.sum(20,20);//now int arg sum() method gets invoked
  }
}
```

Test it Now

```
Output:int arg method invoked
```

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
class OverloadingCalculation3{
  void sum(int a,long b){System.out.println("a method invoked");}
  void sum(long a,int b){System.out.println("b method invoked");}

  public static void main(String args[]){
  OverloadingCalculation3 obj=new OverloadingCalculation3();
  obj.sum(20,20);//now ambiguity
  }
}
```

Test it Now

```
Output:Compile Time Error
```

Polymorphism in Java

- **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
- Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms.
- So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.

Runtime Polymorphism in Java

- Dynamic method dispatch is a technique by which call to a overridden method is resolved at runtime.
- When an overridden method is called by a reference, then which version of overridden method is to be called is decided at runtime according to the type of object it holds.
- Dynamic method dispatch is performed by JVM not compiler.

```
class Test{
	void show(){
		System.out.println("Test class - show method");
	}
}
class Best extends Test{
	void show(){
		System.out.println("Best class - show method");
	}
}

class Main{
	public static void main(String args[]){
		Test ob = new Best();
		ob.show();
	}
}
```

Final Modifier

- Final keyword provides restriction.
- Final is the modifier applicable for classes, methods and variables.

final class

- If a class is declared as final, then we cannot create child class for that class.
- It prevents inheritance.

```
final class Test{
	void show(){
		System.out.println("Test class - show method");
	}
}
class Best extends Test{
	void show(){
		System.out.println("Best class - show method");
	}
}

class Main{
	public static void main(String args[]){
		Test ob = new Best();
		ob.show();
	}
}
```

final method

- If a method declared as a final we cannot override that method in child class.
- It prevents overriding.

```
class Test{
	final void show(){
		System.out.println("Test class - show method");
	}
}
class Best extends Test{
	void show(){
		System.out.println("Best class - show method");
	}
}

class Main{
	public static void main(String args[]){
		Test ob = new Best();
		ob.show();
	}
}
```

final variable

- If a variable declared as a final we cannot reassign that variable.

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

Super Keyword in Java

- super keyword is used to refer parent class object.
- super can be used to invoke parent class instance variable.
- super can be used to invoke parent class method.
- super() can be used to invoke parent class constructor.

Example#1

```
class Parent{
	void show(){
		System.out.println("Parent - show method");
	}
}
class Child extends Parent{
	void show(){
		super.show();
		System.out.println("Child - show method");
	}
}
class MethodOverridingDemo{
	public static void main(String args[]){
		Child ob = new Child();
		ob.show();
	}
}
```

Example#2

```
class Parent{
	Parent(){
		System.out.println("Parent - constructor");
	}
}
class Child extends Parent{
	Child(){
		super();
		System.out.println("Child - constructor");
	}
}
class MethodOverridingDemo{
	public static void main(String args[]){
		Child ob = new Child();
	}
}
```

Example#3

```
class Test
{
	int a=10,b=30;
}
class Best extends Test
{
	int a = 100,b=300;
	void add(int a,int b)
	{
		System.out.println(a+b);
		System.out.println(this.a+this.b);
		System.out.println(super.a+super.b);
	}
}
class Main
{
	public static void main(String[] args)
	{
		Best ob = new Best();
		ob.add(1,3);
	}
}
```

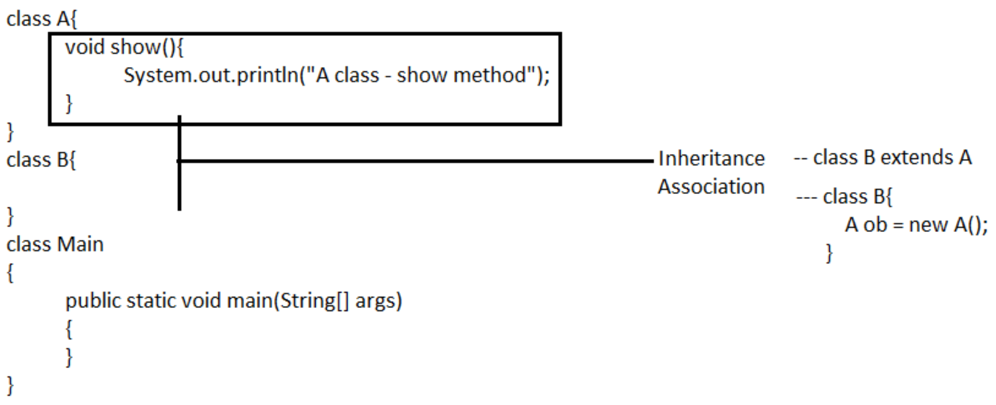
Is-A Relationship

- Is-A relationship also known as inheritance.
- By using extends keyword we can implement inheritance.

```
class Vehicle
{

}
class Car extends Vehicle
{

}
```

Has-A Relationship

- Has-A relationship also known as association.
- class A is in Has-A relationship with class B if class A holds reference of Class B.



```
class A{
	void show(){
		System.out.println("A class - show method");
	}
}
class B{

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

```
class A{
	void show(){
		System.out.println("A class - show method");
	}
}
class B{
	void display(){
		A ob = new A();
		ob.show();
	}
}
class Main
{
	public static void main(String[] args)
	{
		B ob = new B();
		ob.display();
	}
}
```

Defined Method/ Concreate Method

- Method with body is called defined/ concreate method.

```
void add(int x, int y)
{
        //body of the method
}
```

Undefined Method/ Abstract Method

- Method without body is called undefined/ abstract methof.

```
void add(int x, int y);
```

Concreate Class

- The class which contains all defined methods is called concreate class.

Example:

```
class Test{
    void show(){
        // ............
    }
    void add(int x, int y){
        // ..........
    }
}
```

Abstract Class

- The class which can contain both defined and undefined method is called abstract class.

Example:

```
abstract class Test{
        void show(){

        }
        void add(int x, int y);
}
```

Interface

- Interface contains only undefined method. (upto java 7)

Example: chod diya hai

```
interface Test{
        void show();
        void display();
        void add(int x,int y);
}
```

Abstract class in Java

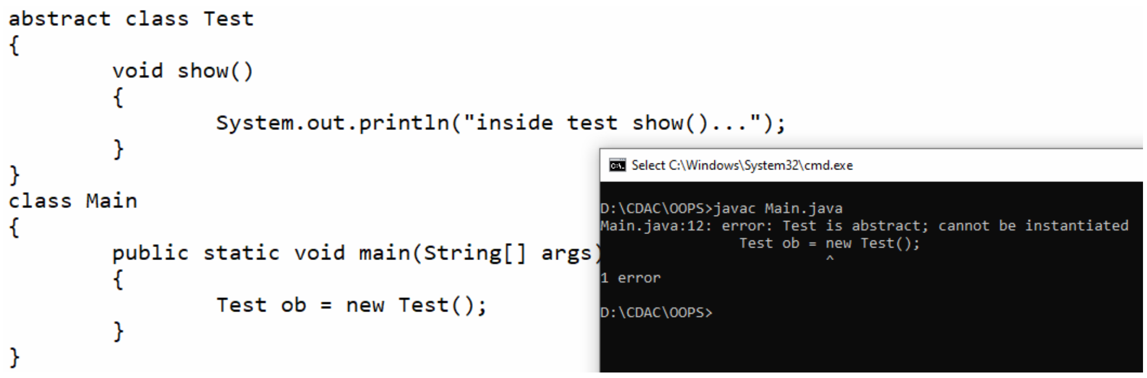
- Abstract class is a java class which contains both concrete and abstract method.
- abstract modifier is used to create abstract class.

Abstract class with only concrete method	Abstract class with both concrete and abstract method	Abstract class with only abstract method
`abstract class Test { void show() { } void display() { } }`	`abstract class Test { abstract void show(); void display() { } }`	`abstract class Test { abstract void show(); abstract void display(); }`

Abstract Class cannot be Instantiated

- Abstract classes are partially implemented classes hence object creation is not possible.

Example#1



```
abstract class Test
{
        void show()
        {
                System.out.println("inside test show()...");
        }
}
class Main
{
        public static void main(String[] args)
        {
                Test ob = new Test();
        }
}
```

```
D:\CDAC\OOPS>javac Main.java
Main.java:12: error: Test is abstract; cannot be instantiated
                Test ob = new Test();
                          ^
1 error

D:\CDAC\OOPS>
```

Example#2

```
abstract class Test{
        void show(){
                System.out.println("inside test show()...");
        }
}
class Best extends Test{

}
class Main
{
        public static void main(String[] args)
        {
                Best ob = new Best();
                ob.show();
        }
}
```

Example#3

```
abstract class Test{
        void show(){
                System.out.println("inside test show()...");
        }
        abstract void display();
}
class Best extends Test{
        void display(){
                System.out.println("inside best display()...");
        }
}
class Main
{
	public static void main(String[] args)
	{
		Best ob = new Best();
		ob.show();
		ob.display();
	}
}
```

Note:

- If abstarct class contains abstract method then we need to override in it's child class otherwise we have to make child class as abstract.

Example#4

```
abstract class Test{
	abstract void show();
	abstract void display();
}
class Best extends Test{
	void show(){
		System.out.println("inside show()...");
	}
	void display(){
		System.out.println("inside display()...");
	}
}
class Main
{
	public static void main(String[] args)
	{
		Best ob = new Best();
		ob.show();
		ob.display();
	}
}
```

Partial Implementation

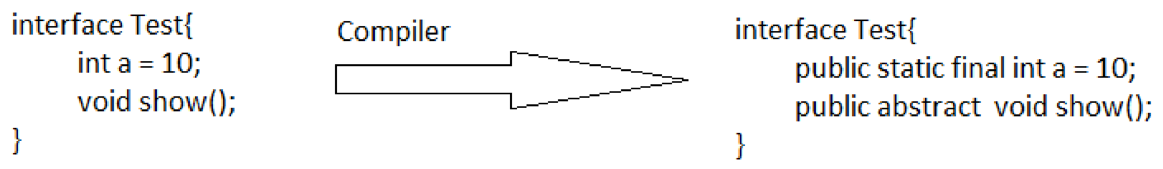
- If the child class is unable to provide the implementation of all abstract methods of parent class then declare that class with abstract and take one more child class to complete the implementation of remaining abstract methods.

Exaample#6

```
abstract class Test
{
        abstract void show1();
        abstract void show2();
}
abstract class Best extends Test
{
        void show1()
        {
                System.out.println("inside best show1()...");
        }
}
class Nest extends Best
{
        void show2()
        {
                System.out.println("inside best show2()...");
        }
}
class Main
{
        public static void main(String[] args)
        {
                Nest ob = new Nest();
                ob.show1();
                ob.show2();
        }
}
```

Interface in Java

- Interface is also one type of java class which contains only abstract method.
- interface keyword is used to create interface.
- Interface cannot be instantiated.
- Every interface methods are by default ***public abstract*** and variables are ***public static*** *and* ***final***.



```
interface Test{
      int a = 10;
      void show();
}
```

```
interface Test{
      public static final int a = 10;
      public abstract  void show();
}
```

Example#1

```
interface Test
{
        int a = 10;
        void show();
}
```

```
class Best implements Test{
	public void show(){
		System.out.println("inside show()...");
	}
}
class Main
{
	public static void main(String[] args)
	{
		Best ob = new Best();
		ob.show();
	}
}
```

Nested Interface in Java

- Declaring interface inside the class or abstract class or interface is called nested interface.

Example#1

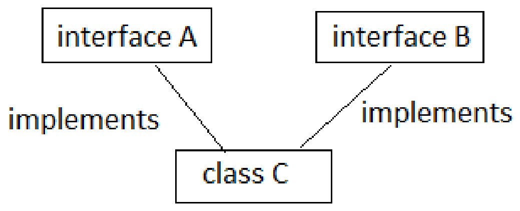
```
class Test
{
	interface intf1
	{
		void show();
	}
}
class Best implements Test.intf1
{
	public void show()
	{
		System.out.println("show method..");
	}
}

class Main
{
	public static void main(String[] args)
	{
		Best ob = new Best();
		ob.show();
	}
}
```

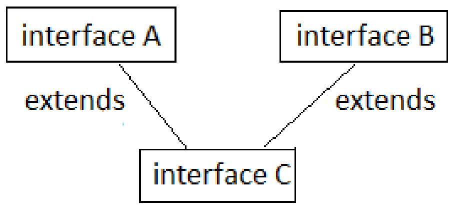
Example#2

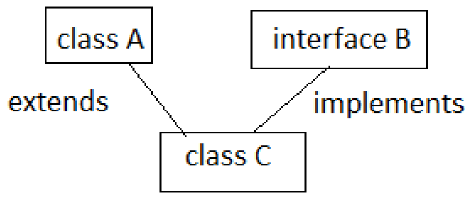
```
interface Test
{
    interface intf1
    {
        void show();
    }
}
class Best implements Test.intf1
{
    public void show()
    {
        System.out.println("show method..");
    }
}

class Main
{
    public static void main(String[] args)
    {
        Best ob = new Best();
        ob.show();
    }
}
```

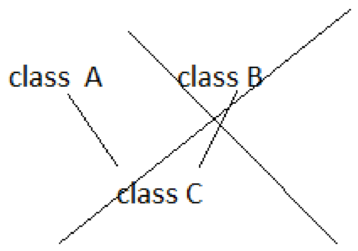
Multiple inheritance in Java by interface

- An interface can extend more than one parent interface. The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.









Example#1

```
interface A{
    void show1();
}
interface B{
	void show2();
}
class C implements A,B{
	public void show1(){
		System.out.println("show1 method...");
	}
	public void show2(){
		System.out.println("show2 method...");
	}
}
class Main{
	public static void main(String args[]){
		C ob = new C();
		ob.show1();
		ob.show2();
	}
}
```

Example#2

```
interface A{
	void show1();
}
interface B{
	void show2();
}
interface C extends A, B{

}
class D implements C{
	public void show1(){
		System.out.println("show1 method...");
	}
	public void show2(){
		System.out.println("show2 method...");
	}
}
class Main{
	public static void main(String args[]){
		D ob = new D();
		ob.show1();
		ob.show2();
	}
}
```

Example#3

```
class A{
	void show1(){
		System.out.println("show1 method...");
	}
}
interface B{
	void show2();
}
class C extends A implements B {
	public void show2(){
		System.out.println("show2 method...");
	}
}
class Main{
	public static void main(String args[]){
		C ob = new C();
		ob.show1();
		ob.show2();
	}
}
```

Marker interface

- An interface that has no members (methods and variables) is called as marker interface.
- Marker interface is used to inform the JVM that the classes implementing them will have some special behavior.
- User defined empty interfaces are not a marker interfaces, only predefined empty interfaces are marker interfaces.

New interface features (Java 8)

Default Method

Default method allows the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

Static Method

Static methods cannot be override in the classes that implements these interfaces.

Example#1

```
interface A{
	void show1();
	default void show2(){
		System.out.println("show2...");
	}
```

```
		static void show3(){
			System.out.println("show3...");
		}
}
class B implements A{
		public void show1(){
			System.out.println("show1...");
		}
}
class Main{
		public static void main(String args[]){
			B ob = new B();
			ob.show1();
			ob.show2();
			A.show3();
		}
}
```

Java Package

- Package is a folder that contains group of related classes and interface.

Advantage of Java Package

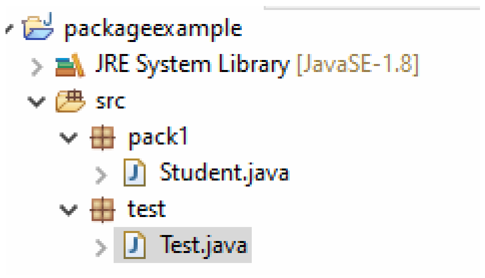
- Accessing is Faster
- It resolves naming conflict

Types of packages

- Predefined Package
    - The packages which are created by the vendor is called predefined package.
- User Defined Package
    - The packages which are created by the programmer is called user defined package.

Creation of User Defined Package

- Create a folder with any name (e.g. pack). The package name and folder name must be same.
- Write a program and save it inside package.



Student.java

```
package pack1;

public class Student
{
    public void display()
    {
        System.out.println("Inside package : display()...");
    }
}
```

How to access user defined package

Test.java

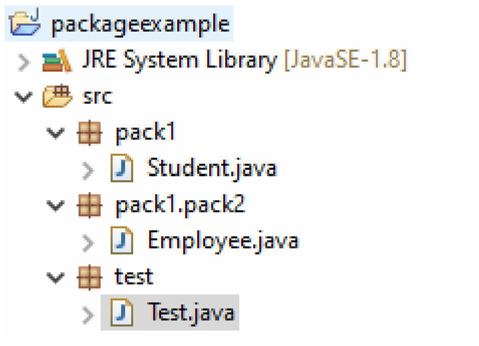
```
package test;

import pack1.Student;

public class Test {
    public static void main(String[] args) {
        Student s = new Student();
        s.display();
    }
}
```

Note

- Inside the source file it is possible to declare only one package statement and that statement must be first statement of the source file.

valid
package student;
import java.io.*;
invalid
import java.io.*;
package student;
invalid
package student1;
package student2;

Sub Package



```
packageexample
> JRE System Library [JavaSE-1.8]
v src
  v pack1
    > Student.java
  v pack1.pack2
    > Employee.java
  v test
    > Test.java
```

Employee.java

```java
package pack1.pack2;

public class Employee {
    public void show() {
        System.out.println("inside show method...");
    }
}
```

Test.java

```java
package test;

import pack1.pack2.Employee;

public class Test {
    public static void main(String[] args) {
        Employee ob = new Employee();
        ob.show();
    }
}
```

Access Modifiers in Java

- Access modifier decides the visibility of class members.
- Java support following access modifier for visibility:
    - Private
    - Default
    - Protected
    - Public
- **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
- **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
- **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
- **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
-

Understanding Java Access Modifiers

	Private	**Default**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package sub class	No	Yes	Yes	Yes
Same package non sub class	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non sub class	No	No	No	Yes

Example#1

```
package test;

class Demo {
        private int a = 10;
        int b = 20;
        protected int c = 30;
        public int d = 40;

        void show() {
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}

public class Test {
        public static void main(String[] args) {
                Demo ob = new Demo();
                ob.show();
        }
}
```

Example#2

```
package test;

class Demo {
        private int a = 10;
        int b = 20;
        protected int c = 30;
        public int d = 40;
}
class Best extends Demo{
        void show() {
                //System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}

public class Test {
        public static void main(String[] args) {
                Best ob = new Best();
                ob.show();
        }
}
```

Example#3

```
package test;

class Demo {
	private int a = 10;
	int b = 20;
	protected int c = 30;
	public int d = 40;
}
class Best{
	void show() {
		Demo ob = new Demo();
		//System.out.println("a = " + ob.a);
		System.out.println("b = " + ob.b);
		System.out.println("c = " + ob.c);
		System.out.println("d = " + ob.d);
	}
}

public class Test {
	public static void main(String[] args) {
		Best ob = new Best();
		ob.show();
	}
}
```

Example#4

```
package pack1;

public class Student
{
	private int a = 10;
	int b = 20;
	protected int c = 30;
	public int d = 40;
}
```

```
package test;

import pack1.Student;

class Best extends Student{
	void show() {
```

```
                //System.out.println("a = " + a);
                //System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
        }
}

public class Test {
        public static void main(String[] args) {
                Best ob = new Best();
                ob.show();
        }
}
```

Example#5

```
package pack1;

public class Student
{
        private int a = 10;
        int b = 20;
        protected int c = 30;
        public int d = 40;
}



package test;

import pack1.Student;

class Best {
        void show() {
                Student ob = new Student();
                //System.out.println("a = " + ob.a);
                //System.out.println("b = " + ob.b);
                //System.out.println("c = " + ob.c);
                System.out.println("d = " + ob.d);
        }
}

public class Test {
        public static void main(String[] args) {
                Best ob = new Best();
                ob.show();
```

```
	}
}
```

Static Import

If we are using the static import it is possible to call static variables and static methods of a particular class directly without using class name.

```
package pack1;

public class Student
{
	public static int a = 10;
	public static void show() {
		System.out.println("inside show method");
	}
}
```

```
package test;

import static pack1.Student.*;

public class Test {
	public static void main(String[] args) {
		System.out.println(a);
		show();
	}
}
```

Encapsulation in Java

- **Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several medicines.

- We can create a fully encapsulated class in Java by making all the data members of the class private.
- Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of a fully encapsulated class.

Advantage of Encapsulation in Java

- By providing only a setter or getter method, you can make the class **read-only or write-only**.
- In other words, you can skip the getter or setter methods.
- It provides you the **control over the data**.
- Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method.
- You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.

The encapsulate class is **easy to test**. So, it is better for unit testing.

The standard IDE's are providing the facility to generate the getters and setters. So, it is **easy and fast to create an encapsulated class** in Java.

Simple Example of Encapsulation in Java

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

File: Student.java

```java
//A Java class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
package com.javatpoint;
```

```java
public class Student{
//private data member
private String name;
//getter method for name
public String getName(){
return name;
}
//setter method for name
public void setName(String name){
this.name=name
}
}
```

File: Test.java

```java
//A Java class to test the encapsulated class.
package com.javatpoint;
class Test{
public static void main(String[] args){
//creating instance of the encapsulated class
Student s=new Student();
//setting value in the name member
s.setName("vijay");
//getting value of the name member
System.out.println(s.getName());
}
}
```

```
Compile By: javac -d . Test.java
Run By: java com.javatpoint.Test
```

Output:

```
vijay
```

Read-Only class

```java
//A Java class which has only getter methods.
public class Student{
//private data member
private String college="AKG";
//getter method for college
public String getCollege(){
return college;
}
}
```

Now, you can't change the value of the college data member which is "AKG".

```java
s.setCollege("KITE");//will render compile time error
```

Write-Only class

```java
//A Java class which has only setter methods.
public class Student{
//private data member
private String college;
//getter method for college
public void setCollege(String college){
this.college=college;
}
}
```

Now, you can't get the value of the college, you can only change the value of college data member.

```java
System.out.println(s.getCollege());//Compile Time Error, because there is no such me
thod
System.out.println(s.college);//Compile Time Error, because the college data membe
r is private.
//So, it can't be accessed from outside the class
```

Another Example of Encapsulation in Java

Let's see another example of encapsulation that has only four fields with its setter and getter methods.

File: Account.java

```java
//A Account class which is a fully encapsulated class.
//It has a private data member and getter and setter methods.
```

```java
class Account {
//private data members
private long acc_no;
private String name,email;
private float amount;
//public getter and setter methods
public long getAcc_no() {
    return acc_no;
}
public void setAcc_no(long acc_no) {
    this.acc_no = acc_no;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getEmail() {
    return email;
}
public void setEmail(String email) {
    this.email = email;
}
public float getAmount() {
    return amount;
}
public void setAmount(float amount) {
    this.amount = amount;
}

}
```

File: TestAccount.java

```java
//A Java class to test the encapsulated class Account.
public class TestEncapsulation {
```

```
public static void main(String[] args) {
    //creating instance of Account class
    Account acc=new Account();
    //setting values through setter methods
    acc.setAcc_no(7560504000L);
    acc.setName("Sonoo Jaiswal");
    acc.setEmail("sonoojaiswal@javatpoint.com");
    acc.setAmount(500000f);
    //getting values through getter methods
    System.out.println(acc.getAcc_no()+" "+acc.getName()+" "+acc.getEmail()+" "+acc.
getAmount());
}
}
```

Test it Now

Output:

```
7560504000 Sonoo Jaiswal sonoojaiswal@javatpoint.com 500000.0
```

GARBAGE Collection

- Garbage Collection is the process of destroying unused objects from heap memory.
- Garbage collection is done by Garbage collector.
- JVM calls garbage collector randomly.
- Programmer can request JVM to run garbage collector.
- Unused are eligible for garbage collection.

Different ways to make object eligible for garbage collection

- Nullifying a reference variable
- Re-assigning a reference variable
- Island of isolation

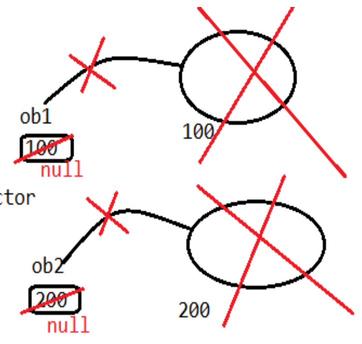
Nullifying a reference variable

- If an object no longer required then assign null to all its reference variables, this approach is called nullifying a reference variable.

Example#1

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Test s1 = new Test();
        Test s2 = new Test();
        s1 = null;
        s2 = null;
        System.gc();
        Thread.sleep(2000);
        System.out.println("End of main method");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalized method called");
    }
}
```

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Test ob1 = new Test();
        Test ob2 = new Test(); ———— Used Object: 2 Unused Object: 0
        ob1 = null;——————————————— Used Object:1, Unused Object: 1
        ob2 = null;——————————————— Used Object: 0, Unused Object:2
        System.gc(); ————————————— Requesting JVM to call Garbage Collector
        Thread.sleep(3000); —————— Waiting for 3sec
        System.out.println("End of main method");
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalized method called");
    }
}
```



o/p:

```
finalized method called
finalized method called
End of main method
```

Re-assigning a reference variable

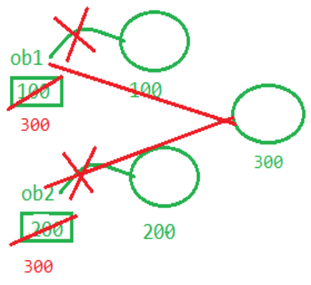
- If an object no longer required then reassign it's reference variable to some other object.

Example#2

```
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Test s1 = new Test();
        Test s2 = new Test();
        s1 = new Test();
        s2 = s1;
        System.gc();
        Thread.sleep(2000);
        System.out.println("End of main method");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalized method called");
    }
}
```

```java
public class Test {
    public static void main(String[] args) throws InterruptedException {
        Test ob1 = new Test();
        Test ob2 = new Test(); ————— Used Object: 2, Unused Object : 0
        ob1 = new Test(); ———————— Used Object: 2, Unused Object: 1
        ob2 = ob1; ——————————————— Used Object: 1, Unused Object: 2
        System.gc();—————————————— Request JVM to call GC
        Thread.sleep(3000);——————— Wait for 3 sec
        System.out.println("End of main method");
    }
    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalized method called");
    }
}
```

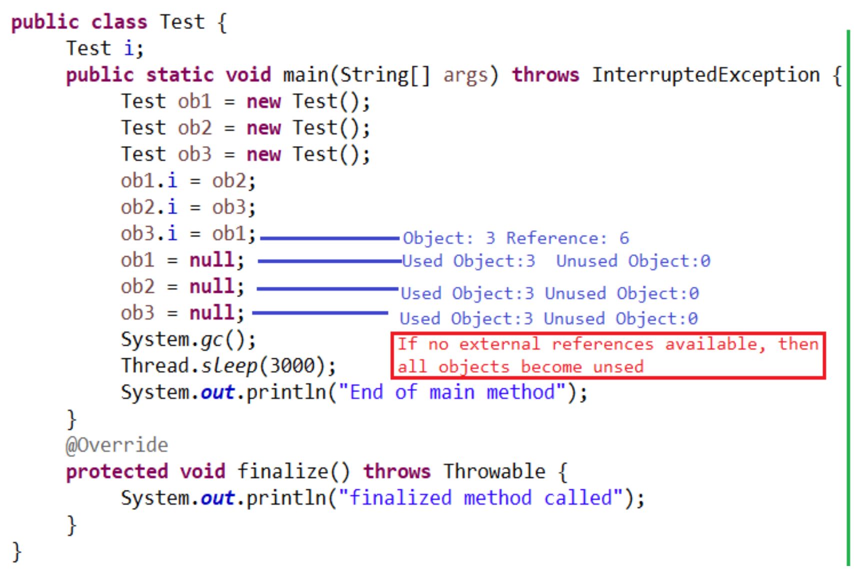
```
o/p:
finalizied method called
finalizied method called
End of main  method
```



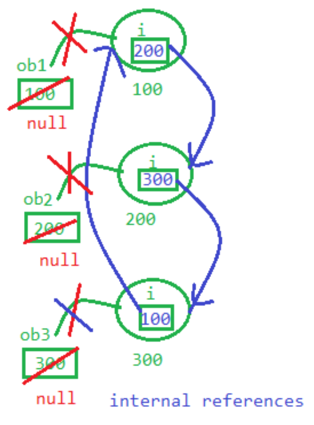
Island of isolation

- A group of objects that reference each other but are not referenced by any active/external object in the application.

```java
public class Test {
    Test i;
    public static void main(String[] args) throws InterruptedException {
        Test ob1 = new Test();
        Test ob2 = new Test();
        Test ob3 = new Test();
        ob1.i = ob2;
        ob2.i = ob3;
        ob3.i = ob1;
        ob1 = null;
        ob2 = null;
        ob3 = null;
        System.gc();
        Thread.sleep(2000);
        System.out.println("End of main method");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("finalized method called");
    }
}
```



```
public class Test {
      Test i;
      public static void main(String[] args) throws InterruptedException {
            Test ob1 = new Test();
            Test ob2 = new Test();
            Test ob3 = new Test();
            ob1.i = ob2;
            ob2.i = ob3;
            ob3.i = ob1;  ————————Object: 3 Reference: 6
            ob1 = null;  ————————Used Object:3  Unused Object:0
            ob2 = null;  ————————Used Object:3 Unused Object:0
            ob3 = null;  ———————— Used Object:3 Unused Object:0
            System.gc();           If no external references available, then
            Thread.sleep(3000);    all objects become unsed
            System.out.println("End of main method");
      }
      @Override
      protected void finalize() throws Throwable {
            System.out.println("finalized method called");
      }
}
```



Requesting JVM to run garbage collection

- By Using System.gc()
- By Using Runtime.getRuntime().gc()

Finalization

- Garbage collector calls finalize() method before destroying object to perform cleanup operation.
- finalize() method present in object class with following declartion

  protected void finalize()

Wrapper classes in Java

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

Use of Wrapper classes in Java

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

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

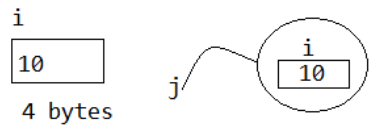
- The eight classes of the *java.lang* package are known as wrapper classes in Java. The list of eight wrapper classes are given below:
-

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
char	Char
boolean	Boolean

Autoboxing

- Conversion of primitive data type into its corresponding wrapper class/object type is known as autoboxing.

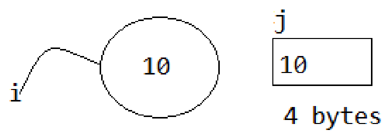
```
public class AutoBoxingDemo {
    public static void main(String[] args) {
        int i =10;
        Integer j = i; ———Autoboxing
        System.out.println("i = "+i);
        System.out.println("j = "+j);
    }
}
```



Unboxing

- Conversion of wrapper class/ object type into its corresponding primitive type is known as unboxing.

```
public class UnboxingDemo {
    public static void main(String[] args) {
        Integer i = new ~~Integer(10)~~;
        int j = i;
        System.out.println(i);
        System.out.println(j);
    }
}
```



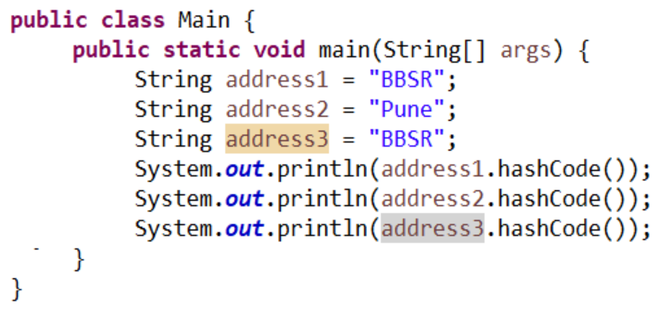
Java String

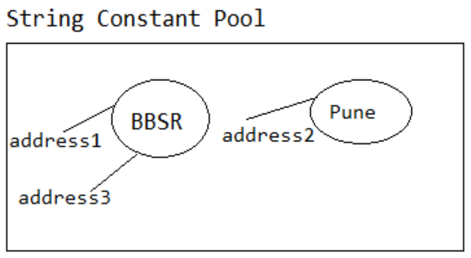
- String is used to represent group of characters enclosed with in the double quotes.

```
public class Test {
    public static void main(String[] args) {
        String address1 = "BBSR";
        String address2 = new String("PUNE");
        System.out.println(address1);
        System.out.println(address2);
    }
}
```

Creating a string object without using new operator

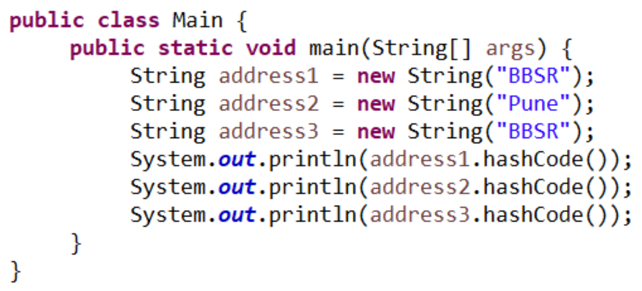
- When we create String object without using new operator the objects are created in SCP (String constant pool) area.
- When we create object in SCP area then just before object creation it is always checking previous objects.
    - If the previous object is available with the same content then it won't create new object that reference variable pointing to existing object.
    - If the previous objects are not available then JVM will create new object.

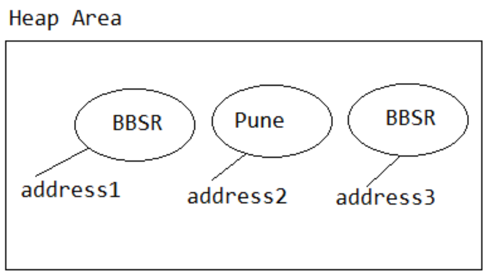

```
public class Main {
     public static void main(String[] args) {
          String address1 = "BBSR";
          String address2 = "Pune";
          String address3 = "BBSR";
          System.out.println(address1.hashCode());
          System.out.println(address2.hashCode());
          System.out.println(address3.hashCode());
     }
}
```



Creating a string object by using new operator

- Whenever we are creating String object by using new operator the object created in heap area.
- When we create object in Heap area instead of checking previous objects it directly creates objects.


```
public class Main {
     public static void main(String[] args) {
          String address1 = new String("BBSR");
          String address2 = new String("Pune");
          String address3 = new String("BBSR");
          System.out.println(address1.hashCode());
          System.out.println(address2.hashCode());
          System.out.println(address3.hashCode());
     }
}
```



By Using == operator

- It is comparing reference type. If two reference variables are pointing to same object then it returns true otherwise false.

Example#1

```
public class Main {
        public static void main(String[] args) {
                String address1 = "BBSR";
                String address2 = "Pune";
                String address3 = "BBSR";
                System.out.println(address1 ==  address2);
                System.out.println(address1 ==  address3);
        }
}
```

Example#2

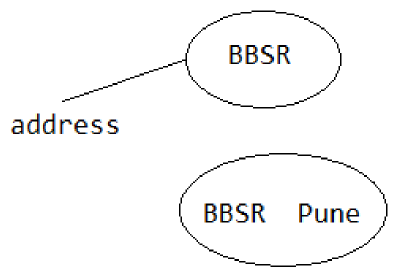
```
public class Main {
        public static void main(String[] args) {
                String address1 = new String("BBSR");
                String address2 = new String("Pune");
                String address3 = new String("BBSR");
                System.out.println(address1 ==  address2);
                System.out.println(address1 ==  address3);
        }
}
```

Immutable String in Java

- String is immutable, it means once we are creating String objects it is not possible to perform modifications on existing object.

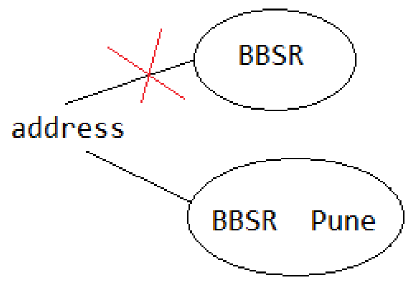
Example#1

```
public class Main {
    public static void main(String[] args) {
        String address = "BBSR";
        address.concat(" Pune");
        System.out.println(address);
    }
}
```



Example#2

```
public class Main {
    public static void main(String[] args) {
        String address = "BBSR";
        address = address.concat(" Pune");
        System.out.println(address);
    }
}
```



StringBuffer

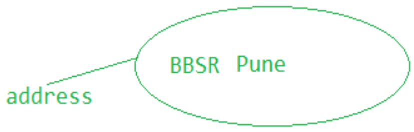
- StringBuffer is used to represents group of character like String.
- We are able to create StringBuffer object by using new operator.

```
public class Test {
    public static void main(String[] args) {
        StringBuffer address = new StringBuffer("BBSR");
        System.out.println(address);
    }
}
```

StringBuffer Mutability

- StringBuffer is mutable, it means once we are creating StringBuffer objects on that existing object it is possible to perform modification.

```
public class Main {
    public static void main(String[] args) {
        StringBuffer address = new StringBuffer("BBSR");
        address.append(" Pune");
        System.out.println(address);
    }
}
```

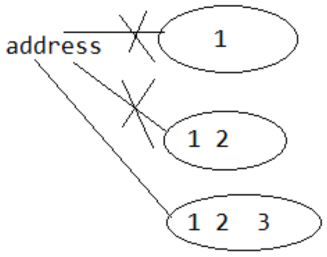


StringBuilder

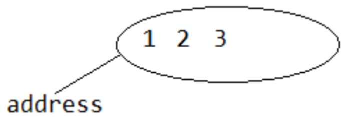
- StringBuilder is identical to StringBuffer except one important difference.
- *Every method present in the StringBuffer is Synchronized but method in StringBuilder is not Synchronized.*
- Multiple threads are allow to operate on StringBuilder methods hence the performance of the application is increased.
- Not Synchronized means not thread safe.

Which one is better String or String Buffer?

```
String address = "1";
address = address.concat("2");
address = address.concat("3");
```



```
StringBuffer address = new StringBuffer("1");
address.apppend("2");
address.apppend("3");
```



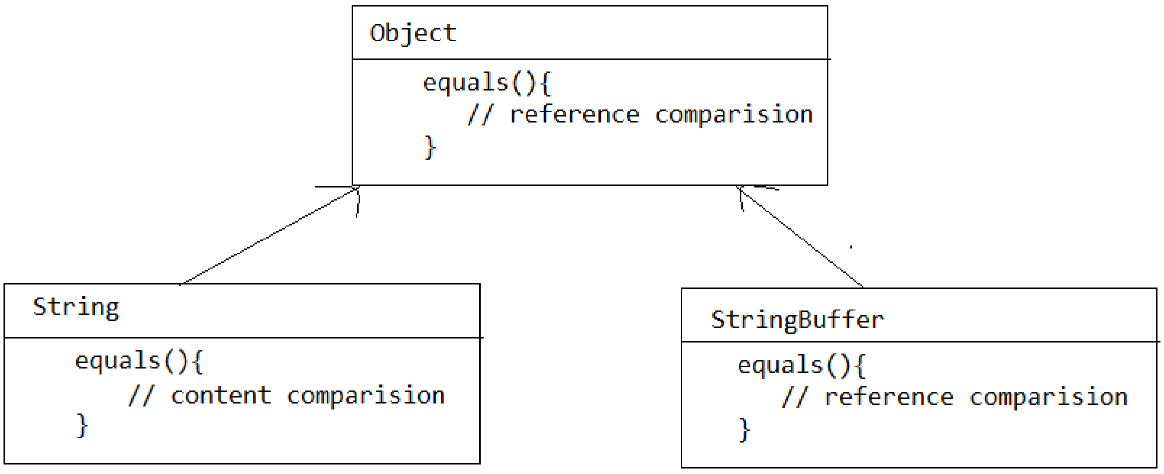
StringBuffer is better based on Memory and Performace.

Which one is better StringBuffer or StringBuilder?

StringBuffer	StringBuilder
- All methods are synchronized.	- All methods are not synchronized.
- One thread can execute at a time.	- Multiple thread can execute simultaneously.
- performace is less	- Performace is more.
- thread safe	- Thread unsafe.

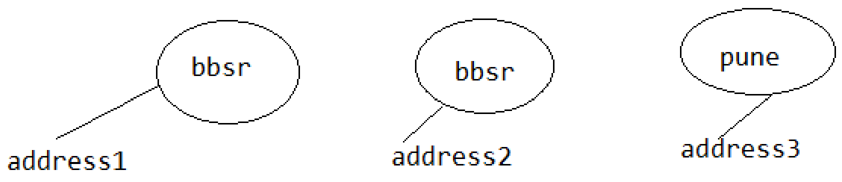
String equals()

- equals() method present in object used for reference comparison.
- String is child class of object and it is overriding equals( ) methods used for content comparison.
- StringBuffer class is child class of object and it is not overriding equals() method hence it is using parent class equals() method.



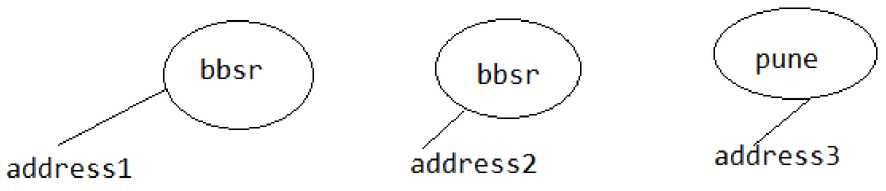
Example#1

```
public class EqualsMethodDemo {
    public static void main(String[] args) {
        String address1 = new String("bbsr");
        String address2 = new String("bbsr");
        String address3 = new String("pune");
        System.out.println(address1 == address2); # false
        System.out.println(address1 == address3); # false
        System.out.println(address1.equals(address2));# true
        System.out.println(address1.equals(address3));# false
    }
}
```



Example#2

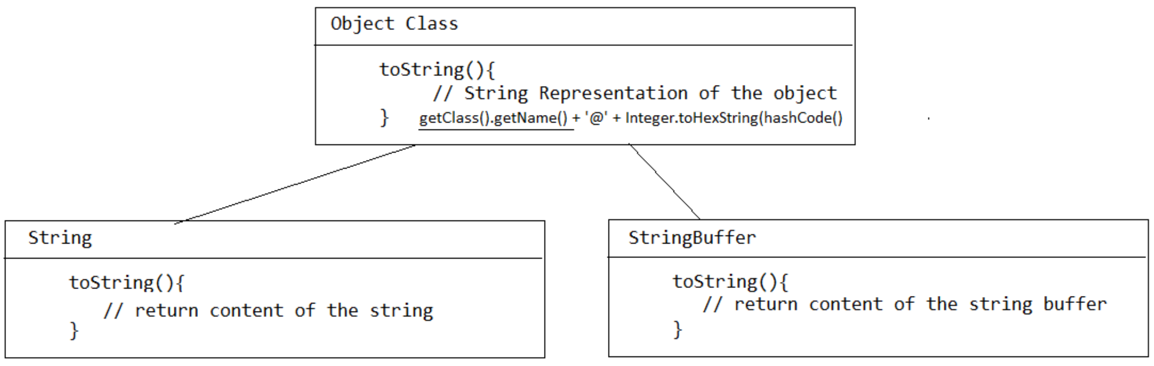
```
public class EqualsMethodDemo {
	public static void main(String[] args) {
		StringBuffer address1 = new StringBuffer("bbsr");
		StringBuffer address2 = new StringBuffer("bbsr");
		StringBuffer address3 = new StringBuffer("pune");
		System.out.println(address1 == address2);# false
		System.out.println(address1 == address3);# false
		System.out.println(address1.equals(address2));# false
		System.out.println(address1.equals(address3));# false
	}
}
```



```
== : Referece Comparision
     If both reference are poiniting to same object then return  true otherwise fasle.

equlas() of object class :  Referece Comparision
equlas() of string class :  Content Comaparision
equlas() of StringBuffer class :  Referece Comparision
```

tostring()

- toString( ) method present in object class returns a string representation of the object.
- String is overriding toString() used to return content of the String.
- StringBuffer is overriding toString() used to return content of the StringBuffer.


Object Class
toString(){
// String Representation of the object
}
getClass().getName() + '@' + Integer.toHexString(hashCode()
String
toString(){
// return content of the string
}
StringBuffer
toString(){
// return content of the string buffer
}

Note:

- Whenever we are printing reference variable internally it is calling toString() method.

```
class Object
{
        public String toString()
        {
                return getClass().getName() + '@' + Integer.toHexString(hashCode());

        }
}
class String extends Object
{
        public String toString()
        {
                return "content of String";
        }
}
class StringBuffer extends Object
{
        public String toString()
        {
                return "content of String";
        }
}
```

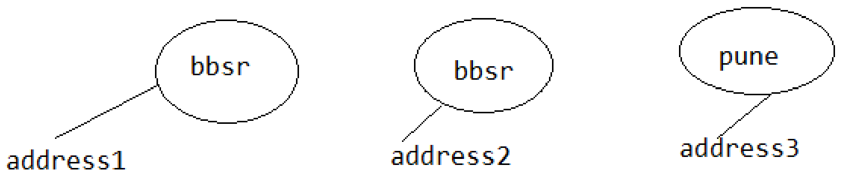
Example

```
class Test{
        int i = 10;
}
public class EqualsMethodDemo {
        public static void main(String[] args) {
                Test t = new Test();
                String address1 = new String("bbsr");
                String address2 = new String("pune");
                System.out.println(t);
                System.out.println(address1);
                System.out.println(address2);
        }
}
```

== operator and equals() method

- == operators for reference comparison and .equals() method for content comparison (e.g. for string).

```
public class EqualsMethodDemo {
    public static void main(String[] args) {
        String address1 = new String("bbsr");
        String address2 = new String("bbsr");
        String address3 = new String("pune");
        System.out.println(address1 == address2); # false
        System.out.println(address1 == address3); # false
        System.out.println(address1.equals(address2));# true
        System.out.println(address1.equals(address3));# false
    }
}
```



String length()

length variable vs. length() method

- length variable used to find length of the Array.
- length() is method used to find length of the String.

Example

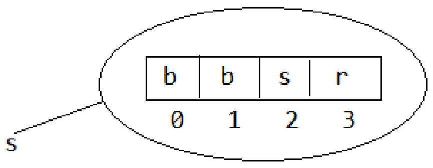
```
public class Main {
        public static void main(String[] args) {
                int arr [] = {10,20,30,40,50};
                String address = "bbsr";
                System.out.println(arr.length);
                System.out.println(address.length());
        }
}
```

String charAt()

- It is used to extract the character from particular index position.

  char charAt(int)

```
public class Main {
    public static void main(String[] args) {
        String s = "bbsr";
        System.out.println(s.charAt(0)); // b
    }
}
```

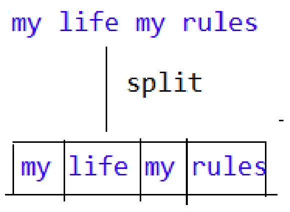


String split()

- It is used to divide string into number of tokens.

  String[] split(String)

```
public class Main {
    public static void main(String[] args) {
        String s = "my life my rules";
        String arr[] = s.split(" ");
        for(String a : arr) {
            System.out.println(a);
        }
    }
}
```

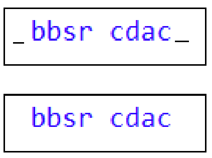


String trim()

- It is used to remove the trail and leading spaces.

  String trim()

```
public class Main {
    public static void main(String[] args) {
        String s = " bbsr cdac ";
        System.out.println(s.length());// 11
        s = s.trim();
        System.out.println(s.length()); // 9
    }
}
```



String toUpperCase() String toLowerCase()

```
public class Test {
    public static void main(String[] args) {
        String s1 = "BBSR";
        String s2 = "bbsr";
        System.out.println(s1.toLowerCase());
        System.out.println(s2.toUpperCase());
    }
}
```

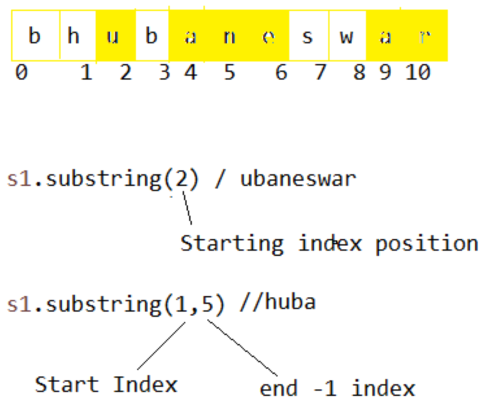
String substring()

- substring() used to find substring in main String.

  String substring(int);

  String substring(int, int)
- while printing substring() it includes starting index & it excludes ending index.

```
public class Main {
     public static void main(String[] args) {
          String s1 = "bhubaneswar";
          String s2 = s1.substring(2);
          String s3 = s1.substring(1,5);
          System.out.println(s2);
          System.out.println(s3);
     }
}
```



Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

What is Exception in Java?

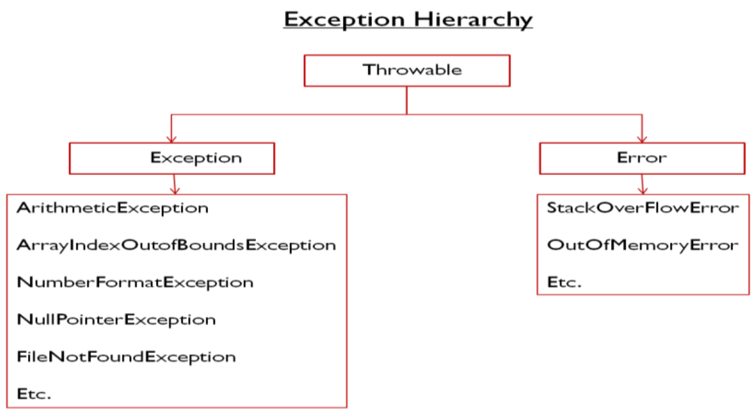
Dictionary Meaning: Exception is an abnormal condition.

- In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is Exception Handling?

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Hierarchy of Java Exception classes



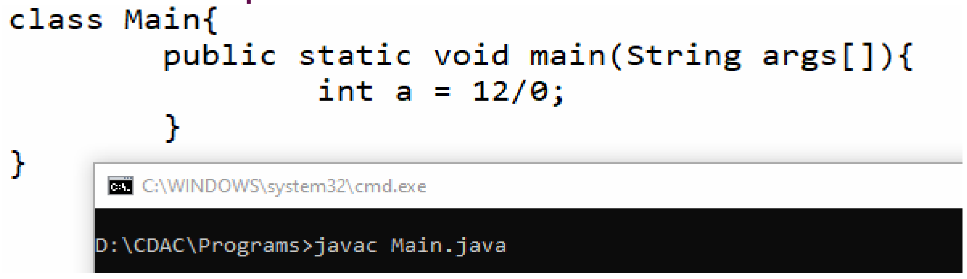
Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception	**Unchecked Exception**
Exception which is checked by the compiler is called checked exception.	Exception which is not checked by the compiler is called checked exception.
Need to either caught or throw explicitly.	No restriction.

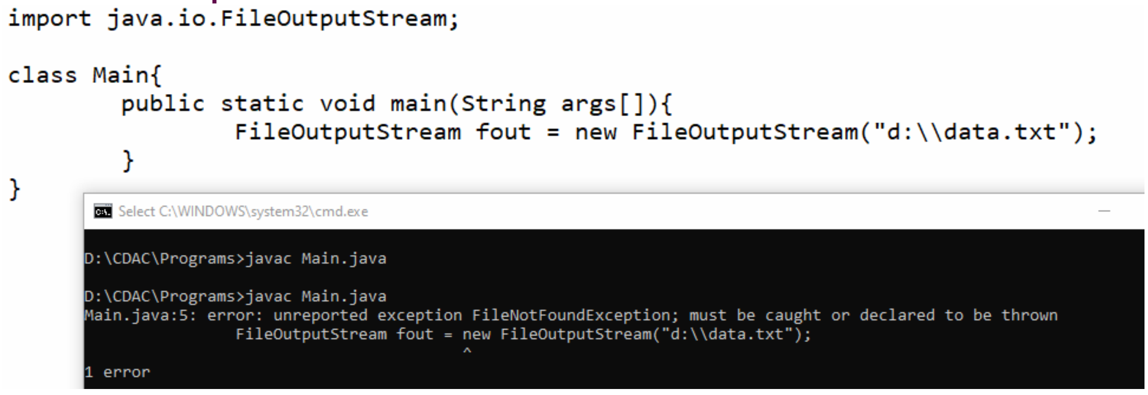
Unchecked Exception



```
class Main{
        public static void main(String args[]){
                int a = 12/0;
        }
}
```

```
C:\WINDOWS\system32\cmd.exe

D:\CDAC\Programs>javac Main.java
```

Checked Exception

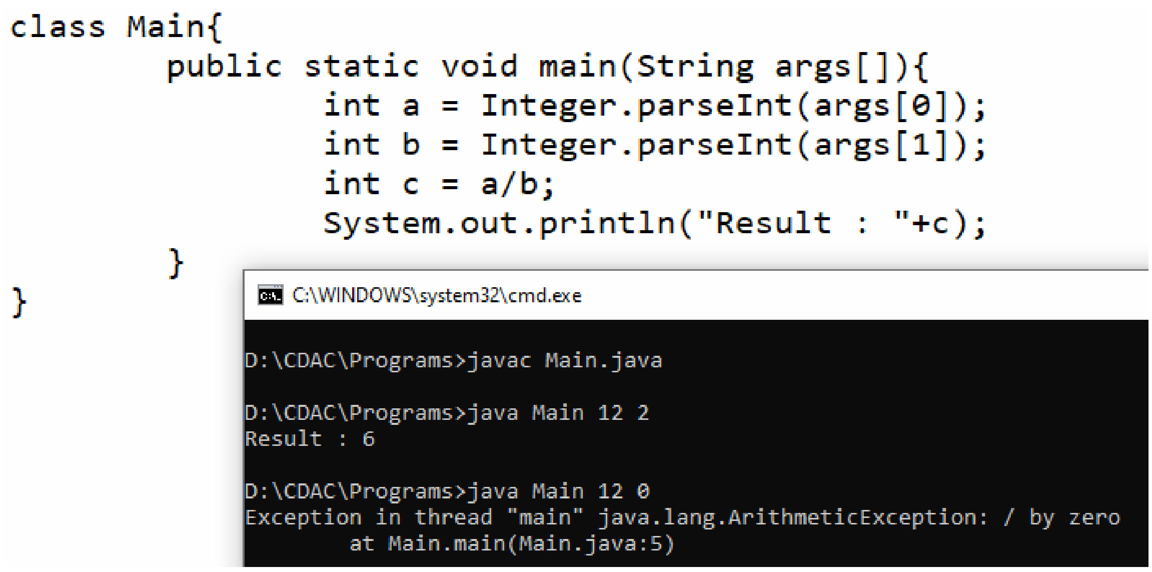


```
import java.io.FileOutputStream;

class Main{
        public static void main(String args[]){
                FileOutputStream fout = new FileOutputStream("d:\\data.txt");
        }
}
```

```
Select C:\WINDOWS\system32\cmd.exe

D:\CDAC\Programs>javac Main.java

D:\CDAC\Programs>javac Main.java
Main.java:5: error: unreported exception FileNotFoundException; must be caught or declared to be thrown
                FileOutputStream fout = new FileOutputStream("d:\\data.txt");
                                        ^
1 error
```

Error

- There are two types of errors occurs in programming language.
- Compile Time Error
  - The error which occurs during the compilation phase of the program is called compile time error.
- Run Time Error
  - The error which occurs during the execution phase of the program is called run time error.
  - Run time error is also known as Exception.



```
class Main{
        public static void main(String args[]){
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int c = a/b;
                System.out.println("Result : "+c);
        }
}
```

```
C:\WINDOWS\system32\cmd.exe

D:\CDAC\Programs>javac Main.java

D:\CDAC\Programs>java Main 12 2
Result : 6

D:\CDAC\Programs>java Main 12 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Main.main(Main.java:5)
```

Common Scenarios of Java Exceptions

1) A scenario where ArithmeticException occurs

- If we divide any number by zero, there occurs an ArithmeticException.

1. int a=50/0;//ArithmeticException

2) A scenario where NullPointerException occurs

- If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

1. String s=**null**;
2. System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

- If the formatting of any variable or number is mismatched, it may result into NumberFormatException.
- Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

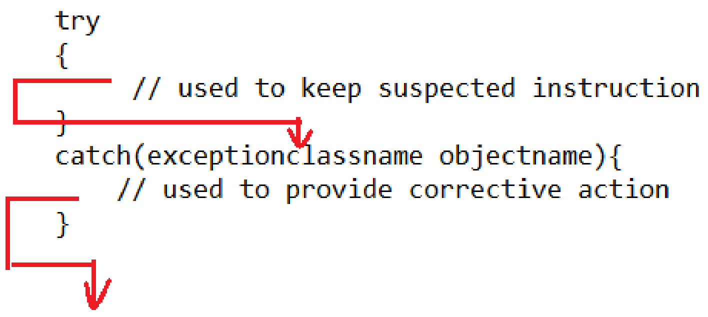
4) A scenario where ArrayIndexOutOfBoundsException occurs

- When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

Java Exception Keywords

- try
- catch
- finally
- throw
- throws



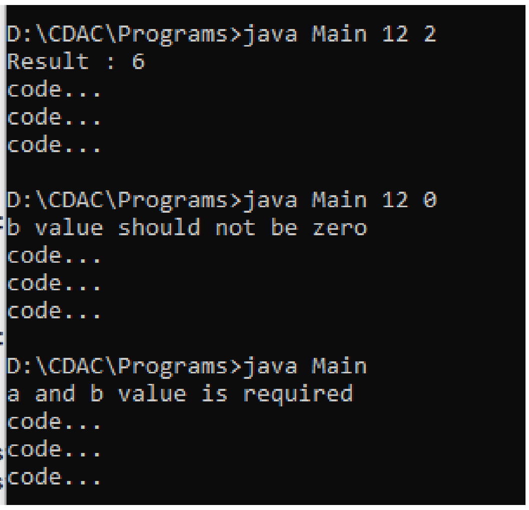
```
try
{
     // used to keep suspected instruction
}
catch(exceptionclassname objectname){
    // used to provide corrective action
}
```

Example#1

```
class Main{
        public static void main(String args[]){
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                try
                {
                        int c = a/b;
                        System.out.println("Result : "+c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("b value should not be zero");
                }
                System.out.println("code...");
                System.out.println("code...");
                System.out.println("code...");
        }
}
```

Example#2

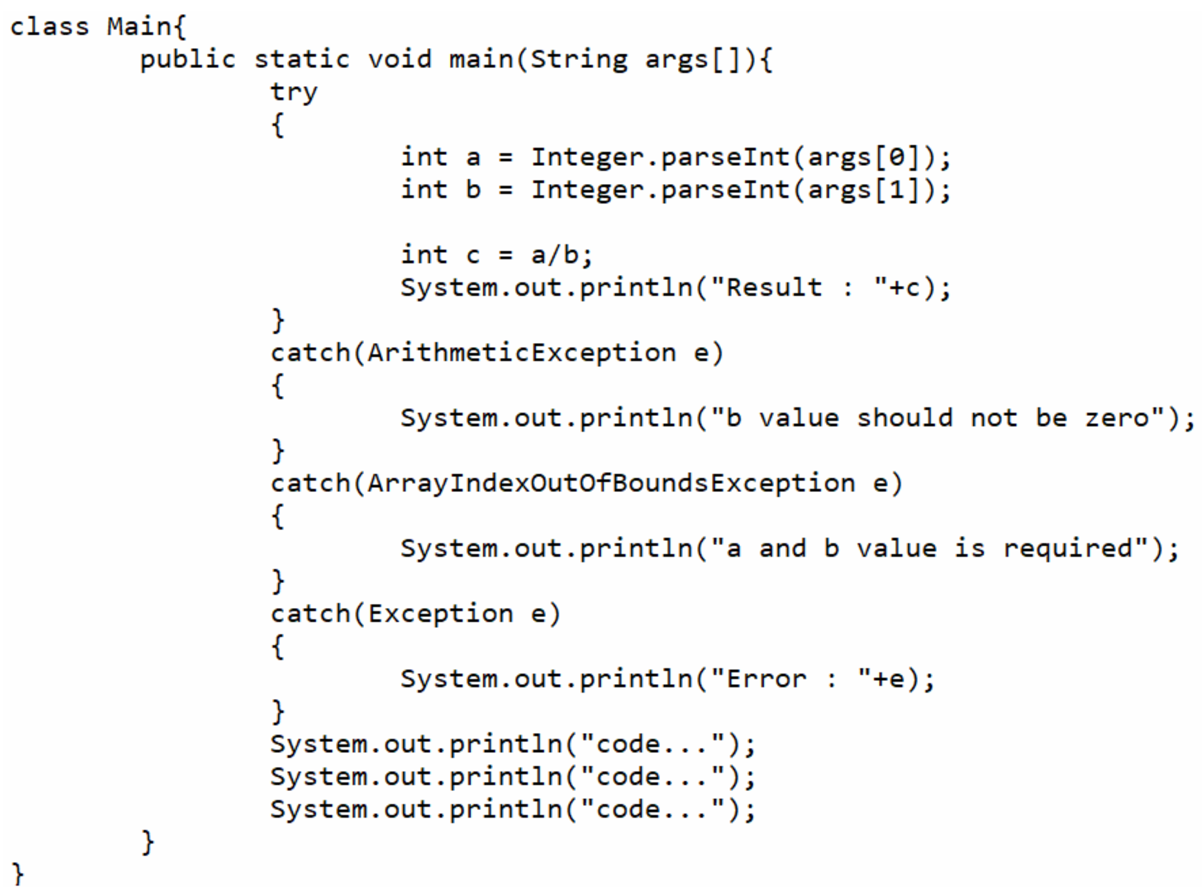
```
class Main{
        public static void main(String args[]){
                try
                {
                        int a = Integer.parseInt(args[0]);
                        int b = Integer.parseInt(args[1]);

                        int c = a/b;
                        System.out.println("Result : "+c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("b value should not be zero");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("a and b value is required");
                }
                System.out.println("code...");
                System.out.println("code...");
                System.out.println("code...");
        }
}
```



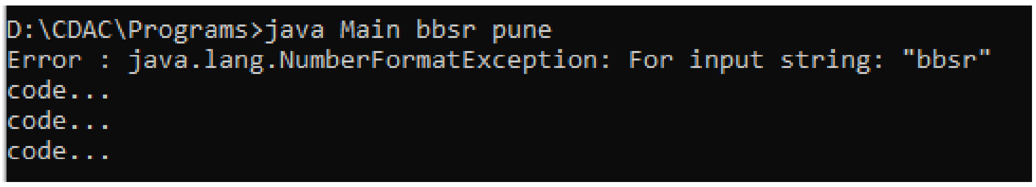
```
D:\CDAC\Programs>java Main 12 2
Result : 6
code...
code...
code...

D:\CDAC\Programs>java Main 12 0
b value should not be zero
code...
code...
code...

D:\CDAC\Programs>java Main
a and b value is required
code...
code...
code...
```

Generic Exception

Example#3



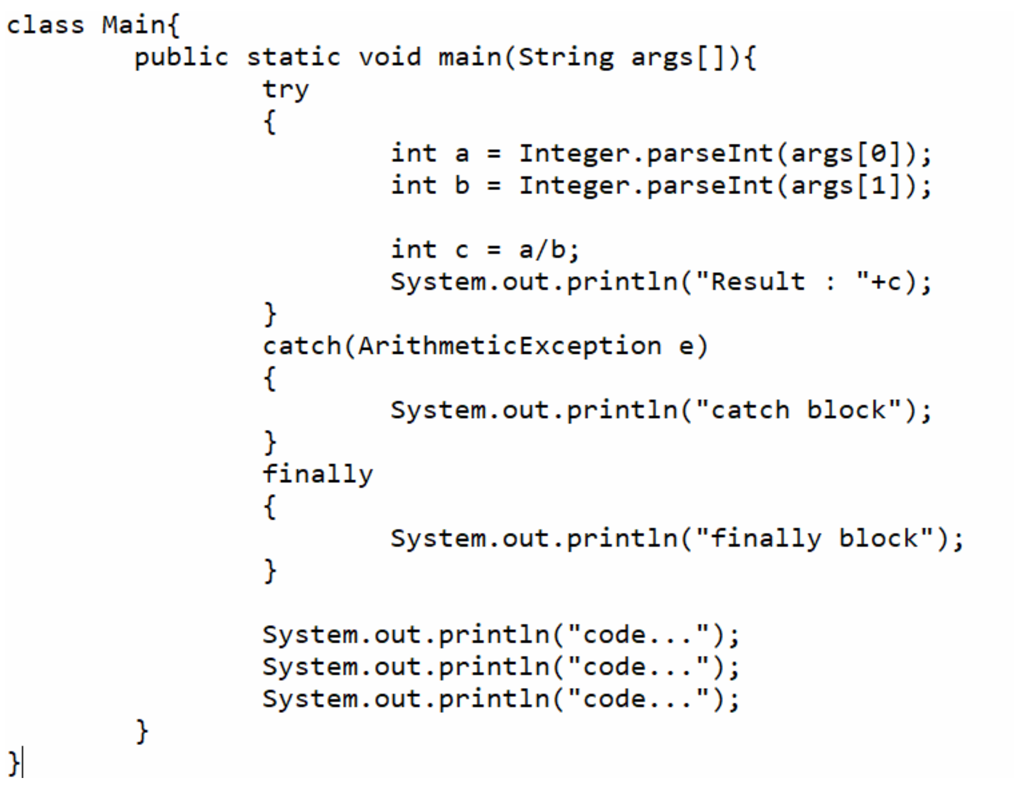
```
class Main{
        public static void main(String args[]){
                try
                {
                        int a = Integer.parseInt(args[0]);
                        int b = Integer.parseInt(args[1]);

                        int c = a/b;
                        System.out.println("Result : "+c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("b value should not be zero");
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println("a and b value is required");
                }
                catch(Exception e)
                {
                        System.out.println("Error : "+e);
                }
                System.out.println("code...");
                System.out.println("code...");
                System.out.println("code...");
        }
}
```



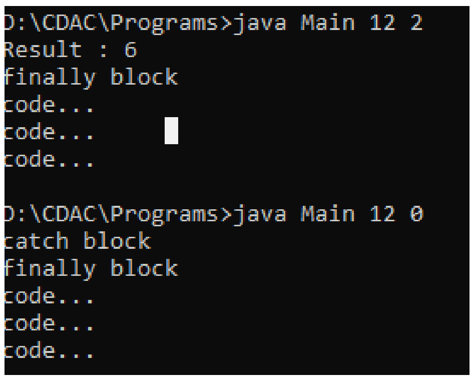
```
D:\CDAC\Programs>java Main bbsr pune
Error : java.lang.NumberFormatException: For input string: "bbsr"
code...
code...
code...
```

finally

- Finally block is always executed whether exception is raised or not.
- It is used to provide clean up code
    - Database connection closing
    - Object destruction
    - File Closing
    - Etc.



```
class Main{
	public static void main(String args[]){
		try
		{
			int a = Integer.parseInt(args[0]);
			int b = Integer.parseInt(args[1]);

			int c = a/b;
			System.out.println("Result : "+c);
		}
		catch(ArithmeticException e)
		{
			System.out.println("catch block");
		}
		finally
		{
			System.out.println("finally block");
		}

		System.out.println("code...");
		System.out.println("code...");
		System.out.println("code...");
	}
}
```



```
D:\CDAC\Programs>java Main 12 2
Result : 6
finally block
code...
code...
code...

D:\CDAC\Programs>java Main 12 0
catch block
finally block
code...
code...
code...
```

Note

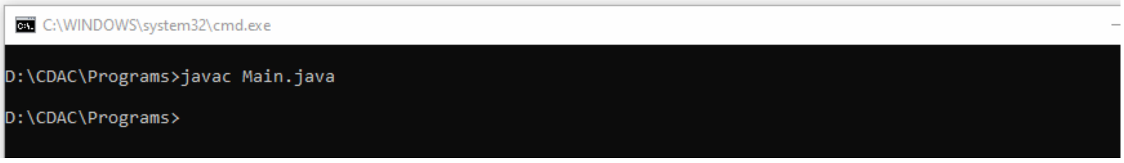
- In the exception handling must handle the exception in two ways
  - By using try catch blocks
  - By using throws keyword

Java throws keyword

- throws keyword is bypassing the exception from present method to caller method.
- If main method is throws the exception then JVM is responsible to handle the exception.

Example# (throws)

```
import java.io.*;

class Main{
        public static void main(String args[])throws FileNotFoundException{
                FileOutputStream fout = new FileOutputStream("d:\\data.txt");
        }
}
```



```
D:\CDAC\Programs>javac Main.java

D:\CDAC\Programs>
```

Java throw keyword

- Throw is used to throw predefined or user defined exception explicitly.

Difference between throw and throws

throw	throws
Throw is used to throw exception explicitly	Throws is used to bypass exception to the caller.
We are using throw keyword at method implementation level	We are using throws keyword at method declaration level

StackOverFlowError

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

OutOfMemoryError

```
class Main
{
	public static void main(String args[])
	{
		int arr[] = new int[1000000000];
	}
}
```

User defined exception

Exceptions created by user are called user defined Exceptions.

```
class MyException extends Exception
{
	MyException(String msg)
	{
		super(msg);
	}
}
class Main
{
	public static void main(String args[])
	{
		try
		{
			int age = Integer.parseInt(args[0]);
			if(age<18)
			{
				throw new MyException("You are not eligible for voting");
			}
			System.out.println("You are eligible for voting");
		}
		catch(Exception e)
		{
			System.out.println(e.getMessage());
		}
	}
}
```

Try with resource

```
import java.io.*;
class Main
{
	public static void main(String args[])
	{
		try(FileOutputStream fout =new FileOutputStream("data.txt"))		{
```

```
                    fout.write("BBSR1".getBytes());
                    System.out.println("Date Saved..");
            }
            catch(Exception e)
            {
                    System.out.println(e);
            }
        }
}
```

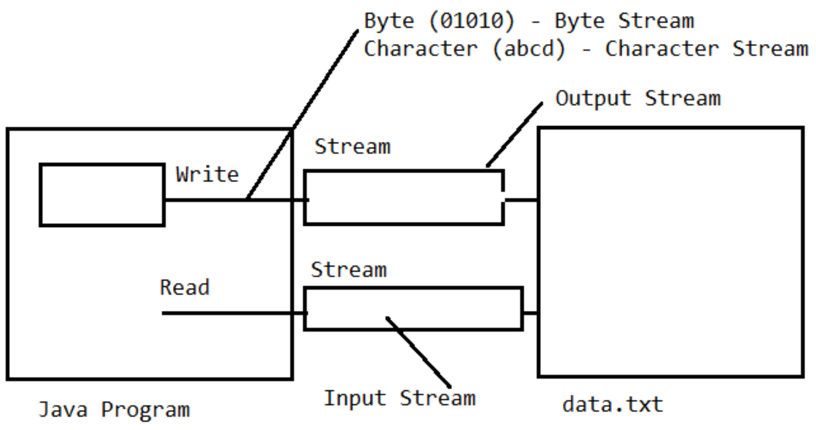
Java I/O Tutorial

IOStream

- The data or information which given into the program is called input.
- The data or information which given by the program is called output.
- Stream is a channel or medium which allows to send data from one place to the another place.
- All IOStream classes are aviable in “java.io” package.

Types of Stream

- Stream is divided into 2 types based on type of data passed through stream.
    - Byte Stream
    - Character Stream
- Stream is divided into 2 types based on data flow direction
    - Input Stream
    - Output Stream



Java FileOutputStream Class

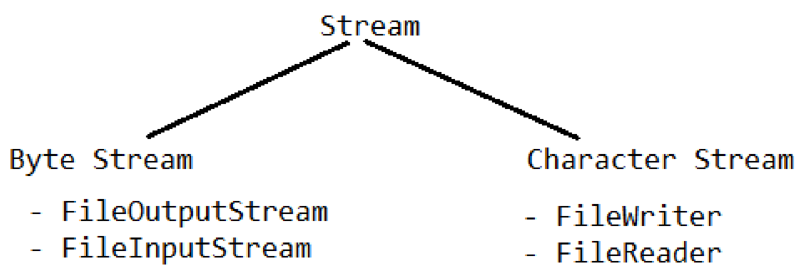
- It is used to write data into the destination file from java application. To write data into the .txt file we have to use write () method.

```java
import java.io.FileOutputStream;

public class WriteData1 {
    public static void main(String[] args)throws Exception {
        String data = "CDAC";
        FileOutputStream fout = new FileOutputStream("d:\\data.txt");
        fout.write(data.getBytes());
        fout.close();
        System.out.println("Data Saved Successfully...");
    }
}
```

Java FileInputStream Class

- It is used to read the data from the destination file to the java application. To read the data from the .txt file we have to use read () method.

```java
import java.io.FileInputStream;

public class ReadData1 {
    public static void main(String[] args)throws Exception {
        FileInputStream fin = new FileInputStream("d:\\data.txt");
        int c = fin.read();
        while(c!=-1) {
            System.out.print((char)c);
            c = fin.read();
        }
        fin.close();
    }
}
```



Java FileWriter Class

- FileWriter class is given for writing character files. Whether or not a file is available or may be created depends upon the underlying platform.

```java
import java.io.FileWriter;

public class WriteData2 {
    public static void main(String[] args) throws Exception {
        String data = "BBSR";
        FileWriter fw = new FileWriter("d:\\data.txt");
        fw.write(data);
        fw.close();
        System.out.println("Data saved ...");
    }
}
```

Java FileReader Class

- FileReader is a convenience class for reading character files.

```
import java.io.FileReader;

public class ReadData2 {
    public static void main(String[] args) throws Exception {
        FileReader fr = new FileReader("d:\\data.txt");
        int c = fr.read();
        while(c!=-1) {
            System.out.print((char)c);
            c = fr.read();
        }
        fr.close();
    }
}
```

Assignments#5

Write a java program to copy a File

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

public class CopyFile {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("d:\\f1\\data.txt");
        FileOutputStream fout = new FileOutputStream("d:\\f2\\data.txt");

        int c = fin.read();
        while(c!=-1) {
            fout.write(c);
            c = fin.read();
        }
        fin.close();
        fout.close();
        System.out.println("copied file....");
    }
}
```

Assignments#6

Write a java program to move a File.

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;

public class MoveFile {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("d:\\f1\\data.txt");
        FileOutputStream fout = new FileOutputStream("d:\\f2\\data.txt");

        int c = fin.read();
        while(c!=-1) {
            fout.write(c);
            c = fin.read();
        }
        fin.close();
        fout.close();
        File f = new File("d:\\f1\\data.txt");
        f.delete();
        System.out.println("moved file....");
    }
}
```

Serialization

- The process of writing the state of an object into a file is called serialization.

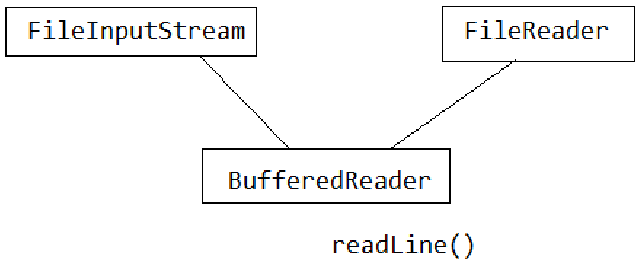
```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

class Student implements Serializable {
    int age;
    String name;
}
public class SerializationDemo {
    public static void main(String[] args) throws Exception {
        Student s = new Student();
        s.age = 101;
        s.name = "Raj";
        FileOutputStream fout = new FileOutputStream("d:\\data.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fout);
        oos.writeObject(s);
        oos.close();
        fout.close();
        System.out.println("Object saved ....");
    }
}
```

Deserialization

- The process of reading the state of an object from a file is called Deserialization.

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

public class DeserializationDemo {
    public static void main(String[] args) throws Exception {
        FileInputStream fin = new FileInputStream("d:\\data.ser");
        ObjectInputStream ois = new ObjectInputStream(fin);
        Student s = (Student)ois.readObject();
        System.out.println("Name : "+s.name);
        System.out.println("Age : "+s.age);
    }
}
```

Java BufferedReader Class



- BufferedReader class is used to read the text from a character based input stream.
- It can be used to read data line by line by readLine () method. It makes the performance fast. It inherits Reader class.

```
import java.io.*;

class BufferedReaderDemo
{
        public static void main(String args[])throws Exception
        {
                FileReader fr=new FileReader("D:\\demo.txt");
                BufferedReader br=new BufferedReader(fr);
                String data=br.readLine();
                while(data != null)
                {
                        System.out.print(data);
                        data=br.readLine();
                }
                br.close();
                fr.close();
        }
}
```

Java InputStreamReader

- An input stream reader is a bridge from byte stream to character stream. It reads byte and decodes them into characters.

```
import java.io.*;

class BufferedReaderDemo
{
        public static void main(String args[])throws Exception
        {
                FileInputStream fis = new FileInputStream("d:\\demo.txt");
                BufferedReader br=new BufferedReader(new InputStreamReader(fis));

                String data;
                while((data=br.readLine())!=null)
                {
                System.out.print(data);
                }
                br.close();
                fis.close();
        }
}
```

Reading Data from the Keyboard

- Using Command line argument
- Using Scanner Class
- Using Buffered Reader
- Using Console

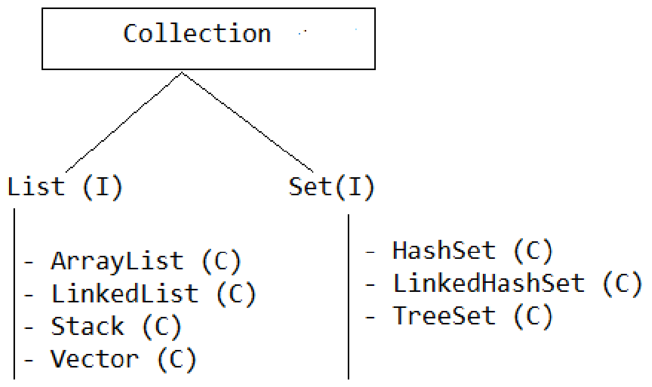
Reading Data from the Keyboard using BufferedReader

```
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class ReadData5 {
    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your name : ");
        String name = br.readLine();
        System.out.print("Enter your age : ");
        int age =  Integer.parseInt(br.readLine());
        System.out.print("Enter your email id : ");
        String email = br.readLine();

        System.out.println("Name = "+name);
        System.out.println("Age = "+age);
        System.out.println("Email = "+email);
    }
}
```

Collections in Java

- Collection framework provides set of classes and interfaces that are used to represent group of objects as a single entity.

Hierarchy of Collection Framework



List Interface

- List interface has following proprieties
  - Allows null insertion.
  - Allows duplicate objects.
  - Preserved insertion order

ArrayList

- ArrayList stores Heterogeneous objects.
- ArrayList allowed null insertion.
- ArrayList preserved Insertion order.
- Duplicate objects are allowed.
- The under laying data structure is growable array.
- Every method present in the ArrayList is not synchronized

Example

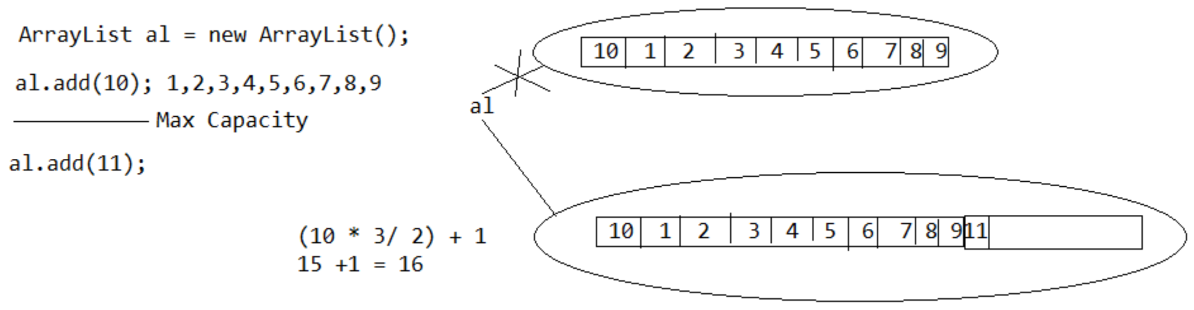
```java
import java.util.ArrayList;

public class ArrayListDemo {
	public static void main(String[] args) {
		ArrayList al = new ArrayList();
		al.add(10);
		al.add(20);
		al.add("bbsr");
		al.add(null);
		al.add(20);
		System.out.println(al);
```

```
        }
}
```

Note

- The default capacity of the ArrayList is **10** once it reaches its maximum capacity then size is automatically increased by:

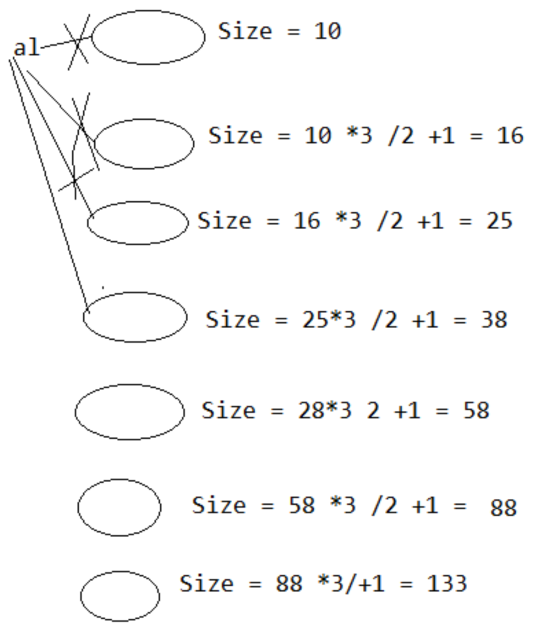
New capacity = ((old capacity*3)/2)+1



```
ArrayList al = new ArrayList();
al.add(10); 1,2,3,4,5,6,7,8,9
———————— Max Capacity
al.add(11);

(10 * 3/ 2) + 1
15 +1 = 16
```

Example

```
import java.util.ArrayList;

public class Main
{
      public static void main(String[] args)
      {
            ArrayList al = new ArrayList();
            for(int i=1;i<=100;i++)
            {
                  al.add(i);
            }
            System.out.println(al);
      }
}
```



```
import java.util.ArrayList;

public class Main
{
        public static void main(String[] args)
        {
                ArrayList al = new ArrayList(100);
                for(int i=1;i<=100;i++)
```

```
                    {
                              al.add(i);
                    }
                    System.out.println(al);
          }
}
```

Example

```
import java.util.ArrayList;

public class ArrayListDemo {
          public static void main(String[] args) {
                    ArrayList al = new ArrayList();
                    al.add("A");
                    al.add("B");
                    System.out.println(al); //[A, B]
                    al.add(1,"C");
                    System.out.println(al);// [A, C, B]
                    al.remove("B");
                    System.out.println(al); // [A, C]
                    al.remove(0);
                    System.out.println(al);// [C]
          }
}
```

Generic version of ArrayList

```
import java.util.ArrayList;

public class ArrayListDemo {
          public static void main(String[] args) {
                    ArrayList<String> al = new ArrayList<String>();
                    al.add("Ram");
                    al.add("Raj");
                    System.out.println(al);
          }
}
```

LinkedList

- Heterogeneous objects are allowed.
- Null insertion is possible.
- Insertion order is preserved
- Duplicate objects are allowed.
- The under laying data structure is double linked list.
- Every method present in the ArrayList is not synchronized

Example

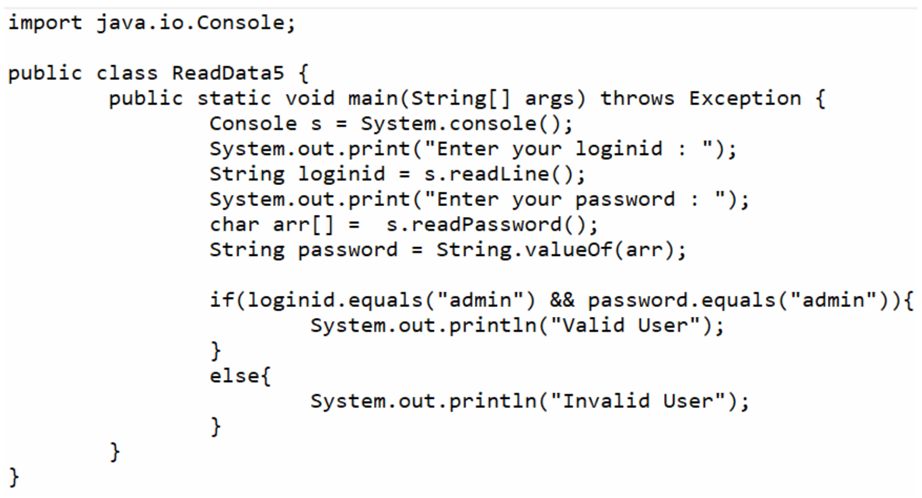
```
import java.util.LinkedList;

public class LinkedListDemo {
	public static void main(String[] args) {
		LinkedList list = new LinkedList();
		list.add(10);
		list.add(30);
		list.add("bbsr");
		list.add(null);
		list.add(30);
		System.out.println(list);
	}
}
```

Note

- In ArrayList, if we are adding/removing object at the middle of ArrayList then number of shift operations are requires. Hence ArrayList is not suitable for insertion and deletion operation.
- LinkedList is recommended to perform insertion and deletion operation.
- ArrayList is suitable for fetch operation but LinkedList is not suitable.

Reading Data from the Keyboard using Console

- If an application needs to read a password or other secure data then we should use readPassword () method.



```
import java.io.Console;

public class ReadData5 {
	public static void main(String[] args) throws Exception {
		Console s = System.console();
		System.out.print("Enter your loginid : ");
		String loginid = s.readLine();
		System.out.print("Enter your password : ");
		char arr[] =  s.readPassword();
		String password = String.valueOf(arr);

		if(loginid.equals("admin") && password.equals("admin")){
			System.out.println("Valid User");
		}
		else{
			System.out.println("Invalid User");
		}
	}
}
```

Vector

- Heterogeneous objects are allowed
- Null insertion is possible
- Duplicate objects are allowed
- Insertion order is preserved
- The under laying data structure is growable array
- Every method present in the Vector is synchronized

Example

```
import java.util.Vector;

public class VectorDemo {
	public static void main(String[] args) {
		Vector v = new Vector();
		v.add(10);
		v.add(20);
		v.add("BBSR");
		v.add(null);
		v.add("BBSR");
		System.out.println(v);
	}
}
```

Note

- The default capacity of the vector is 10 once it reaches its maximum capacity then size is automatically increased by:

New capacity = current capacity*2

Example

```
import java.util.Vector;

public class VectorDemo {
	public static void main(String[] args) {
		Vector v = new Vector();
		for(int i=1;i<=10;i++)
		{
			v.add(i);
		}
		System.out.println("Capacity = "+v.capacity());
		System.out.println("Size = "+v.size());
		v.add(11);
```

```
                System.out.println("Capacity = "+v.capacity());
                System.out.println("Size = "+v.size());
        }
}
```

Stack

- It is a child class of vector.
- It is designed for LIFO (last in fist order)

Example

```
import java.util.Stack;

public class StackDemo {
        public static void main(String[] args) {
                Stack s = new Stack();
                s.push(10);
                s.push(20);
                s.push(30);
                System.out.println(s);
                s.pop();
                System.out.println(s);
        }
}
```

HashSet

- HashSet stores Heterogeneous objects.
- HashSet allowed null insertion.
- HashSet not preserved Insertion order.
- Duplicate objects are not allowed.
- The under laying data structure is hash table.
- Every method present in the HashSet is not synchronized.

Example

```
import java.util.HashSet;

public class HashSetDemo {
        public static void main(String[] args) {
                HashSet hs = new HashSet();
                hs.add(10);
                hs.add(20);
                hs.add("bbsr");
                hs.add(null);
                hs.add("bbsr");
                System.out.println(hs);
        }
}
```

LinkedHashSet

- HashSet stores Heterogeneous objects.
- HashSet allowed null insertion.
- LinkedHashSet preserved Insertion order.
- Duplicate objects are not allowed.
- The under laying data structure is hash table and linked list.
- Every method present in the HashSet is not synchronized.

Example

```
import java.util.LinkedHashSet;

public class LinkedHashSetDemo {
        public static void main(String[] args) {
                LinkedHashSet hs = new LinkedHashSet();
                hs.add(10);
                hs.add(20);
                hs.add("bbsr");
                hs.add(null);
                hs.add("bbsr");

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

TreeSet

- TreeSet does not allowed heterogeneous object.
- TreeSet class doesn't allow null element.
- TreeSet class does not allowed duplicate value.
- Every methods present in TreeSet are not synchronized.
- TreeSet class maintains ascending order.

Example

```
import java.util.TreeSet;

public class TreeSetDemo {
	public static void main(String[] args) {
		TreeSet ts = new TreeSet();
		ts.add(10);
		ts.add(30);
		ts.add(20);
		ts.add(30);
		System.out.println(ts);
	}
}
```

HashMap

- It used to hold key value pairs.
- Duplicate keys are not allowed but values can be duplicated.
- Insertion order is not preserved.
- Null is allowed for key (only once)and allows for values any number of times.
- Underlying data Structure is Hashtable.
- Every method is non-synchronized.

Example

```
import java.util.*;

class HashMapDemo
{
	public static void main(String[] args)
	{
		HashMap h=new HashMap();
		h.put("Android",7000);
		h.put("Core Java",5000);
		h.put("PHP",7000);
		System.out.println(h);

	}
}
```

LinkedHashMap

- Underlying data Structure is Hash Table & LinkedList.
- Duplicate keys are not allowed but values can be duplicated.
- Insertion order is preserved.

Example

```
import java.util.*;

class LinkedHashMapDemo
{
        public static void main(String[] args)
        {
                LinkedHashMap h=new LinkedHashMap();
                h.put("Android",7000);
                h.put("Core Java",5000);
                h.put("PHP",7000);
                System.out.println(h);

        }
}
```

Cursor

- Cursor is used to retrieve objects one by one from the collection.
- There are three types of cursors are available in java
    - Enumeration
    - Iterator
    - ListIterator

Enumeration

- We can use Enumeration to get objects one by one from the old collection objects(e.g. vector, stack).

Example

```
import java.util.*;

public class Main
{
        public static void main(String[] args)
        {
                Vector v = new Vector();
                for(int i=0;i<=10;i++)
                {
                        v.add(i);
                }
                System.out.println(v);
                Enumeration e = v.elements();
                while(e.hasMoreElements())
                {
                        Integer i = (Integer)e.nextElement();
                        System.out.println(i);
                }
        }
}
```

Iterator

- We can apply Iterator concept for any Collection object hence it is universal cursor.
- By using Iterator we can perform both read and remove operations.

Example

```
import java.util.*;
public class Main
{
	public static void main(String[] args)
	{
		ArrayList<Integer> al = new ArrayList<Integer>();
		for(int i=0;i<10;i++)
		{
			al.add(i);
		}
		System.out.println(al);
		Iterator itr = al.iterator();
		while(itr.hasNext())
		{
			Integer i = (Integer)itr.next();
			System.out.println(i);
		}
	}
}
```

Example

```
import java.util.*;
public class Main
{
	public static void main(String[] args)
	{
		ArrayList<Integer> al = new ArrayList<Integer>();
		for(int i=0;i<10;i++)
		{
			al.add(i);
		}
		System.out.println(al);
		Iterator itr = al.iterator();
		while(itr.hasNext())
		{
			Integer i = (Integer)itr.next();
			if(i%2==0)
				itr.remove();
		}
		System.out.println(al);
```

```
          }
}
```

ListIterator

- By using ListIterator we can move either to the forward direction or to the backward direction.
- By using ListIterator we can perform addition, replacement ,read and remove operations.

Method of List Iterator

- Forward direction
    - public boolean hasNext()
    - public void next()
    - public int nextIndex()
- Backward direction
    - public boolean hasPrevious()
    - public void previous()
    - public int previousIndex()
- Other capability method
    - public void remove()
    - public void set(Object new)
    - public void add(Object new)

Example

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

public class ListIteratorDemo {
          public static void main(String[] args) {
                    ArrayList<String> al = new ArrayList<String>();
                    al.add("bbsr");
                    al.add("pune");
```

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

                ListIterator li = al.listIterator();
                System.out.println("Forward Direction : ");
                while(li.hasNext()) {
                        String s = (String)li.next();
                        System.out.println(s);
                }
                System.out.println("Backward Direction : ");
                while(li.hasPrevious()) {
                        String s = (String)li.previous();
                        System.out.println(s);
                }
        }
}
```

Property	Enumeration	Iterator	ListIterator
Applicable for	Only legacy classes	Any Collection classes	Only list classes
Movement	Only forward direction	Only Forward direction	Both forward and backward direction
Accessibility	Only read access	Both read and remove	Read, remove, replace and addition of new objects
How to get object	By using elements() method of vector class.	By using iterator() method of Collection interface.	By using listIterator() method of List interface.

Java Member Inner class

- A non-static class that is created inside a class but outside a method is called member inner class. It is also known as a regular inner class.

Example

```
class Outer
{
        class Inner{
                void show(){
                        System.out.println("show()...");
                }
        }
        public static void main(String args[]){
                Outer obj=new Outer();
                Outer.Inner in=obj.new Inner();
                in.show();
        }
}
```

Java Local inner class

- A class created inside a method is called local inner class.

Example

```
public class LocalInner{
        void display(){
                class Local{
                        void display1(){
                                System.out.println("display");
                        }
                }
                Local l=new Local();
                l.display1();
        }
        public static void main(String args[]){
                LocalInner obj=new LocalInner();
                obj.display();
        }
}
```

Java static nested class

- A static class is a class that is created inside a class, is called a static nested class.

Example

```
class Outer{
        static class Inner{
                void display(){
                        System.out.println("dispaly");
                }
        }
        public static void main(String args[]){
                Outer.Inner obj=new Outer.Inner();
                obj.display();
        }
}
```

Java Anonymous inner class

- Anonymous inner class is an inner class without a name.

```
class Test{
        void display(){
          #3| System.out.println("display method started...");
            | for(int i=1;i<=10;i++){
          #4|         System.out.println("display() : "+i);
            | }
        }
}
class Main{
        public static void main(String args[]){
          #1| System.out.println("Main method started...");
            |
          #2| Test ob = new Test();
            | ob.display();

          #5| for(int i=1;i<=10;i++){
            |         System.out.println("main() : "+i);
            | }

        }
}
```

- Main class
- main method
- logic [main thread]

```
o/p
----
Main method started
display method started
display()...1...10
main() 1....10
```

Sequential Execution

Multithreading in Java

- Executing different parts of the program simultaneously is called multithreading.
- Thread is the separate path of sequential execution.
- The main advantage of multithreading is to improve the CPU utilization. Hence execution speed will be increased and response time will be decreased.

Advantages of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Main Thread

- When a java program started one thread is running immediately that thread is called main thread.

```
class Main {
        public static void main(String[] args)
        {
                Thread t = Thread.currentThread();
                System.out.println("Thread Name : "+t.getName());
                t.setName("My Thread..");
                System.out.println("Thread Name : "+t.getName());
        }
}
```

Different ways to create thread

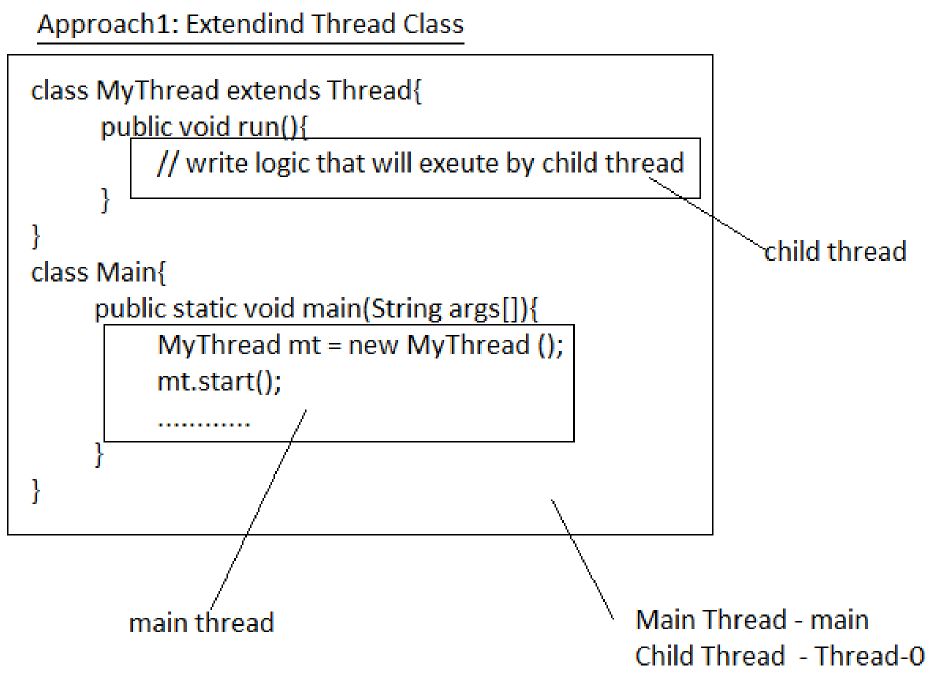
- By extending Thread class
- By implementing Runnable interface

Internal of Thread Class and Runnable Interface

```
interface Runnable{
        void run();
}
class Thread implements Runnable{
        public void run(){
                // empty body
        }
        public void start(){
                1. Threading Creation
                2. Call run()
        }
}
```

By extending Thread class

- Create a class by extending Thread class.
- Override the run() method to write the logic of the thread
- Create user defined thread class object
- Start thread by calling start() method of Thread class


Approach1: Extendind Thread Class
class MyThread extends Thread{
public void run(){
// write logic that will exeute by child thread
}
}
class Main{
public static void main(String args[]){
MyThread mt = new MyThread ();
mt.start();
....
}
}
child thread
main thread
Main Thread - main
Child Thread - Thread-0

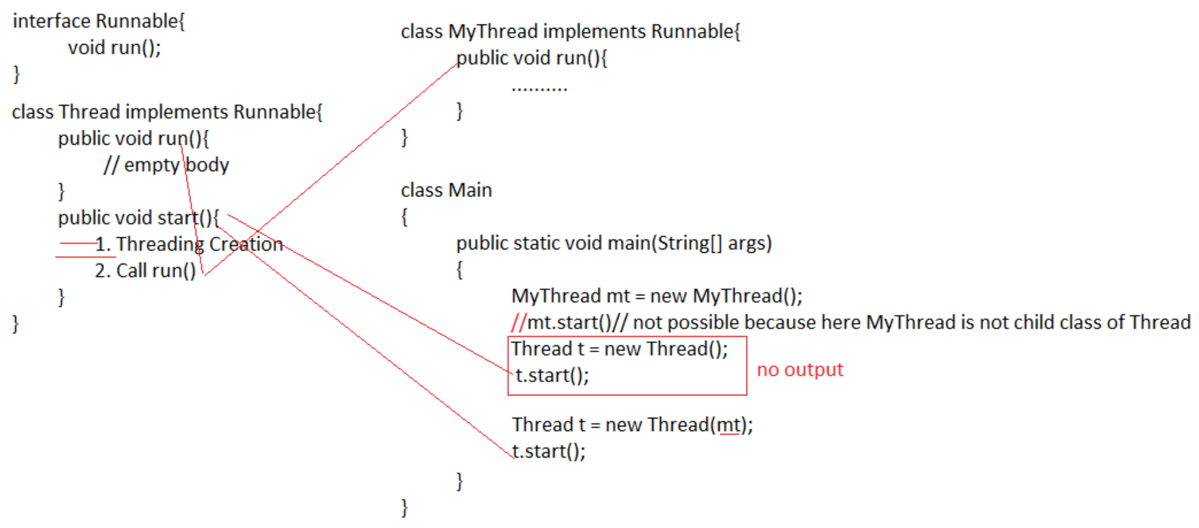
Example

```
class MyThread extends Thread{
        public void run(){
                Thread t = Thread.currentThread();
                System.out.println("Thread Name : "+t.getName());

                for(int i=1;i<=10;i++){
                        System.out.println("Child Thread : "+i);
                }
        }
}

class Main {
        public static void main(String[] args)
        {
                Thread t = Thread.currentThread();
                System.out.println("Thread Name : "+t.getName());

                MyThread mt = new MyThread();
                mt.start();

                for(int i=1;i<=10;i++){
                        System.out.println("Main Thread : "+i);
                }

        }
}
```

By implementing Runnable Interface

- Create a class by implementing Runnable interface.

- Override the run() method to write the logic of the thread
- Create user defined thread class object
- Creates a Thread class object
- Start thread by calling start() method of Thread class



```
interface Runnable{
        void run();
}
class Thread implements Runnable{
        public void run(){
                // empty body
        }
        public void start(){
                1. Threading Creation
                2. Call run()
        }
}
```

```
class MyThread implements Runnable{
        public void run(){
                ..........
        }
}

class Main
{
        public static void main(String[] args)
        {
                MyThread mt = new MyThread();
                //mt.start()// not possible because here MyThread is not child class of Thread
                Thread t = new Thread();
                t.start();                no output

                Thread t = new Thread(mt);
                t.start();
        }
}
```

Example

```
class MyThread implements Runnable{
        public void run(){

                for(int i=1;i<=10;i++){
                        System.out.println("Child Thread : "+i);
                }
        }
}

class Main {
        public static void main(String[] args)
        {
                MyThread mt = new MyThread();
                Thread t = new Thread(mt);
                t.start();

                for(int i=1;i<=10;i++){
                        System.out.println("Main Thread : "+i);
                }
        }
}
```

Priority of a Thread (Thread Priority)

- The valid range of thread priorities is 1 – 10. Where one is lowest priority and 10 is highest priority.
- The default priority of main thread is 5. The priority of child thread is inherited from the parent.

- Three constant values for the thread priority.
  MIN_PRIORITY = 1
  NORM_PRIORITY = 5
  MAX_PRIORITY = 10

Example

```
class MyThread extends Thread
{

        public void run()

        {

                Thread t = Thread.currentThread();
                System.out.println("Child Thread : "+t.getPriority());
        }

}
class Main{

        public static void main(String arg[])

        {
                Thread t = Thread.currentThread();
                t.setPriority(10);
                System.out.println("Main Thread : "+t.getPriority());

                MyThread mt = new MyThread();
                mt.setPriority(5);

                mt.start();


        }

}
```

Preventing Thread from execution

- We can prevent a thread from execution by using the following methods.
    - sleep()
    - join()

sleep()

- If a method has to wait some predefined amount of time with out execution then we should go for sleep() method.

Example

```
class MyThread extends Thread{
	public void run(){
		for(int i=1;i<=50;i++){
			System.out.println("Child : "+i);
		}
		System.out.println("End of child thread");
	}
}
class Main{
	public static void main(String args[]){
		MyThread mt = new MyThread();
		mt.start();

		for(int i=1;i<=5;i++){
			System.out.println("Main : "+i);
		}

		try{
			Thread.sleep(5000);
		}catch(Exception e){
			System.out.println("Error : "+e);
		}

		System.out.println("End of main thread");
	}
}
```

join()

- If a thread wants to wait until some other thread completion then we should go for join method.

Example

```
class MyThread extends Thread{
	public void run(){
		for(int i=1;i<=50;i++){
```

```
                System.out.println("Child : "+i);
            }
            System.out.println("End of child thread");
        }
}
class Main{
        public static void main(String args[]){
            MyThread mt = new MyThread();
            mt.start();

            for(int i=1;i<=5;i++){
                System.out.println("Main : "+i);
            }

            try{
                mt.join();
            }catch(Exception e){
                System.out.println("Error : "+e);
            }

            System.out.println("End of main thread");
        }
}
```

Life cycle of a Thread (Thread States)

1. New
2. Ready
3. Running state
4. Blocked / waiting
5. Dead state

New

```
MyThread mt = new MyThread();
```

Ready

```
mt.start();
```

Running State

- If thread scheduler allocates CPU for particular thread.

- Thread goes to running state.
- The Thread is running state means the run() is executed.

Blocked State

- If the running thread got interrupted of goes to sleeping state at that moment it goes to the blocked state.
-

Dead State

- If the business logic of the project is completed means run() over thread goes dead state.
-

Concurrent execution

- When more than one thread need access to one resource at a time that is called concurrent execution, which may creates a conflict.

Example

```
class Account
{
        float balance = 20000;
        void PassChq(String ThreadName, float amount)
        {
                if( amount < balance)
                {
                        System.out.println("Processing " + ThreadName);
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch(Exception e)
                        {
                                System.out.println("Error : " + e);
                        }
                        balance = balance - amount;
                        System.out.println(ThreadName + " is passed");
                }
                else
                {
                        System.out.println(ThreadName + " is not passed");
                }
        }
}
class Process extends Thread
{
        String ThreadName;
        float amount;
        Account obj;

        Process(Account obj, String ThreadName, float amount)
        {
                this.obj = obj;
                this.ThreadName = ThreadName;
                this.amount = amount;
        }
        public void run()
        {
                obj.PassChq(ThreadName, amount);
        }
}
class Bank
{
        public static void main(String ar[])
        {
                Account obj = new Account();
                Process chq1 = new Process(obj, "cheque1", 15000);
                Process chq2 = new Process(obj, "cheque2", 10000);
        chq1.start();
                chq2.start();
        }
}
```

Note

- So the threads need some way to ensure that the resource will be used by only one thread at a time.
- The process by which this is achieved is called synchronization.

Synchronization in Java

- ***Synchronized*** keyword is used to implement synchronization.
- If a method or block declared as synchronized then at a time only one thread is allowed to execute that method or block on the given object.
- Synchronized is the modifier applicable only for methods and blocks and we can't apply for classes and variables.

Example

```
class Account

{

        float balance = 20000;

        synchronized void PassChq(String ThreadName, float amount)
        {
                if( amount < balance)
                {
                        System.out.println("Processing " + ThreadName);
                        try
                        {
                                Thread.sleep(2000);
                        }
                        catch(Exception e)
                        {
                                System.out.println("Error : " + e);
                        }
                        balance = balance - amount;
                        System.out.println(ThreadName + " is passed");
                }
                else
                {
                        System.out.println(ThreadName + " is not passed");
                }
        }
}
class Process extends Thread
{
        String ThreadName;
        float amount;
        Account obj;

        Process(Account obj, String ThreadName, float amount)
        {
                this.obj = obj;
                this.ThreadName = ThreadName;
                this.amount = amount;
        }
        public void run()
        {
                obj.PassChq(ThreadName, amount);
        }
}
class Bank
{
        public static void main(String ar[])
        {
                Account obj = new Account();
                Process chq1 = new Process(obj, "cheque1", 15000);
                Process chq2 = new Process(obj, "cheque2", 10000);
        chq1.start();
                chq2.start();
        }
}
```

Inter-thread Communication in Java

- Following methods are used to implement IPC

wait() method

  - Tell the calling thread to give up the lock and go to sleep until some other thread enters the same lock and calls notify().

notify() method

  - Wakes up a thread that called wait().

notifyAll() method

  - Wakes up all the threads that called wait() and one of the threads will be granted access.

Example

```
class Customer
{
          int amount=10000;

          synchronized void withdraw(int amount)
          {
                    System.out.println("going to withdraw...");
                    if(this.amount<amount)
                    {
                              System.out.println("Less balance; waiting for deposit...");
                              try
                              {
                                        wait();
                              }
                              catch(Exception e)
                              {
                              }
                    }
                    this.amount = this.amount - amount;
                    System.out.println("withdraw completed...");
          }
          synchronized void deposit(int amount)
          {
                    System.out.println("going to deposit...");
                    this.amount+=amount;
                    System.out.println("deposit completed... ");
                    notify();
          }
}
class MyThread1 extends Thread
{
          Customer c;
          MyThread1(Customer c)
          {
                    this.c = c;
          }
          public void run()
          {
                    c.withdraw(15000);
          }
}
class MyThread2 extends Thread
{
          Customer c;
          MyThread2(Customer c)
          {
                    this.c = c;
```

```
	}
	public void run()
	{
		c.deposit(10000);
	}
}
class Test
{
	public static void main(String args[])
	{
		Customer c=new Customer();
		MyThread1 t1 = new MyThread1(c);
		MyThread2 t2 = new MyThread2(c);
		t1.start();
		t2.start();
	}
}
```

Assign same task to multiple threads

```
class MyThread extends Thread{
	public void run(){
		System.out.println("task1...");
	}
}
class Main{
	public static void main(String args[]){
		MyThread t1=new MyThread();
		MyThread t2=new MyThread();
		MyThread t3=new MyThread();

		t1.start();
		t2.start();
		t3.start();
	}
}
```

Assign different task to multiple threads

```
class MyThread1 extends Thread{
	public void run(){
		System.out.println("task1...");
	}
}
class MyThread2 extends Thread{
        public void run(){
                System.out.println("task2...");
        }
}

class Main{
        public static void main(String args[]){
                MyThread1  t1=new MyThread1();
                MyThread2  t2=new MyThread2();

                t1.start();
                t2.start();
        }
}
```

Deadlock in Java

- When a thread holds a resource and waits for another resource to be released by second thread, the second thread holding a resource and waiting for a resource to be released by first thread, then in such case both the thread will be waiting and they never execute. This is called deadlock.

```
class Test{
        public synchronized void show1(Best b){
                System.out.println("Thraed1 start execution of show1()..");
                try{
                        Thread.sleep(6000);
                }
                catch(Exception e){
                }
                System.out.println("Thraed1 is trying to call display method of Best class");
                b.display();
        }
        public synchronized void display(){
                System.out.println("display() method of Best class");
        }
}
class Best{
        public synchronized void show2(Test t){
                System.out.println("Thraed2 start execution of show2()..");
                try{
                        Thread.sleep(6000);
                }
                catch(Exception e){
                }
                System.out.println("Thraed2 is trying to call display method of Test class");
                t.display();
        }
```

```
	public synchronized void display(){
		System.out.println("display() method of Test class");
	}
}
class Deadlock extends Thread{
	Test t = new Test();
	Best b = new Best();
	public void m1(){
		this.start();
		t.show1(b);
	}
	public void run(){
		b.show2(t);
	}
	public static void main(String args[]){
		Deadlock d = new Deadlock();
		d.m1();
	}
}
```

ThreadGroup

```
class MyThread implements Runnable{
	public void run() {
		System.out.println(Thread.currentThread().getName());
	}
}
class ThreadGroupDemo{
	public static void main(String[] args) {
		MyThread runnable = new MyThread();
		ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

		Thread t1 = new Thread(tg1, runnable,"one");
		t1.start();
		Thread t2 = new Thread(tg1, runnable,"two");
		t2.start();
		Thread t3 = new Thread(tg1, runnable,"three");
		t3.start();

		System.out.println("Thread Group Name: "+tg1.getName());
		tg1.list();
  }
}
```

Calling Private Method Using Refelection API

```
import java.lang.reflect.Method;

class Student{
	private int getAge(){
		return 10;
	}
}
class Main{
	public static void main(String args[])throws Exception{
		Student t = new Student();
		Method privateMethod = Student.class.getDeclaredMethod("getAge");
		privateMethod.setAccessible(true);
		int age = (int)privateMethod.invoke(t);
		System.out.println("Age of Student: " + age);
	}
}
```

Difference between new and newInstance()

new Operator

```
class Student{
	Student(){
		System.out.println("constructor...");
	}
}
class Main{
	public static void main(String args[]){
		Student s = new Student();
	}
}
```

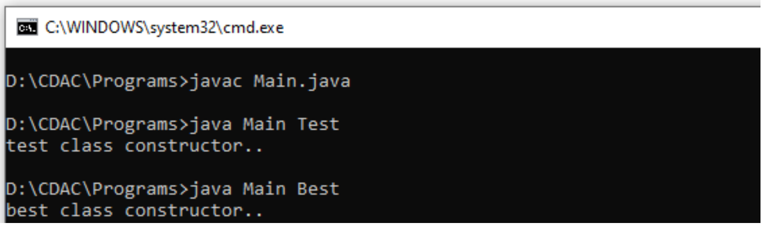
newInstance()

```
class Student{
	Student(){
		System.out.println("constructor...");
	}
}
class Main{
	public static void main(String args[])throws Exception{
		Class c = Class.forName("Student");
		c.newInstance();
	}
}
```

Note

If class name is given at runtime then we have to use newInstance() method to crate object.

```
class Test{
        Test(){
                System.out.println("test class constructor..");
        }
}
class Best{
        Best(){
                System.out.println("best class constructor..");
        }
}
class Main{
        public static void main(String args[])throws Exception{
                Class.forName(args[0]).newInstance();
        }
}
```



C:\WINDOWS\system32\cmd.exe

```
D:\CDAC\Programs>javac Main.java

D:\CDAC\Programs>java Main Test
test class constructor..

D:\CDAC\Programs>java Main Best
best class constructor..
```

Java Date and Time

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

public class Test {
        public static void main(String[] args) {
                Date dt = new Date();
                System.out.println(dt);

                Calendar c = Calendar.getInstance();
                System.out.println(c.get(Calendar.DATE)+"-"+(c.get(Calendar.MONTH)+1)+"-
"+c.get(Calendar.YEAR));

        System.out.println(c.get(Calendar.HOUR)+":"+c.get(Calendar.MINUTE)+":"+c.get(Cal
endar.SECOND));
        }
}
```

Java SimpleDateFormat

```
import java.text.SimpleDateFormat;
import java.util.Date;

public class Test {
        public static void main(String[] args) {
                Date dt = new Date();

                SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/YYYY");
                String dt2 = sdf.format(dt);

                SimpleDateFormat sdf2 = new SimpleDateFormat("dd-MMM-YYYY
hh:mm:ss");
                String dt3 = sdf2.format(dt);
                System.out.println(dt2);
                System.out.println(dt3);
        }
}
```