

# **MY JAVA NOTES**

FROM JSPIDER, HEBBAL

**Prepared By :**

***SAGAR SUNAR***

**Contact No. : +91-7483666261**

+977-9864453622(Whatsapp)

**E-mail :- [sagarsunar202@gmail.com](mailto:sagarsunar202@gmail.com)**

**Address : Thirumenahalli, Yelanhanka, Bangalore-64.**



*These notes are completely based on lectures given by respected sir, Mr. **Subham K.S.** , whose teachings always made new hope in learning programming concepts. Hence, it is only for my personal educational purpose and should not be used for any commercial printing or selling purpose.*

*-Thank you!  
SAGAR*



## **MODULE-3(Libraries)**

- Object Class and its methods
- String Class and its methods
- Exception Handling\*\*
- Collection Framework Library\*\*
  - Multithreading\*\*
  - File Handling



# Exception Handling

An exception is a unexpected or unwanted event which leads to abnormal termination of the program.

**or**

An exception is an unexpected event triggered by JVM which will force JVM to abnormally terminate the program.

**Error :** Errors are the expected mistakes done by the programmer which might occur during compile time or run time.

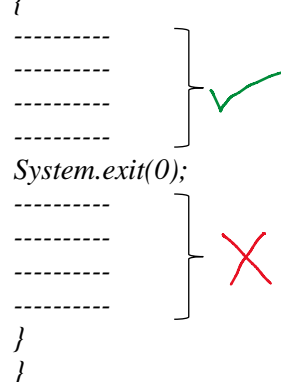
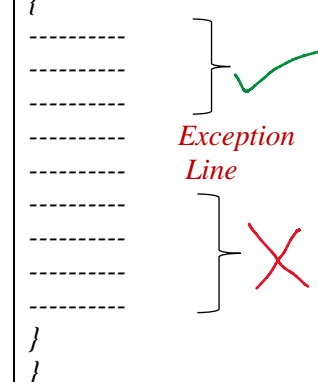
The error which occurs during compilation is known as Compile Time Error(**CTE**).

The error which occurs during execution of program is known as Run Time Error(**RTE**).

## NOTE :

1. Exceptions are also a type of run-time error.

## Types of Program Termination :

<pre>class A {     p s v m(-)     {         -----         -----         -----         -----         -----         -----         -----         -----         -----         -----     } }</pre>	<pre>class B {     p s v m(-)     {         -----         -----         -----         -----         -----         System.exit(0);         -----         -----         -----         -----     } }</pre> 	<pre>class C {     p s v m(-)     {         -----         -----         -----         -----         -----         -----         -----         -----         -----         -----     } }</pre> 
Normal Termination	Forceful Termination	Abnormal Termination

## Default Exception Handler :

When an exception occurs in the program, the JVM will analyse and create respective type of exception object: the JVM will throw the exception object back to the program. If the user has not returned any code to handle the exception object, the JVM will handle it by giving the exception object to **default exception handler**.

The default exception handler will first terminate the program and display the exception message.

## Example :

```
public class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Starts");
        int x=10;
        int y=0;
        System.out.println("x/y = "+(x/y));
        System.out.println("Main Ends");
    }
}
```

```
}
}
```

**Output :**

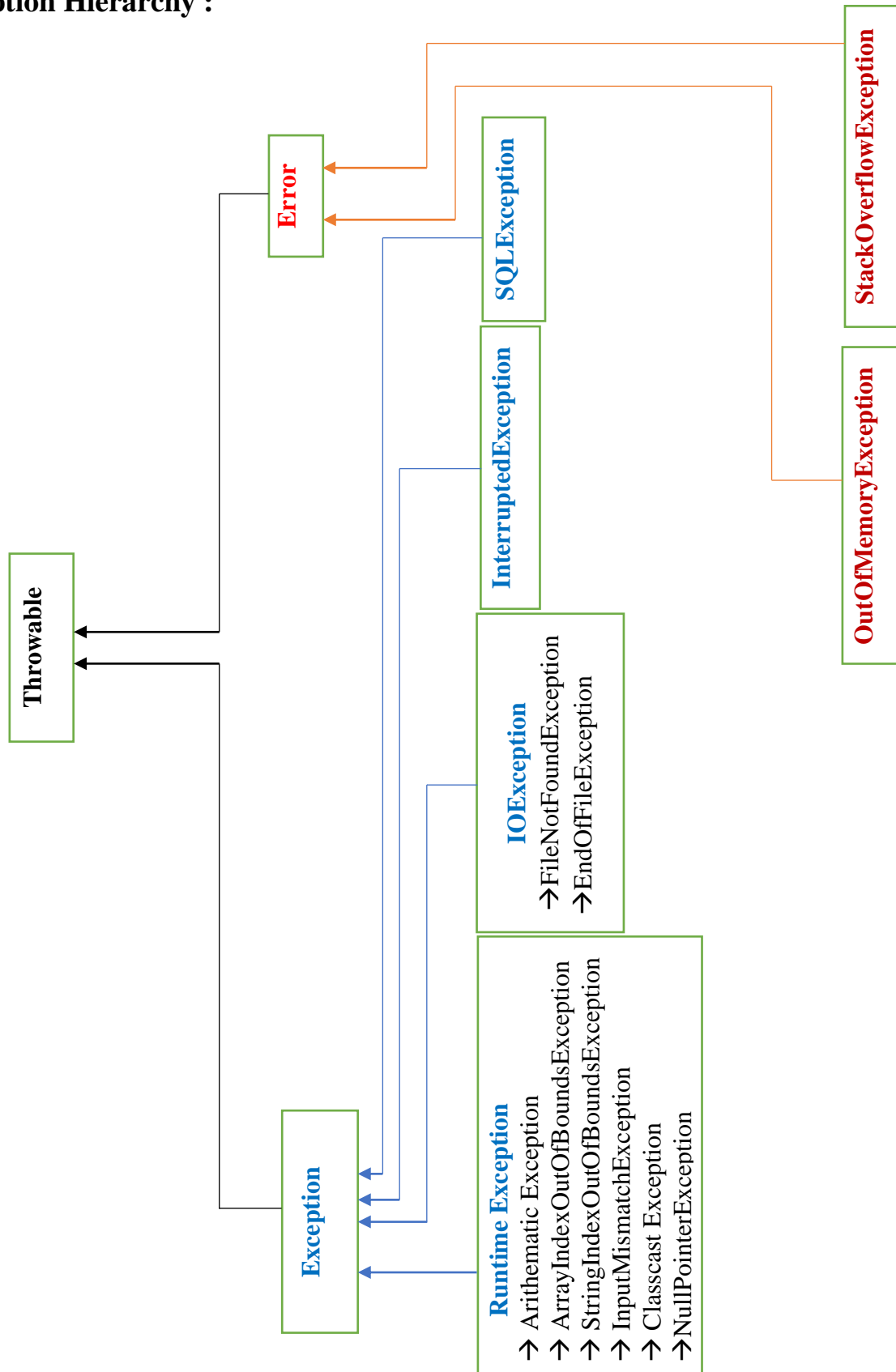
Main Starts

Exception in thread "main" [java.lang.ArithmeticException](#): / by zero

at

[edu.jspiders.MyProject.Assignment2.A.main\(A.java:8\)](#)

## Exception Hierarchy :





1. In Java, every exception is a class.
2. RuntimeException is also known as **unchecked exception**.
3. IOException, InterruptedException & SQLException are also known as **checked exception**.
4. Whenever the default exception handler handles the exception, the program will be terminated abnormally i.e., the code written after exception line will never get executed.

If we want normal termination of a program even after exception being occurred, the user should handle the exception explicitly by using any of the exception handlers as mentioned below:

#### Exception Handlers :

-----

- **try** block
- **catch** block
- **throw** keyword
- **throws** keyword
- **finally** blocks

#### try & catch block :

- ✓ Within the **try** block, we write those lines of code which might throw an exception.
- ✓ In **catch** block, we will write the code to handle the **exception object** which is thrown from the **try** block.
- ✓ The **catch** block will be executed only when exception occurs in the **try** block.
- ✓ It is non-mandatory that the code written in **try** block will always throw an exception, it might or it might not.

#### Syntax :

```
try
{
    -----
    ----- → those lines of code which might throw exception
    -----
}
catch(ExceptionClass ref_var)
{
    -----
    ----- → lines of code to handle the exception that occurred in try block
    -----
}
```

#### Example 1 :

```
public class A
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        int x=10;
        int y=0;
        try
        {
            System.out.println("x/y = "+(x/y));
        }
        catch(ArithmeticException a1)
        {
            System.out.println("Exception Handled!!");
        }
    }
}
```

```

    }
    System.out.println("main ends");
}
}

```

### OUTPUT :

main starts  
Exception Handled!!  
main ends

### Example 2 :

```

public class A
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        int x=10;
        int y=2;
        try
        {
            System.out.println("x/y = "+(x/y));
        }
        catch(ArithmeticException a1) //it will not executed as no exception occurred in try block
        {
            System.out.println("Exception Handled!!");
        }
        System.out.println("main ends");
    }
}

```

### OUTPUT :

main starts  
x/y = 5  
main ends

- ✓ It is recommended to write minimum lines of code inside **try** block because once the JVM leaves the try block, it will not come back again as shown in the below scenario:

```

public class Demo1
{
    public static void main(String[] args)
    {
        int x=10;
        int y=0;
        System.out.println("main starts");
        try
        {
            System.out.println("try starts");
            System.out.println("x/y = "+(x/y));
            System.out.println("try ends");
        }
        catch(ArithmeticException a1)
        {
            System.out.println("catch starts");
            System.out.println("exception handled!!");
            System.out.println("catch ends");
        }
        System.out.println("main ends");
    }
}

```

```
}  
}
```

**OUTPUT :**

main starts  
try starts  
catch starts  
exception handled!!  
catch ends  
main ends

**NOTE :**

Whenever we try to divide a number by zero, we will get `ArithmeticException`.

- ✓ While extracting character by character from a string; if we go below or beyond the index, we will end up getting **`StringIndexOutOfBoundsException`**.

```
public class Demo1  
{  
    public static void main(String[] args)  
    {  
        String s="Hi";  
        System.out.println("main starts");  
        try  
        {  
            System.out.println(s.charAt(0));  
            System.out.println(s.charAt(1));  
            System.out.println(s.charAt(2));  
        }  
        catch(StringIndexOutOfBoundsException a1)  
        {  
            System.out.println("exception handled!!");  
        }  
        System.out.println("main ends");  
    }  
}
```

**OUTPUT :**

main starts  
H  
i  
exception handled!!  
main ends

- ✓ While inserting or extracting values from an array: if we go below or beyond the index, we will end up getting **`ArrayIndexOutOfBoundsException`**.

```
public class Demo1  
{  
    public static void main(String[] args)  
    {  
        int a[]={1,5,6,3,8};  
        System.out.println("main starts");  
        try  
        {  
            System.out.println(a[1]);  
            System.out.println(a[4]);  
        }  
    }  
}
```

```

        System.out.println(a[6]);
    }
    catch(ArrayIndexOutOfBoundsException a1)
    {
        System.out.println("exception handled!!");
    }
    System.out.println("main ends");
}
}

```

#### OUTPUT :

```

main starts
5
8
exception handled!!
main ends

```

- ✓ Whenever we try to access the non-static properties of a class using a reference variable which is not pointing to any object, we might end up getting **NullPointerException**.

```

public class Demo1
{
    int x=10;
    int y=20;
    static Demo1 d1;
    public static void main(String[] args)
    {
        try
        {
            System.out.println(d1.x);
            System.out.println(d1.y);
        }
        catch(NullPointerException a1)
        {
            System.out.println("Exception handled");
        }
    }
}

```

#### OUTPUT :

```

Exception Handled

```

- ✓ Under class typecasting concept, whenever we try to perform downcasting without performing upcasting, we end up getting **ClassCastException**.

ClassCastException

-----

Object obj=new Demo(); *//upcasting*

Demo d = (Demo)obj; *//downcasting the upcasted obj.(no issues)*

*//downcasting without upcasting*

Demo d = new Object(); *//ClassCastException*

- ✓ While reading input from a user, if the input type and the variable type are not same: we will end up getting **InputMismatchException**.

Example:

```
sop("enter a value");
int a = sc.nextInt();
o/p:
```

-----  
Enter a value

10.12

**InputMismatchException**

### try with multiple catch block :

If there is a possibility of multiple exception being occurred in the program, we can write multiple catch blocks but out of all the catch blocks, any one will execute during execution of the program depending upon which exception has occurred first.

```
Syntax ;
try
{
    -----
}
catch(-----)
{
    -----
}
catch(-----)
{
    -----
}
```

### example ;

```
public class Demo1
{
    public static void main(String[] args)
    {
        try
        {
            System.out.println(10/0); //1st exception to occur
            Demo1 d1=(Demo1)new Object();//will not be executed by JVM
        }
        catch(ClassCastException a1)
        {
            System.out.println("CCE Exception handled");
        }
        catch(ArithmeticException d1)
        {
            System.out.println("AE Exception handled");
        }
    }
}
```

### OUTPUT :

AE Exception handled

## Multiple try & catch block :

If we want to handle multiple exception in a single program, we will make use of multiple try & catch block.

Syntax :

```
try
{
    -----
}
catch(-----)
{
    -----
}
try
{
    -----
}
catch(-----)
{
    -----
}
```


### Example :

```
import java.util.InputMismatchException;
import java.util.Scanner;
public class Demo1
{
    public static void main(String[] args)
    {
        Scanner sc=new Scanner(System.in);
        try
        {
            System.out.println("Enter a value");
            int a=sc.nextInt();
            System.out.println("a = "+a);
        }
        catch(InputMismatchException a1)
        {
            System.out.println("IME handled!");
        }
        try
        {
            System.out.println(10/0);
        }
        catch(ArithmeticException d1)
        {
            System.out.println("AE Exception handled");
        }
    }
}
```


### OUTPUT :

```
Enter a value
da
IME handled!
AE Exception handled
```


1.  
try  
{  
-----  
}  
catch(-----)  
{  
-----  
}




2.  
try  
{  
-----  
}




3.  
catch(-----)  
{  
-----  
}



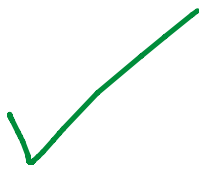
4.  
try  
{  
-----  
}  
System.out.println("hi");  
catch(-----)  
{  
-----  
}




5.  
catch(-----)  
{  
-----  
}  
try  
{  
-----  
}



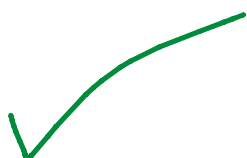
6.  
try  
{  
-----  
}  
catch(-----)  
{  
-----  
}  
catch(-----)  
{  
-----  
}



7.  
try  
{  
-----  
}  
try  
{  
-----  
}  
catch  
{  
-----  
}



8.  
try  
{  
-----  
}  
catch(-)  
{  
-----  
}  
System.out.println("hi");  
try  
{  
-----  
}  
catch  
{  
-----  
}



## Specialized Catch Block vs Generalized Catch Block

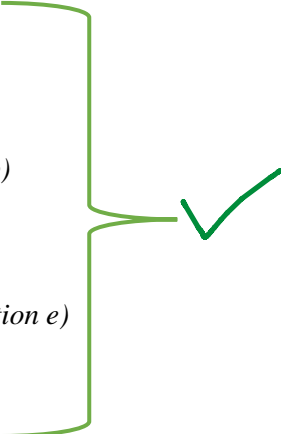
1. A catch block that accepts specific exception object is known as **specialized catch block**.
2. A catch block that accepts any type of exception object is known as **generalized catch block**.

```
try
{
    -----
}
Specialized catch Block {
    catch(AE e) //accepts only AE object
    {
        -----
    }
    catch(CCE a) //accepts only CCE object
    {
        -----
    }
}

try
{
    -----
}
Generalized catch Block {
    catch(Exception e) //accepts any type of exception object
    {
        -----
    }
}
```

3. It is possible to have both specialized catch block and generalized catch block in a single program but the specialized catch must be written first and then the generalized catch block.

```
try
{
    -----
}
catch(AE e)
{
    -----
}
catch(CCE b)
{
    -----
}
catch(Exception e)
{
    -----
}
```

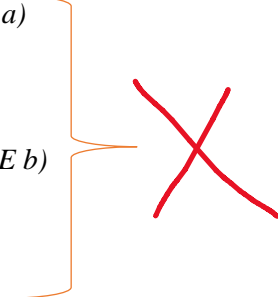




```

try
{
    -----
}
catch(Exception e)
{
    -----
}
catch(AE a)
{
    -----
}
catch(CCE b)
{
    -----
}

```



### String getMessage() :

It is an in-built method which is responsible for printing the exception message that has occurred in the try block.

Example :

```

public class Demo1
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        try
        {
            System.out.println(10/0);
        }
        catch(Exception e)
        {
            System.out.println("e1.getMessage() = "+e.getMessage());
        }
        System.out.println("main ends");
    }
}

```

### OUTPUT :

```

main starts
e1.getMessage() = / by zero
main ends

```

### void printStackTrace() :

It is an in-built method which is responsible for displaying the complete details about the exception occurred in try block.

Example:

```

public class Demo2
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        try
        {
            System.out.println(10/0);
        }
    }
}

```

```

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
    System.out.println("main ends");
}
}

```

**OUTPUT :**  
main starts  
[java.lang.ArithmeticException: / by zero](#)  
at edu.jspiders.MyProject.Assignment2.Demo2.main([Demo2.java:10](#))  
main ends

#### throw keyword :

1. Usually whenever exception occurs in the program , the JVM will create respective type of exception object and throw it back to the program but if the programmer wants to throw the exception object explicitly by themselves, they will make use of **throw** keyword.
2. Using **throw** keyword, the programmer can throw both checked and unchecked exception objects.
3. **throw** keyword can be used only for those class objects which are having the properties of **throwable** class.
4. Using **throw** keyword, we can throw only one exception object at a time.

#### Syntax :

```
throw new ExceptionType();
```

#### Example 1:

```
throw new ArithmeticException();
throw new ClassCastException();
```

#### Example 2 :

```
throw new Demo();
throw new Sample();
throw new A();
throw new ArithmeticException();
throw new ClassCastException();
throw new NullPointerException();
```

*invalid, but it can be valid only if these classes extend from **Throwable** class.*

**VALID**

#### Example 3:

```
throw new ArithmeticException(), new ClassCastException(); //invalid, only one object allowed
```

#### Important Note :

**throw** keyword is used by a developer when they are writing their own *customed exception classes*.

**Example**→ user throwing exception object and user handling it.

```

import java.util.Scanner;

public class Demo2
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        System.out.println("Enter 1st value:");
        int x = sc.nextInt();
    }
}

```

```

        System.out.println("Enter 2nd value:");
        int y = sc.nextInt();

        if(y!=0)
        {
            System.out.println("x/y = "+(x/y));
        }
        else
        {
            try
            {
                throw new ArithmeticException();
            }
            catch(ArithmeticException a)
            {
                System.out.println("Don't divide by 0");
            }
        }
    }
}

```

#### OUTPUT :

Enter 1st value:  
14  
Enter 2nd value:  
0  
Don't divide by 0

- User throwing exception object and default exception handler handles it.

```
import java.util.Scanner;
```

```

public class Demo4
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        String s = "Hello";

        System.out.println("Enter Index");
        int index = sc.nextInt();
        if(index>=0 && index<=s.length()-1)
        {
            System.out.println(s.charAt(index));
        }
        else
        {
            throw new StringIndexOutOfBoundsException();
        }
    }
}


```

#### OUTPUT :

Enter Index  
8  
Exception in thread "main" java.lang.StringIndexOutOfBoundsException  
at edu.jspiders.MyProject.Assignment2.Demo4.main(Demo4.java:20)

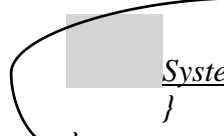
### Important Note :

```
public class Demo5
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        System.out.println(10/0);
        System.out.println("main ends");
    }
}
```



Decision of object creation and actual object creation will happen during runtime by JVM.

```
public class Demo6
{
    public static void main(String[] args)
    {
        System.out.println("main starts");
        throw new ArithmeticException();
        System.out.println("main ends");
    }
}
```



Decision of object creation done in compile time but object creation will happen during runtime.

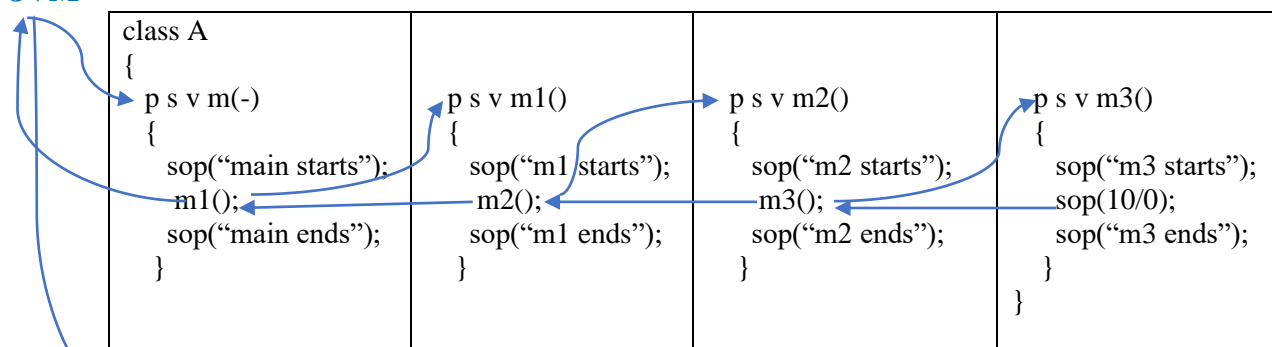
If a user is throwing exception object explicitly by themselves using the **throw** keyword, the **throw statement** must be the last statement.

## EXCEPTION PROPAGATION :

When any exception occurs in a user-defined method, the JVM will create respective type of exception object and throw it back to the program. If the user has not written any code to handle the exception object, the exception object will automatically propagate back to its column, this propagation will take place till the exception object reaches the JVM. Once the exception object reaches the JVM, it will be handled with the help of default exception handler.

**Exception propagation will come into the picture only if the user is not handling the exception object using try and catch block.**

### JVM



### DEFAULT EXCEPTION HANDLER

#### OUTPUT :

main starts  
m1 starts  
m2 starts  
m3 starts

**Exception Message**

## throws keyword :

Whenever an exception occurs in a method, the JVM will create respective type of exception object and throw it back to the method: if the user has not any code to handle the exception, the exception object will propagate back to the caller.

Automatic propagation capability is available only for **unchecked** exception object i.e., **checked** exception objects cannot propagate to caller by themselves. In order to propagate a checked exception, we must make use of **throws** keyword.

**throws** keyword will be used in the method declaration.

### Example :

1.

```
p s v m1() throws IOException
{
    -----
    -----
    -----
    throw new IOException();
}
```

2.

```
p s v Sort() throws SQLException
{
    ----
    ----
    throw new SQLException;
}
```

Using throws keyword, we can inform the caller about multiple checked exception at the same time.

```
p s v m1() throws IOException, SQLException, InterruptedException
{
    -----
    -----
    -----
    -----
}

import java.sql.SQLException;
public class Demo6
{
    public static void main(String[] args) throws SQLException
    {
        System.out.println("main starts");
        m1();
        System.out.println("main ends");
    }
    public static void m1() throws SQLException
    {
        System.out.println("m1 starts");
        m2();
        System.out.println("m1 ends");
    }
    public static void m2() throws SQLException
    {
        System.out.println("m2 starts");
        m3();
    }
}
```

```

        System.out.println("m2 ends");
    }
    public static void m3() throws SQLException
    {
        System.out.println("m3 starts");
        throw new SQLException();
    }
}

```

### OUTPUT :

main starts  
m1 starts  
m2 starts  
m3 starts

Exception in thread "main" [java.sql.SQLException](#)  
 at edu.jspiders.MyProject.Assignment2.Demo6.m3([Demo6.java:26](#))  
 at edu.jspiders.MyProject.Assignment2.Demo6.m2([Demo6.java:20](#))  
 at edu.jspiders.MyProject.Assignment2.Demo6.m1([Demo6.java:14](#))  
 at edu.jspiders.MyProject.Assignment2.Demo6.main([Demo6.java:8](#))

### Important Note :

1. *throws keyword concept come into picture only when the user is not handling the **checked** exception explicitly using try and catch block.*
2. *Differences between checked exception and unchecked exception.*

Checked Exception	Unchecked Exception
1. It is known at compile time but will occur during runtime.	1. It is known and occur during runtime.
2. It does not have capability of propagating by itself.	2. It can propagate automatically by itself.
3. throws keyword is mandatory if the user is not handling it.	3. throws keyword is not required even if user is not handling it.

3. *Differences between throw and throws keyword.*

throw	throws
1. It is used to throw exception objects explicitly by the user.	1. It is used to propagate checked exception back to the caller.
2. It is used inside method body or definition.	2. It is used in the method declaration.
3. Using throw keyword, we can throw only one exception object at a time.	3. Using throws keyword, we can inform the caller about multiple exception at the same time.
4. <pre> p s v m1() {     throw new ArithmeticException(); } </pre>	4. <pre> p s v m1() throw IOException {     ===== } </pre>

## finally block :

As per IT standards, all the closing statements must be written inside **finally** block. The catch block will be executed only if exception occurs in the try block but the finally block will mandatory execute irrespective of exception has occurred or not in the try block.















```
try
{
    //code that might throw exception
}
catch(-)
{
    //code to handle the exception object which is thrown from try block.
    // it will execute only if exception occurs in try block
}
finally
{
    //all the closing statements. //it will mandatory
    e.g.: close a file, close a server, close a DB connection....
}
```

### Example :

```
public class Demo7
{
    public static void main(String[] args)
    {
        System.out.println("Application started");
        int x=10;
        int y=0;
        try
        {
            System.out.println("x/y= "+(x/y));
        }
        catch(ArithmeticException e)
        {
            System.out.println("Exception message : "+e.getMessage());
        }
        finally
        {
            System.out.println("All connections closed");
        }
        System.out.println("Application terminated");
    }
}
```

### OUTPUT :

```
Application started
Exception message : / by zero
All connections closed
Application terminated
```

<b>1.</b> <pre>try {     sop(10/0); } catch(AE e1) {     ==== } finally {     === }</pre>  	<b>3.</b> <pre>try {     sop(10/6); } finally {     === }</pre>  <p><b>Note :</b>  <i>Exception handled by Default Exception Handler.</i></p>	<b>5.</b> <pre>try {     return; } catch(AE e1) {     ==== } finally {     === }</pre>  	<b>7.</b> <pre>catch(---) {     ==== } finally {     === }</pre> 
<b>2.</b> <pre>try {     sop(10/5); } catch(AE e1) {     ==== } finally {     === }</pre>  	<b>4.</b> <pre>try {     sop(10/5); } finally {     === }</pre> 	<b>6.</b> <pre>try {     System.exit(0); } catch(---) {     ==== } finally {     === }</pre>   <p><i>(only this scenario, finally block will not execute.)</i></p>	<b>8.</b> <pre>finally {     === }</pre> 
			<b>9.</b> <pre>try {     sop(10/5); } finally {     === } finally {     === }</pre>  

## Writing Customized Exception Classes :

### 1. Customized Unchecked Exception :

If a user wants to write their own customized unchecked exception classes, the user defined exception class must extend from **RuntimeException** class.

#### Example :

*Write a program to check if a citizen is eligible to vote or not. If the age is greater than 17, they can vote else throw **NotEligibleToVoteException**. Handle the exception explicitly.*

```
public class NotEligibleToVoteException extends RuntimeException
{
    @Override
    public String getMessage()
    {
        return "You cannot vote.";
    }
}

public class Validation
{
    public void check(int age)
    {
        if(age>17)
        {
            System.out.println("You can vote\nVote Wisely");
        }
    }
}
```



```

    }
    else
    {
        try
        {
            EXCEPTION OBJECT
            throw new NotEligibleToVoteException();
        }
        catch (NotEligibleToVoteException e)
        {
            System.out.println("Exception Message : "+e.getMessage());
        }
        finally
        {
            System.out.println("Connections closed");
        }
    }
}

import java.util.Scanner;

public class Client
{
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your age");
        int age = sc.nextInt();
        Validation v = new Validation();
        v.check(age);
    }
}

```

#### OUTPUT :

```

Enter your age
17
Exception Message : You cannot vote.
Connections closed

```

## 2. Customized Checked Exception :

If a user wants to write their own checked exception classes, the user-defined exception class must extend from **Exception** class.

### Example :

*Write a program to checked if a student is eligible for exam or not. If the attendance percentage of a student is more than 75, they are eligible else throw **NotEligibleForExamException**(Allow the default exception handler to handle the exception.)*

```

public class NotEligibleForExamException extends Exception
{
    @Override
    public String getMessage()
    {
        return "Come next semester.";
    }
}

public class Validate
{
    public void check(double att) throws NotEligibleForExamException
    {

```

```

        if(att>75)
        {
            System.out.println("You can write exam\nBest of Luck");
        }
        else
        {
            throw new NotEligibleForExamException();
        }
    }
}

public class Student
{
    public static void main(String[] args) throws NotEligibleForExamException
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter your attendance percentage");
        int att = sc.nextInt();
        Validate v = new Validate();
        v.check(att);
    }
}

```

#### OUTPUT :

Enter your attendance percentage

74

Exception in thread "main"

[edu.jspiders.JavaLearning.CustomCheckedException.NotEligibleForExamException](#): Come next semester.

at edu.jspiders.JavaLearning.CustomCheckedException.Validate.check([Validate.java:13](#))

at edu.jspiders.JavaLearning.CustomCheckedException.Student.main([Student.java:13](#))

# OBJECT CLASS & ITS METHODS

**Object class** is the super most class in entire Java in the hierarchy, that means every class in Java either directly or indirectly is the sub-class of object class. It will contain those properties which are frequently used by the programmer while developing the application. Object class will contain no-argument constructor(Default constructor).

## Methods of Object Class : 12 methods of object class

- i. `public String toString()`
- ii. `public int hashCode()`
- iii. `public boolean equals(Object obj)`
- iv. `final public void wait(long ms)`
- v. `final public void wait(long ms,int ns)` //ms → *millisecond* ns → *nanosecond*
- vi. `final public void notify()`
- vii. `final public void notifyAll()`
- viii. `public Class getClass()`
- ix. `public void finalize()`
- x. `protected Object clone()`
- xi. `private void register native()`

Object class is present in **Java.lang.package**. The user need not write the **import java.lang** statement because every time when the programmer written any code, the **Java.lang.package** properties will be imported **implicitly by the JVM**.

## EXPLANATION FOR METHODS OF OBJECT CLASS

### \* **public String toString() :-**

It is an in-built method of object class, which is responsible for returning the **String representation of an object**.

The String representation will contain the **qualified name** in the below format :

[packagename.classname@hashCode](#) in Hexadecimal format

### \* **public int hashCode() :-**

It is an in-built method of object class which is responsible for **returning a unique integer value** which is generated with the help of **Address of the object**.

If two objects are having same address, there hashCode value will be same.

### \* **public Boolean equals(Object obj) :-**

It is an in-built method of object class which is **responsible for composing the current object** with the given object.

If two objects are having same hashCode value, this method will return true , else it will return false.

### Example :-

```
public class Demo8
{
    public static void main(String[] args)
    {
        System.out.println("Working of toString()");
    }
}
```

```

        System.out.println("=====");
        Demo8 d1 = new Demo8();
        System.out.println("d1 = "+d1.toString());
        Object obj = new Object();
        System.out.println("Obj = "+obj.toString());
        Demo8 d2 = new Demo8();
        System.out.println("d2 = "+d2.toString());
        Demo8 d3 = d2;
        System.out.println("d3 = "+d3.toString());
        Demo8 d4 = d1;
        System.out.println("d4 = "+d4);
        System.out.println();
        System.out.println("Working of Hashcode");
        System.out.println("=====");
        System.out.println("d1 = "+d1.hashCode());
        System.out.println("Obj = "+obj.hashCode());
        System.out.println("d2 = "+d2.hashCode());
        System.out.println("d3 = "+d3.hashCode());
        System.out.println("d4 = "+d4.hashCode());
        System.out.println();
        System.out.println("Working of equals(Object obj)");
        System.out.println("=====");
        System.out.println(d1.equals(d2));
        System.out.println(d1.equals(d3));
        System.out.println(d2.equals(d3));
        System.out.println(d2.equals(d4));
        System.out.println(d1.equals(d4));
    }
}

```

## OUTPUT :

Working of toString()

```

=====
d1 = edu.jspiders.MyProject.Assignment2.Demo8@15db9742
Obj = java.lang.Object@6d06d69c
d2 = edu.jspiders.MyProject.Assignment2.Demo8@7852e922
d3 = edu.jspiders.MyProject.Assignment2.Demo8@7852e922
d4 = edu.jspiders.MyProject.Assignment2.Demo8@15db9742

```

Working of Hashcode

```

=====
d1 = 366712642
Obj = 1829164700
d2 = 2018699554
d3 = 2018699554
d4 = 366712642

```

Working of equals(Object obj)

```

=====
false
false
true
false
true

```

### Important Note :

1. The **toString()** method can be overridden to print the content of the object **instead of printing the fully qualified name**.
2. The **hashCode()** method can be overridden to generate a **unique value** based on the content of the object instead of the **address of the object**.
3. The **equals(Object obj)** method can be overridden to compare the content of the objects instead of comparing the hashCode of the object.

### Example :-

```
public class Student
{
    int id;
    String name;
    double marks;
    public Student(int id, String name, double marks)
    {
        this.id = id;
        this.name = name;
        this.marks = marks;
    }
    @Override
    public String toString()
    {
        return this.id+" "+this.name+" "+this.marks;
    }
    @Override
    public int hashCode()
    {
        return this.id;
    }
}

public class MainOfStudent
{
    public static void main(String[] args)
    {
        System.out.println("Working of Overridden toString()");
        System.out.println("=====");
        Student s1 = new Student(112,"Sagar",59.6);
        System.out.println("s1 = "+s1);
        Student s2 = new Student(114,"Luri",79.6);
        System.out.println("s2 = "+s2);
        System.out.println();
        System.out.println("Working of Overridden tohashCode()");

        System.out.println("=====");
        System.out.println("s1 = "+s1.hashCode());
        System.out.println("s2 = "+s2.hashCode());
    }
}
```

### OUTPUT :

Working of Overridden toString()

```
=====
s1 = 112 Sagar 59.6
s2 = 114 Luri 79.6
```

## Working of Overridden hashCode()

```
s1 = 112  
s1 = 114
```

Create a faculty class with the attributes id, name, subject and salary. Initialise these attributes with the help of constructor for 4 faculties and perform the below operation:

a. Display the faculty details in table format.

b. Find out the faculty name who is teaching the same subject as that of Suresh.

```
public class Faculty  
{  
    int id;  
    String name;  
    String Subject;  
    double sal;  
    public Faculty(int id, String name, String subject, double sal)  
    {  
        this.id = id;  
        this.name = name;  
        Subject = subject;  
        this.sal = sal;  
    }  
    @Override  
    public String toString()  
    {  
        return this.id + "\t" + this.name + "\t" + this.Subject + "\t" + this.sal;  
    }  
    @Override  
    public boolean equals(Object obj)  
    {  
        Faculty f=(Faculty)obj;  
        return this.Subject.equals(f.Subject);  
    }  
}  
public class MainOfFaculty  
{  
    public static void main(String[] args)  
    {  
        Faculty f1= new Faculty(114,"Suresh","Java",420.13);  
        Faculty f2= new Faculty(112,"Dinga","SQL",474.22);  
        Faculty f3= new Faculty(113,"Ringa","Java",453.3);  
        Faculty f4= new Faculty(115,"Tinga","Web Tech",420.13);  
  
        //display all the details of faculty in table format  
        System.out.println("Id\tName\tSubject\tSalary");  
        System.out.println("=====");  
        System.out.println(f1);  
        System.out.println(f2);  
        System.out.println(f3);  
        System.out.println(f4);  
  
        //find out the faculty who is teaching the same subject as Suresh  
        if(f1.equals(f2))  
        {
```

```

        System.out.println(f1.name+" and "+f2.name+" are teaching same subject");
    }
    else if(f1.equals(f3))
    {
        System.out.println(f1.name+" and "+f3.name+" are teaching same subject");
    }
    else if(f1.equals(f4))
    {
        System.out.println(f1.name+" and "+f4.name+" are teaching same subject");
    }
    else
    {
        System.out.println("No match found!!");
        System.exit(0);
    }
}
}

```

### OUTPUT :

Id	Name	Subject	Salary
114	Suresh	Java	420.13
112	Dinga	SQL	474.22
113	Ringa	Java	453.3
115	Tinga	Web Tech	420.13

Suresh and Ringa are teaching same subject

*Create a Car class with the attributes name, color, price and brand. Initialize these attributes with the help of a Constructor for 5 car and perform the below operation*

a. Display all the Car details.

b. Display the Car name whose price is same as that of “Fortuner”.

```

public class Car
{
    String name;
    String color;
    double price;
    String brand;
    public Car(String name, String color, double price, String brand)
    {
        this.name = name;
        this.color = color;
        this.price = price;
        this.brand = brand;
    }
    @Override
    public String toString()
    {
        return this.name+"\t"+this.color+"\t"+this.price+"\t"+this.brand;
    }
    @Override
    public boolean equals(Object obj)
    {
        Car c = (Car)obj;
        return this.price==c.price;
    }
}

```

```

}
public class MainOfCar
{
    public static void main(String[] args)
    {
        Car c1 = new Car("TATA", "BLACK", 449000, "INDICA");
        Car c2 = new Car("ROLLS ROYCE", "WHITE", 9449000, "PHANTOM");
        Car c3 = new Car("TOYOTA", "BLUE", 649499, "FORTUNER");
        Car c4 = new Car("MERCEDES", "BLACK", 3449000, "BENZ");
        Car c5 = new Car("LAMBORGHINI", "RED", 649499, "URUS");

        //Display all the Car details
        System.out.println("Name\t\tColour\t\tPrice\t\tBrand");
        System.out.println("=====");
        System.out.println(c1);
        System.out.println(c2);
        System.out.println(c3);
        System.out.println(c4);
        System.out.println(c5);

        //To find the car name whose price is same as that of Fortuner
        if(c3.equals(c1))
        {
            System.out.println(c1.name+" and "+c3.name+" have same price.");
        }
        else if(c3.equals(c2))
        {
            System.out.println(c2.name+" and "+c3.name+" have same price.");
        }
        else if(c3.equals(c4))
        {
            System.out.println(c4.name+" and "+c3.name+" have same price.");
        }
        else if(c3.equals(c5))
        {
            System.out.println(c5.name+" and "+c3.name+" have same price.");
        }
        else
        {
            System.out.println("Program terminated.");
            System.exit(0);
        }
    }
}

```

#### OUTPUT:

Name	Colour	Price	Brand
TATA	BLACK	449000.0	INDICA
ROLLS ROYCE	WHITE	9449000.0	PHANTOM
TOYOTA	BLUE	649499.0	FORTUNER
MERCEDES	BLACK	3449000.0	BENZ
LAMBORGHINI	RED	649499.0	URUS

LAMBORGHINI and TOYOTA have same price.



Create a Watch class with the attributes hr, min and sec. Initialize these attributes with the Constructor for 4 watches and perform the below operation:

a. Display all the watch timings in the format => hr:min:sec

b. Find out watches whose timings are same.

```
public class Watch
{
    int hr;
    int min;
    int sec;
    public Watch(int hr, int min, int sec)
    {
        this.hr = hr;
        this.min = min;
        this.sec = sec;
    }
    @Override
    public String toString() {
        return this.hr + ":" + this.min + ":" + this.sec;
    }
    @Override
    public boolean equals(Object obj)
    {
        Watch w = (Watch)obj;
        return this.hr == w.hr && this.min == w.min && this.sec == w.sec;
    }
}

public class MainOfClass
{
    public static void main(String[] args)
    {
        Watch w1 = new Watch(03,15,20);
        Watch w2 = new Watch(21,23,41);
        Watch w3 = new Watch(13,35,37);
        Watch w4 = new Watch(03,15,20);

        //Display all the watch timings in the format
        System.out.println("Hour:Minutes:Second");
        System.out.println("-----");
        System.out.println(w1);
        System.out.println(w2);
        System.out.println(w3);
        System.out.println(w4);

        //To find out watches whose timings are same.
        if(w1.hr == w2.hr && w1.min == w2.min && w1.sec == w2.sec)
        {
            System.out.println("1st and 2nd watches have same timings");
        }
        else if(w1.hr == w3.hr && w1.min == w3.min && w1.sec == w3.sec)
        {
            System.out.println("1st and 3rd watches have same timings");
        }
        else if(w1.hr == w4.hr && w1.min == w4.min && w1.sec == w4.sec)
        {
            System.out.println("1st and 4th watches have same timings");
        }
    }
}
```

```

    }
    else if(w2.hr==w3.hr && w2.min==w3.min && w2.sec==w3.sec)
    {
        System.out.println("2nd and 3rd watches have same timings");
    }
    else if(w4.hr==w3.hr && w4.min==w3.min && w4.sec==w3.sec)
    {
        System.out.println("1st and 3rd watches have same timings");
    }
    else if(w2.hr==w4.hr && w2.min==w4.min && w2.sec==w4.sec)
    {
        System.out.println("2nd and 4th watches have same timings");
    }
    else
    {
        System.out.println("No match found");
        System.exit(0);
    }
}
}

```

#### OUTPUT:

Hour:Minutes:Second

-----

3:15:20

21:23:41

13:35:37

3:15:20

1st and 4th watches have same timings.

# STRING CLASS AND ITS METHODS

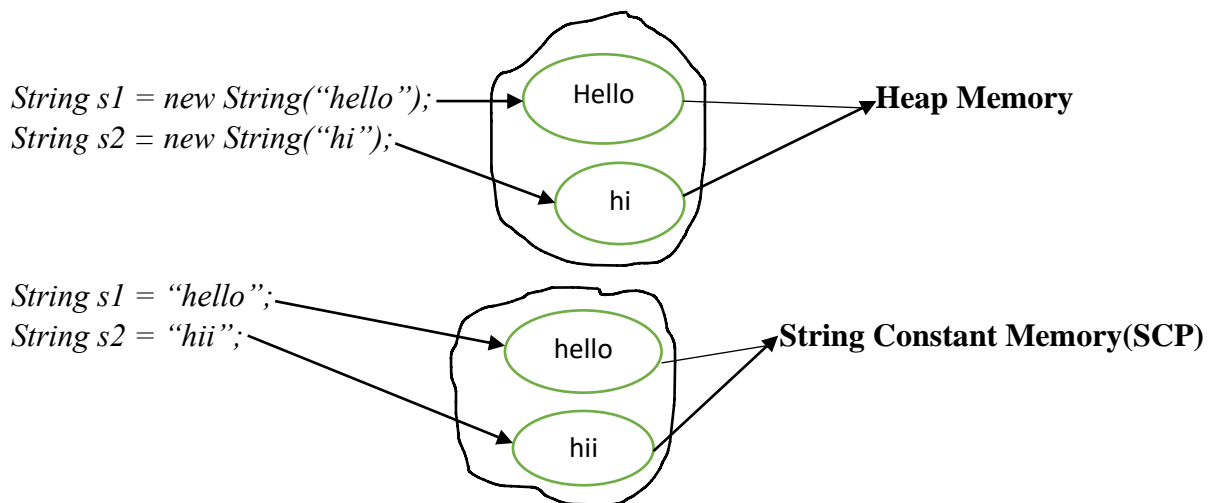
**String class** is a pre-defined class which is available in **java.lang** package.

It is a **final class** i.e., string class cannot have a sub-class but it can have super class(Object class).

It is an **immutable class**.

In entire java, string class is the only class whose object can be created both using **new** keyword & without using **new** keyword. Whenever we create a string class object using the **new** keyword, the object will be stored in heap memory or heap area. Whenever we create a string class object without using the **new** keyword, the object will be stored in **String Constant Pool(SCP)**. SCP is a special memory associated with String class which will come into the existence when we create String class object without using the **new** keyword.

**Example:**



## METHODS OF STRING CLASS :

- 1) `public int length()`
- 2) `public int indexOf(char ch)`
- 3) `public int lastIndexOf(char ch)`
- 4) `public boolean startsWith(String s)`
- 5) `public boolean endsWith(String s)`
- 6) `public boolean equalsIgnoreCase(String s)`
- 7) `public String substring(int start_pos)`
- 8) `public String substring(int start_pos, int end_pos)`
- 9) `public String concat(String s)`
- 10) `public String toUpperCase()`
- 11) `public String toLowerCase()`
- 12) `public String replace(String old_str, String new_str)`
- 13) `public String toString()` *//overridden method of Object class*
- 14) `public int hashCode()` *//overridden method of Object class*
- 15) `public boolean equals(Object obj)` *//overridden method of Object class*

### Important Note :

The following methods of **Object** class are overridden in the **String** class.

- 1) **toString() :**  
It is overridden to print the content of the object instead of printing the fully qualified name.
- 2) **hashCode() :**  
It is overridden to generate a unique integer based on the content of an object instead of the address.

### 3) *equals(Object obj)* :

It is overridden to compare the content of two string objects. If the content is same, it will return true else it will return false.

**Example :**

```
public class Demo1
{
    public static void main(String[] args)
    {
        String s1="hello";
        String s2 = new String("Welcome");
        String s3 = "hello";
        String s4 = "Hello";

        System.out.println("**toString()");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(s4);

        System.out.println();

        System.out.println("**hashCode()**");
        System.out.println("s1 = "+s1.hashCode());
        System.out.println("s2 = "+s2.hashCode());
        System.out.println("s3 = "+s3.hashCode());
        System.out.println("s4 = "+s4.hashCode());

        System.out.println();
        System.out.println("**equals(Object obj)**");
        System.out.println(s1.equals(s2));
        System.out.println(s3.equals(s3));
    }
}
```

**OUTPUT :**

\*\*toString()

hello

Welcome

hello

Hello

\*\*hashCode()\*\*

s1 = 99162322

s2 = -1397214398

s3 = 99162322

s4 = 69609650

\*\*equals(Object obj)\*\*

false

true

## WORKING OF IN-BUILT METHODS OF STRING CLASS :

```
public class Demo1
{
    public static void main(String[] args)
    {
        String s = "Welcome to Nepal";

        //It will return the total number of characters.
        System.out.println("s.length = "+s.length());

        //it will return char at the given index
        System.out.println("s.indexOf(a) = "+s.indexOf('a'));

        //it will return the last index of the string
        System.out.println("s.lastIndexOf(e) = "+s.lastIndexOf('e'));

        //it will check if the actual string starts with the given string
        System.out.println("s.startsWith(we)= "+s.startsWith("we"));

        //it will check if the actual string ends with the given string
        System.out.println("s.endsWith(al) = "+ s.endsWith("al"));

        //it will compare and check if the both strings are equal or not.
        System.out.println(s.equalsIgnoreCase("welcome to nepal"));

        //it will throw the string from given index of main string
        System.out.println(s.substring(5));

        //it will throw the string of given opening and ending index from main string
        System.out.println(s.substring(5,12));

        //it is used to concatenate the Strings
        System.out.println(s.concat(" Sagar"));

        //It is used to make uppercase of the Strings
        System.out.println(s.toUpperCase());

        //It is used to make lowercase of the Strings
        System.out.println(s.toLowerCase());

        //It is used to replace old string with new string
        System.out.println(s.replace("Nepal", "Heaven"));
    }
}
```

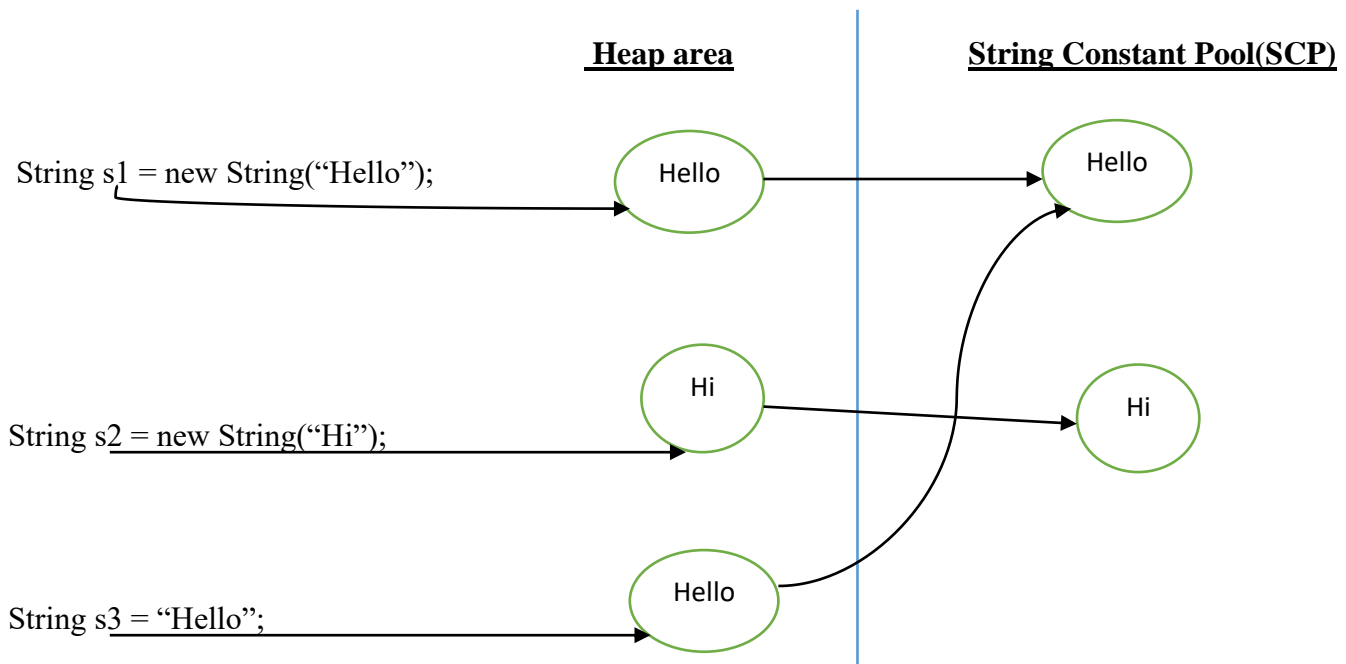
### OUTPUT :

```
s.length = 16
s.indexOf(a) = 14
s.lastIndexOf(e) = 12
s.startsWith(we)= false
s.endsWith(al) = true
true
me to Nepal
me to N
Welcome to Nepal Sagar
```

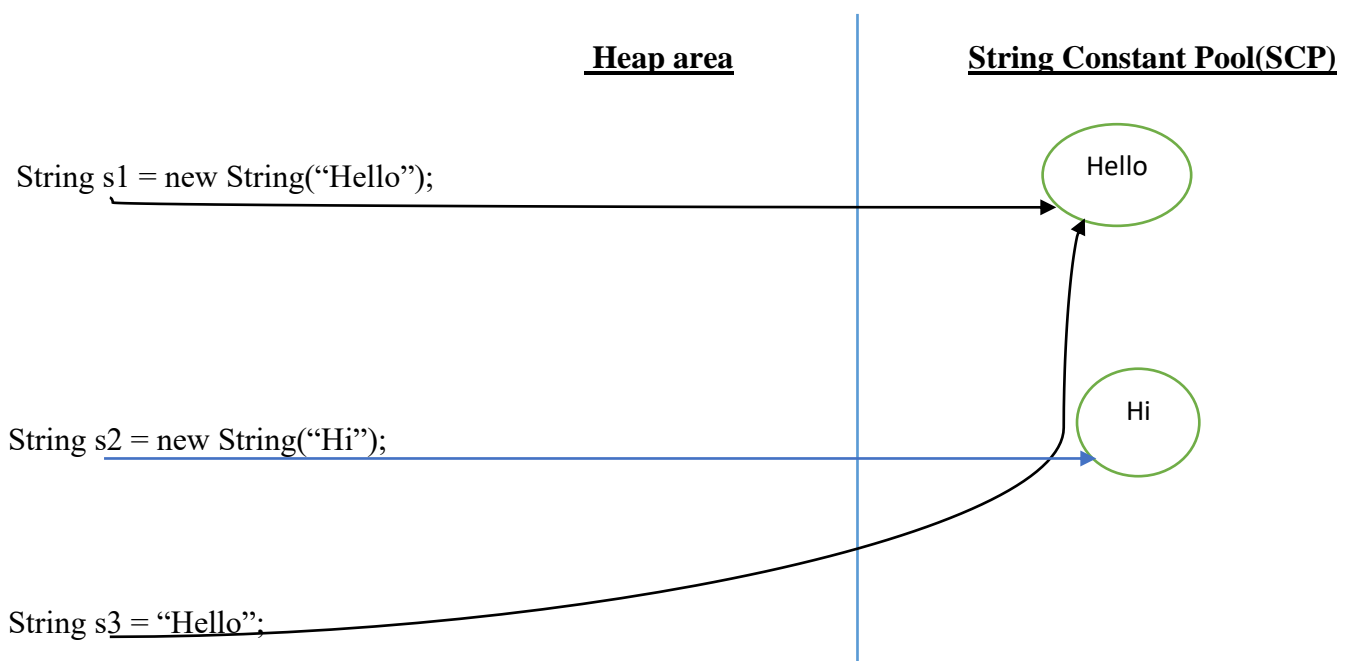
WELCOME TO NEPAL  
welcome to nepal  
Welcome to Heaven

## HEAP MEMORY VS STRING CONSTANT POOL(SCP)

- 1) Whenever we create a String class object using the **new** keyword, the object will be loaded inside the heap memory but implicitly a copy of the object will also be created inside the String Constant Pool(SCP) which will not be referred by any reference variable.
- 2) Every time when we use the **new** keyword, a String object will be created in the heap memory irrespective of the content.

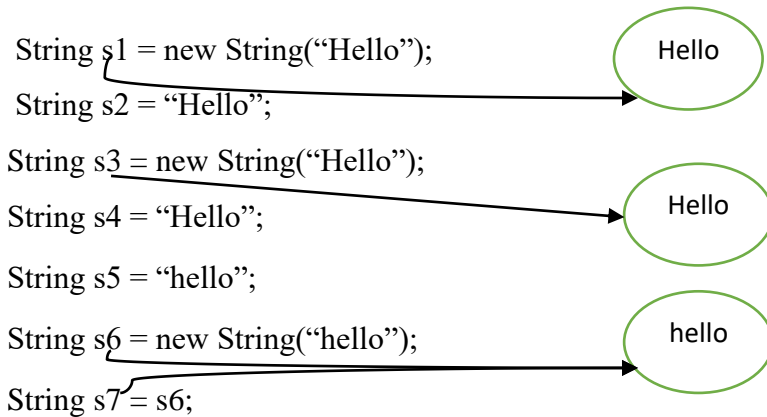


- 3) When we create a String class object without using the **new** keyword, the object will be loaded inside the String Constant Pool and no copy will be created inside the heap memory.
- 4) If the content is same, the multiple reference variable can point to the same object.



- 5) In the heap memory, duplicate objects are allowed whereas in the String Constant Pool, duplicate objects will not be allowed.

### Heap area



```
System.out.println(s1.equals(s4)); //true
```

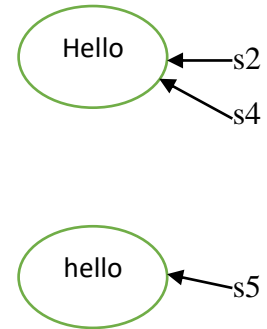
```
System.out.println(s1==s2); //false
```

```
System.out.println(s6==s7); //true
```

```
System.out.println(s7.equals(s5)); //true
```

```
System.out.println(s2==s4); //true
```

### String Constant Pool(SCP)



### **split()** method :

It is used to count the number of words in a sentence.

example :

```
public class Demo1
{
    public static void main(String[] args)
    {
        String s="This is Sagar.";
        String[] str = s.split(" ");
        System.out.println("no of words in given sentence is "+str.length);
    }
}
```

### **OUTPUT :**

no of words in given sentence is 3





# CLONING

Obtaining the replica of an object is known as **cloning**. The object address of cloning will be different than that of cloned object.

**OR**

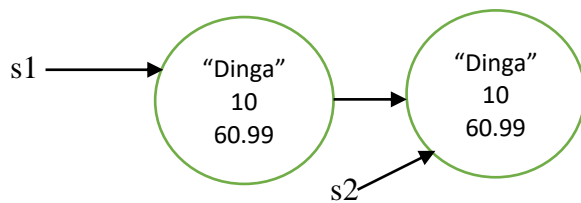
The process of creating a copy of the existing object is known as cloning. To perform cloning, the object must implement **Cloneable(I)** present in **java.lang** package.

The method used to perform cloning is present in **Object** class.

→ Protected Object clone() throws CloneNotSupportedException.

Cloneable(I) is a marker interface.

If we try to clone the object which is not implementing Cloneable(I), we will end up with CloneNotSupportedException.



**Object clone()**

**Example of Cloning :**

**package** edu.jspiders.rakshith.cloning;

```
public class Test implements Cloneable
{
    int x;
    public Test(int x)
    {
        this.x=x;
    }
    @Override
    public Object clone() throws CloneNotSupportedException
        //Visibility of Object class is increased to public from protected
    {
        return super.clone();
    }
    @Override
    public String toString() {
        return "Test [x=" + x + "]";
    }
}
```

```
public class CloneDemo
{
    public static void main(String[] args) throws CloneNotSupportedException{
        Test t1 = new Test(10);
        System.out.println("Original object = "+t1);
        Object obj = t1.clone();
        Test copyt1 = (Test)obj;
        System.out.println("Cloned object = "+copyt1);
    }
}
```

## OUTPUT :

Original object = Test [x=10]

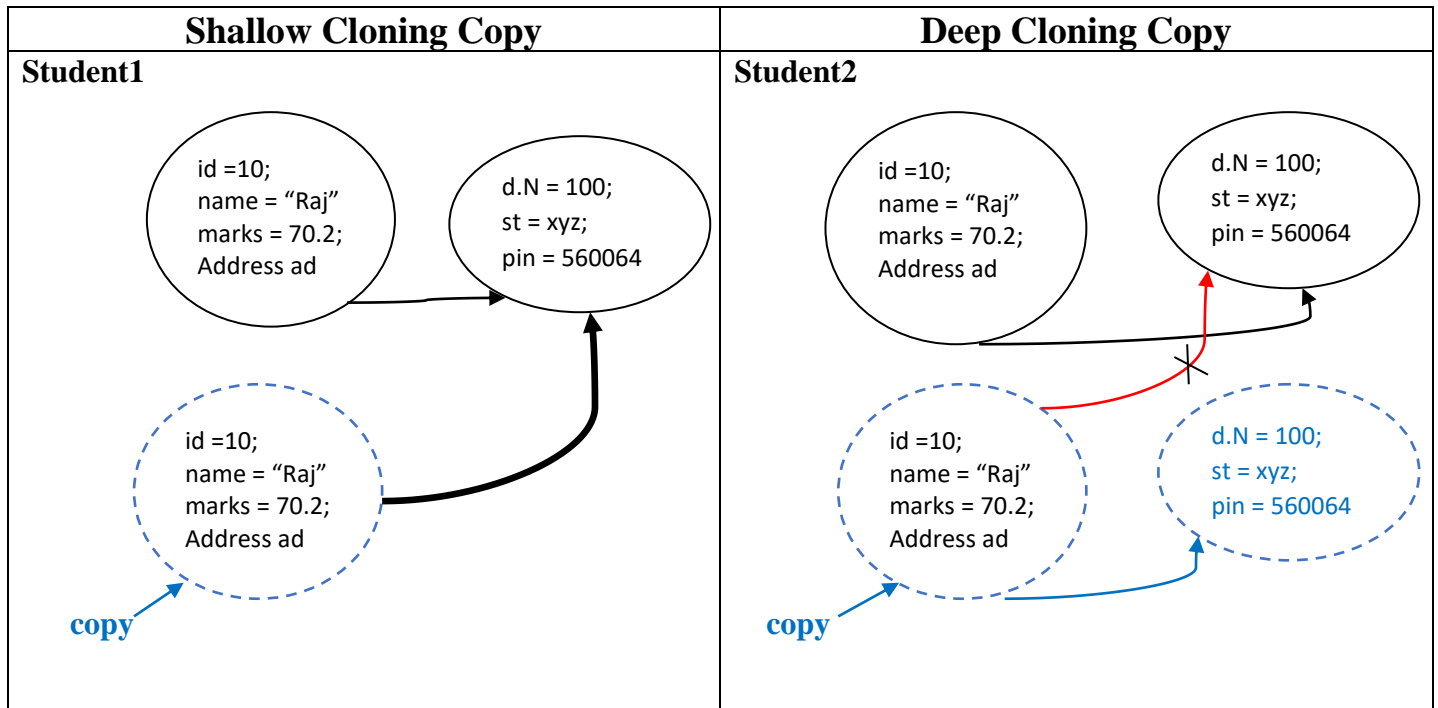
Cloned object = Test [x=10]

## Types of Cloning :

i. Shallow Cloning

ii. Deep Cloning

iii. Lazy Copy



## → Has-a relationship:

Using a reference variable of any class as data member in another class, this relationship is called as **has-a** relationship.

```
public class Address
{
    private String Address;
    public Address(String address)
    {
        super();
        Address = address;
    }
    public String getAddress()
    {
        return Address;
    }
    public void setAddress(String address)
    {
        Address = address;
    }
    @Override
    public String toString()
    {
        return Address;
    }
}
```

```

public class Student implements Cloneable
{
    private String name;
    private int id;
    private double marks;
    private Address addr; //has-A relationship

    public Student(String name, int id, double marks, Address addr)
    {
        this.name = name;
        this.id = id;
        this.marks = marks;
        this.addr = addr;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }

    public double getMarks()
    {
        return marks;
    }

    public void setMarks(double marks)
    {
        this.marks = marks;
    }

    public Address getAddr()
    {
        return addr;
    }

    public void setAddr(Address addr)
    {
        this.addr = addr;
    }
}

```

```

        @Override
        public String toString()
        {
            return id + "\t\t" + name + "\t\t" + marks + "\t\t" + addr;
        }

        @Override
        public Object clone() throws CloneNotSupportedException
        {
            return super.clone();
        }
    }

    public class MainOfStudent
    {
        public static void main(String[] args) throws CloneNotSupportedException
        {
            Student s1 = new Student("Sagar", 12, 59.25, new Address("Butwal"));
            Student s2 = (Student)s1.clone();
            System.out.println(s1);
            System.out.println(s2);
            s1.getAddr().setAddress("Nepal");
            s2.setName("Suri");
            System.out.println(s1);
            System.out.println(s2);
        }
    }

```

#### OUTPUT :

12	Sagar	59.25	Butwal
12	Sagar	59.25	Butwal
12	Sagar	59.25	Nepal
12	Suri	59.25	Nepal

#### → Shallow Cloning :

Whenever we use default implementation of clone method, we get shallow copy of object means it creates new instance and copies all the field of object to that new instance and returns it as object type, we need to explicitly cast it back to our original object. This is **shallow cloning**.

clone() method of the object class support shallow copy of the object. If the object contains primitive as well as non-primitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references gets copied and not the referred objects themselves.

If only primitive type fields or Immutable objects are there then there is no difference between shallow and deep copy in Java.

```

package edu.jspiders.Clonning.Examples;

public class Student implements Cloneable
{
    private String name;
    private int id;
    private double marks;
    public Student(String name, int id, double marks)
    {
        this.name = name;
        this.id = id;
        this.marks = marks;
    }
}

```

```

    }
    public String getName()
    {
        return name;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    public int getId()
    {
        return id;
    }
    public void setId(int id)
    {
        this.id = id;
    }
    public double getMarks()
    {
        return marks;
    }
    public void setMarks(double marks)
    {
        this.marks = marks;
    }
    @Override
    public String toString()
    {
        return id+"\\t\\t"+name+"\\t\\t"+marks;
    }
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
}

public class MainOfStudent
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Student s1 = new Student("Sagar",12,59.36);
        Student s2 = (Student)s1.clone();
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s1.hashCode());
        System.out.println(s2.hashCode());
        s1.setId(20);
        s2.setName("Luri");
        System.out.println(s1);
        System.out.println(s2);
    }
}

```

### OUTPUT :

12	Sagar	59.36
12	Sagar	59.36
366712642		
1829164700		
20	Sagar	59.36
12	Luri	59.36

### → Deep Cloning :

If the object is trying to perform cloning and there is **has-a** relationship, shallow cloning is not recommended. Hence, we have to go for deep cloning.

- Whenever we need own copy not to use default implementation we call it as deep copy, whenever we need deep copy of the object we need to implement according to our need.
- So, for deep copy we need to ensure all the member class also implement the Cloneable interface and override the clone() method of the object class.

A deep copy means actually creating a new array and copying over the values.

```
public class Address
{
    private String Address;
    public Address(String address)
    {
        super();
        Address = address;
    }
    public String getAddress()
    {
        return Address;
    }
    public void setAddress(String address)
    {
        Address = address;
    }
    @Override
    public String toString()
    {
        return Address;
    }
}

public class Student implements Cloneable
{
    private String name;
    private int id;
    private double marks;
    private Address addr; //has-A relationship

    public Student(String name, int id, double marks, Address addr)
    {
        this.name = name;
        this.id = id;
        this.marks = marks;
        this.addr = addr;
    }
}
```

```

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }

    public int getId()
    {
        return id;
    }

    public void setId(int id)
    {
        this.id = id;
    }

    public double getMarks()
    {
        return marks;
    }

    public void setMarks(double marks)
    {
        this.marks = marks;
    }

    public Address getAddr()
    {
        return addr;
    }

    public void setAddr(Address addr)
    {
        this.addr = addr;
    }

    @Override
    public String toString()
    {
        return id + "\t\t" + name + "\t\t" + marks + "\t\t" + addr;
    }

    @Override
    public Object clone() throws CloneNotSupportedException
    {
        Student s = (Student) super.clone();
        s.addr = new Address(s.getAddr().getAddress());
        return s;
    }
}

```

```

public class MainOfStudent
{
    public static void main(String[] args) throws CloneNotSupportedException
    {
        Student s1 = new Student("Sagar",12,59.25,new Address("Butwal"));
        Student s2 = (Student)s1.clone();
        System.out.println(s1);
        System.out.println(s2);
        s1.getAddr().setAddress("Nepal");
        s2.setName("Suri");
        System.out.println(s1);
        System.out.println(s2);
    }
}

```

#### OUTPUT :

12	Sagar	59.25	Butwal
12	Sagar	59.25	Butwal
12	Sagar	59.25	Nepal
12	Suri	59.25	Butwal

#### Lazy Copy :-

A lazy copy can be defined as a combination of both shallow copy and deep copy. The mechanism follows a simple approach – at the initial state, shallow copy approach is used. A counter is also used to keep a track on how many objects share the data. When the program wants to modify the original object, it checks whether the object is shared or not. If the object is shared, then the deep copy mechanism is initiated.

#### NOTE :

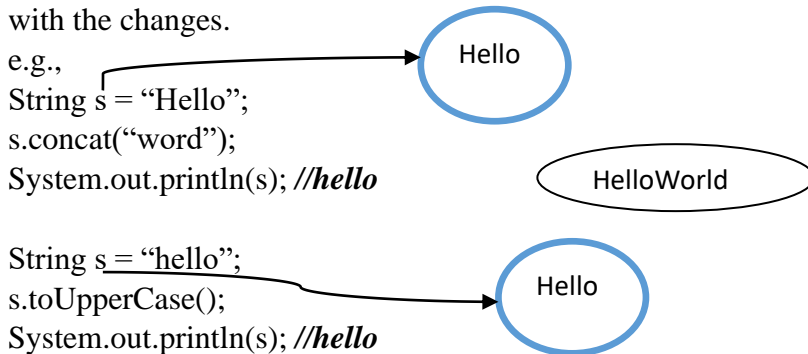
*In shallow copy, only fields of primitive data type are copied while the objects references are not copied. Deep copy involves the copy of primitive data type except String class as well as object references.*



# IMMUTABILITY :

Whenever we create an object of a class, we cannot perform any changes to the content of the object, if at all; we perform any changes to the content of the object, a new object will be created every time along with the changes. Such objects are known as **immutable objects** and the process is known as **immutability**.

String class is an immutable class that means if we create a String class object we cannot perform any changes to the content of the object. If at all, any changes are done a new object will be created along with the changes.



## NOTE :

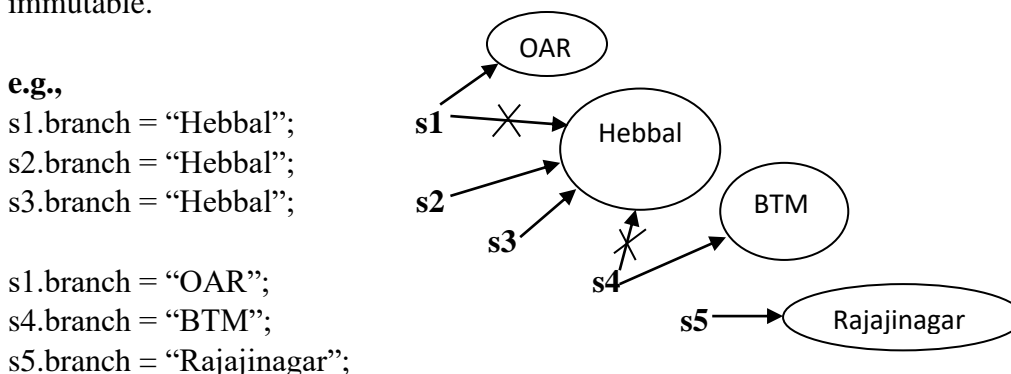
*A new object will be created only when the operation causes some changes to the content of the object.*

## Example :

```
String s = "hello";  
s.toLowerCase(); //no impact on the content of the object that is why, no new object is created.  
System.out.println(s); //hello
```

## Why are String classes immutable?

→ Whenever a String class object is referred by multiple reference variable; if at all using one reference variable, we try to perform any changes to the content of the objects, it should not affect the other reference variable pointing to the object, it is possible only if the object is immutable. Hence, String class is immutable.



→ String class immutable, if we try to change the content of the String class object with changes, a new object will be created. Even if we do minor change to the content, it will end up creating a new object, which was causing **memory leak problem**.

In order to address, this memory leak issue java developers introduced two more classes related to String class as mentioned below.

- a. **String Buffer Class**
- b. **String Builder Class**

<b>String Class</b>	<b>String Buffer Class</b>	<b>String Builder Class</b>
It is a final class available in java.lang package.	It is a final class available in java.lang package.	It is a final class available in java.lang package.
It is immutable class.	It is mutable class.	It is mutable class.
Its object can be created both with new keyword and without new keyword.	Object can be created only using <b>new</b> keyword.	Object can be created only using <b>new</b> keyword.
It overrides <b>toString()</b> , <b>hashCode()</b> & <b>equals(Object obj)</b> .	It overrides only <b>toString()</b> method of Object class.	It overrides only <b>toString()</b> method of Object class.
It implements comparable interface.	It does not implement comparable interface.	It does not implement comparable interface.
String class methods are not synchronized.	StringBuffer() are synchronized.	StringBuilder() are not synchronized.
String class is thread safe.	StringBuffer class is thread safe.	StringBuilder class is not thread safe.

String

```

1. String s = "Hello";
   s.concat("world");
   System.out.println(s); //Hello

```

## 2. StringBuffer

```

StringBuffer s = new StringBuffer("Hello");
s.append("world");
System.out.println(s); //Helloworld

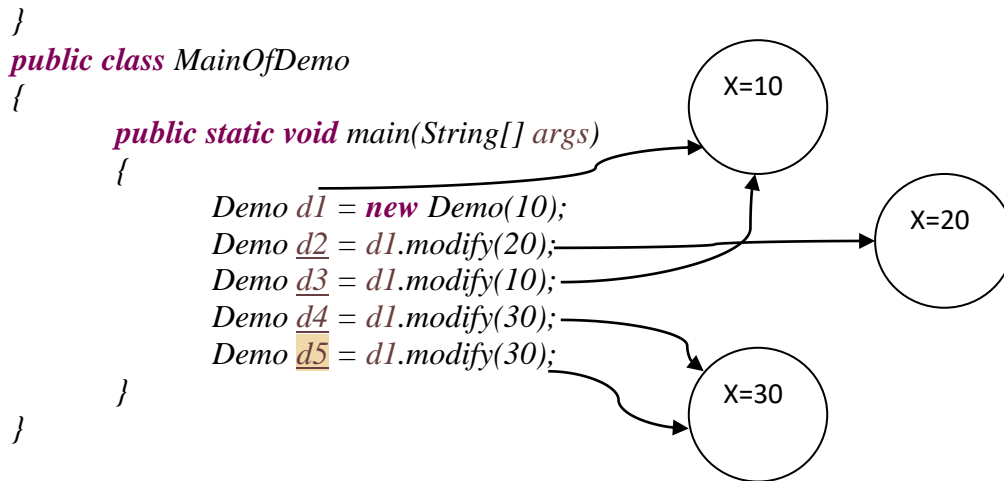
```

## Writing Own Immutable Class :

```

public class Demo
{
    int x;
    public Demo(int x) {
        this.x = x;
    }
    public Demo modify(int x)
    {
        20
        10
        30 30
        if(this.x != x)
        {
            return new Demo(x);
        }
        else
        {
            return this;
        }
    }
}

```



## FILE HANDLING :

The process of creating a file, writing content into the file and reading content from the file is known as **file handling**.

1. For creating a file, we will make use of **File** class which is available in **java.io** package.

**Example :**

```

import java.io.*;
public class Demo1 {
    public static void main(String[] args) {
        //creating a file in the current project
        File file1 = new File("Sagar.txt");
        System.out.println(file1.getName()+" created successfully!!");
        System.out.println("Location of " + file1.getName()+ " is ==> "+file1.getAbsolutePath());
        //creating a file in a user specified location
        File file2 = new File("Sunar.txt");
        System.out.println(file2.getName()+ " is created successfully!");
        System.out.println("Location of " + file2.getName()+ " is ==> "+file2.getAbsolutePath());
    }
}

```

### OUTPUT :

```

Sagar.txt created successfully!!
Location of Sagar.txt is ==> B:\My Workspace\JavaBeanProject!\Sagar.txt
Sunar.txt is created successfully!
Location of Sunar.txt is ==> B:\My Workspace\JavaBeanProject!\Sunar.txt

```

2. In order to perform write operation in a file, we will make use of **FileWriter** class which is available in **java.io** package.

**Example:**

```

import java.io.*;
public class Demo1 {
    public static void main(String[] args) throws IOException {
        //creating a file in the current project
        File file1 = new File("Sagar.txt");
        System.out.println(file1.getName()+" created successfully!!");
        FileWriter fw;
        fw = new FileWriter(file1);
        fw.write("Hello, I am Sagar Sunar!!");
        System.out.println("Write operation Done!!");
        fw.close();
    }
}

```

```
}  
}
```

3. In order to perform read operation of a file, we will make use of **FileReader** class which is available in **java.io** package.

**Example :**

```
import java.io.*;  
public class Demo1 {  
    public static void main(String[] args) throws IOException {  
        //creating a file in the current project  
        File file1 = new File("Sagar.txt");  
        System.out.println(file1.getName()+" created successfully!!");  
  
        //Writing something into a file  
        FileWriter fw;  
        fw = new FileWriter(file1);  
        fw.write("Hello, I am Sagar Sunar!!");  
        System.out.println("Write operation Done!!");  
        fw.close();  
  
        //reading something from a file  
        FileReader fr = new FileReader(file1);  
        int x;  
        System.out.println("Content of "+file1.getName());  
        while((x=fr.read())!=-1)  
        {  
            System.out.print((char)x);  
        }  
        fr.close();  
    }  
}
```

**OUTPUT :**

```
Sagar.txt created successfully!!  
Write operation Done!!  
Content of Sagar.txt  
Hello, I am Sagar Sunar!!
```

# ARRAY

## BASICS OF ARRAYS :

Collection of homogenous data is known as Array

or

Collection of values of same type is known as array.

If we would like to represent multiple values of the same type using a single variable, we will make use of arrays.

### → Creating an Array :

#### Syntax :

```
datatype[] array_name = new datatype[size]
```

e.g., `int[] a1 = new int[5];`

### → Inserting value into array

#### Syntax :

```
array_name[index] = value;
```

e.g.;

`a1[-1] = 1;`

`a1[0] = 10;`

`a1[1] = 20;`

`a1[2] = 30;`

`a1[3] = 40;`

`a1[4] = 50;`

`a1[5] = 60;`

**ArrayIndexOutOfBoundsException**

### → Pointing the value of array

#### Syntax :

```
System.out.println(arrayname[index]);
```

e.g.,

```
System.out.println(a1[0]); //10
```

```
System.out.println(a1[2]); //30
```

```
System.out.println(a1[4]); //50
```

```
System.out.println(a1[6]); //Exception
```

## Interview Question :

→ Create an int array for user specified size, insert the value into it by taking input from user and display only the even values.

*import java.util.Scanner;*

```
public class DemoArray {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter size of an array");  
        int size = sc.nextInt();  
        int[] arr1 = new int[size];  
        System.out.println("Size "+arr1.length);  
        System.out.println("Enter the value of array");  
        for(int i=0;i<=arr1.length-1;i++)  
        {
```

```

        arr1[i] = sc.nextInt();
    }
    System.out.println("the value of an array is");
    for(int i =0; i<=arr1.length-1;i++)
    {
        if(arr1[i]%2==0)
        {
            System.out.println(arr1[i]+ "\t");
        }
    }
}
}

```

### OUTPUT :

Enter size of an array

4

Size 4

Enter the value of array

20

60

40

12

the value of an array is

20

60

40

12

# Create an int array for the user specified size, insert different values into it and display only the odd values.

**import** java.util.Scanner;

```

public class Demo1 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter no. of integer values to be entered:");
        System.out.println("=====");
        int size = sc.nextInt();
        int[] arr1 = new int[size];
        System.out.println("\nSize of array is "+arr1.length);
        System.out.println("\nPlease enter the values:");
        System.out.println("*****");
        for(int i=0;i<=arr1.length-1;i++)
        {
            arr1[i] = sc.nextInt();
        }
        System.out.println("\nThe odd values are :");
        for(int i=0;i<=arr1.length-1;i++)
        {
            if((arr1[i]%2)!=0)
            {
                System.out.println(arr1[i]+ "\t");
            }
        }
    }
}

```

```
}
```

## OUTPUT :

Enter no. of integer values to be entered:

=====

4

Size of array is 4

Please enter the values:

\*\*\*\*\*

5

6

9

16

The odd values are :

5

9

# Create a char array for the user specified size, insert different character into it and display only the vowels.

```
import java.util.Scanner;
```

```
public class Demo2 {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter no. of character values to be entered:");
        System.out.println("=====");
        int size = sc.nextInt();
        char[] arr1 = new char[size];
        System.out.println("\nSize of array is "+arr1.length);
        System.out.println("\nPlease enter the characters:");
        System.out.println("*****");
        for(int i=0;i<=arr1.length-1;i++)
        {
            arr1[i] = sc.next().charAt(0);
        }
        System.out.println("\nThe vowel letters are :");
        for(int i=0;i<=arr1.length-1;i++)
        {
            if(arr1[i]=='a'||arr1[i]=='e'||arr1[i]=='i'||arr1[i]=='o'||arr1[i]=='u')
            {
                System.out.println(arr1[i]+"\\t");
            }
        }
    }
}
```

## OUTPUT :

Enter no. of character values to be entered:

=====

4

Size of array is 4

Please enter the characters:

\*\*\*\*\*

a  
q  
g  
o

The vowel letters are :

a  
o

# Create an int array for the user specified size, insert different values into it and display the sum of all the values.

```
import java.util.Scanner;
```

```
public class Demo3 {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter no. of integer values to be entered:");  
        System.out.println("=====");  
        int size = sc.nextInt();  
        int[] arr1 = new int[size];  
        System.out.println("\nSize of array is "+arr1.length);  
        System.out.println("\nPlease enter the values:");  
        System.out.println("*****");  
        for(int i=0;i<=arr1.length-1;i++)  
        {  
            arr1[i] = sc.nextInt();  
        }  
        int sum=0;  
        for(int i=0;i<=arr1.length-1;i++)  
        {  
            sum=sum+arr1[i];  
        }  
        System.out.println("\nThe sum of all values is "+sum);  
    }  
}
```

**OUTPUT :**

Enter no. of integer values to be entered:

=====

5

Size of array is 5

Please enter the values:

\*\*\*\*\*

2  
6  
8  
4  
2

The sum of all values is 22



# Create a char array for the user specified size, insert different char into it & count the total number of uppercase character.

**import** java.util.Scanner;

```
public class Demo4 {  
    public static void main(String[] args) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Enter no. of character values to be entered:");  
        System.out.println("=====");  
        int size = sc.nextInt();  
        char[] arr1 = new char[size];  
        System.out.println("\nSize of array is "+arr1.length);  
        System.out.println("\nPlease enter the characters:");  
        System.out.println("*****");  
        for(int i=0;i<=arr1.length-1;i++)  
        {  
            arr1[i] = sc.next().charAt(0);  
        }  
        int count=0;  
        for(int i=0;i<=arr1.length-1;i++)  
        {  
            if((char)arr1[i]>=65 && (char)arr1[i]<=90)  
            {  
                count++;  
            }  
        }  
        System.out.println("\nThe total number of uppercase letters is :"+count);  
        sc.close();  
    }  
}
```

### OUTPUT :

Enter no. of character values to be entered:

=====

6

Size of array is 6

Please enter the characters:

\*\*\*\*\*

a  
C  
t  
B  
Q  
U

The total number of uppercase letters is :4

### # Storing non-primitive type into array :

Q. Create a student array for the user specified size, insert different student object into it & perform below operation:

- i. Display all the student details.
- ii. Display the name whose marks is greater than 60.

```

public class Student {
    int id;
    String name;
    double mark;
    public Student(int id, String name, double mark) {
        this.id = id;
        this.name = name;
        this.mark = mark;
    }
    @Override
    public String toString() {
        return this.id+this.name+this.mark;
    }
}

import java.util.Scanner;

public class MainClass {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter size");
        int size = sc.nextInt();
        Student[] s1 = new Student[size];
        for(int i=0;i<=s1.length-1;i++)
        {
            System.out.println("Enter Student "+(i+1)+" Details");
            System.out.println("*****");
            System.out.println("Enter id");
            int id = sc.nextInt();
            System.out.println("Enter name");
            String name = sc.next();
            System.out.println("Enter marks");
            double mark = sc.nextDouble();

            s1[i] = new Student(id,name,mark);
            System.out.println("\nStudent details inserted\n");
        }
        System.out.println("Student details are");
        System.out.println("*****");
        for(int i=0;i<=s1.length-1;i++)
        {
            System.out.println(s1[i]);
        }
        System.out.println("\nStudent whose marks are above 60");
        {
            for(int i=0;i<=s1.length-1;i++)
            {
                if(s1[i].mark>60)
                {
                    System.out.println(s1[i].name);
                }
            }
        }
    }
}

```

## OUTPUT :

Enter name

Sagar

Enter marks

59

Student details inserted

Enter Student 2 Details

\*\*\*\*\*

Enter id

699

Enter name

Bhumi

Enter marks

86

Student details inserted

Student details are

\*\*\*\*\*

3632Sagar59.0

699Bhumi86.0

Student whose marks are above 60

Bhumi

## ASSIGNMENT :

1. Create a Car array for the user specified size, insert different car objects into it and perform the below operation:

- display all the Car details
- display the car name which are manufactured after 2010
- display the car details which are black in colour.

```
package edu.jspider.Arrays.Assignment;
```

```
public class Car {
    String brand;
    String modelName;
    int year;
    String colour;
    public Car(String brand,String modelName, int year, String colour) {
        this.brand = brand;
        this.modelName = modelName;
        this.year = year;
        this.colour = colour;
    }
    @Override
    public String toString() {
        return "\nBrand = "+this.brand+"\nModel = "+this.modelName+"\nYear = "+this.year+"\nColour
            = "+this.colour;
    }
}
import java.util.Scanner;

public class MainOfCar {
```

```

public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter the number of car");
    int size = sc.nextInt();
    Car[] c1 = new Car[size];
    for(int i=0;i<=c1.length-1;i++)
    {
        System.out.println("Enter Car" + (i+1) + " Details");
        System.out.println("*****");
        System.out.println("Enter Car Brand");
        String brand = sc.next();
        System.out.println("Enter Car Model");
        String modelName = sc.next();
        System.out.println("Enter Year of Manufactured");
        int year = sc.nextInt();
        System.out.println("Enter colour");
        String colour = sc.next();

        c1[i] = new Car(brand,modelName,year,colour);
        System.out.println("\nCar details inserted!!\n");
    }
    System.out.println("Car details are:");
    System.out.println("*****");
    for(int i=0;i<=c1.length-1;i++)
    {
        System.out.println(c1[i]);
    }
    System.out.println("\n*****");
    System.out.println("Car name which are manufactured after 2010 are:");
    System.out.println("*****");
    for(int i=0;i<=c1.length-1;i++)
    {
        if(c1[i].year>2010)
        {
            System.out.println(c1[i]);
        }
    }
    System.out.println("\n*****");
    System.out.println("Car details whose colour is Black are:");
    System.out.println("*****");
    for(int i=0;i<=c1.length-1;i++)
    {
        if(c1[i].colour.equalsIgnoreCase("black"))
        {
            System.out.println(c1[i]);
        }
    }
    sc.close();
}
}

```

## OUTPUT :

```

=====
Enter the number of car
2
Enter Car1 Details
*****
Enter Car Brand
TATA
Enter Car Model
NANO

```

Enter Year of Manufactured

2007

Enter colour

WHITE

Car details inserted!!

Enter Car2 Details

\*\*\*\*\*

Enter Car Brand

HYUNDAI

Enter Car Model

CRETA

Enter Year of Manufactured

2013

Enter colour

BLACK

Car details inserted!!

Car details are:

\*\*\*\*\*

Brand = TATA

Model = NANO

Year = 2007

Colour = WHITE

Brand = HYUNDAI

Model = CRETA

Year = 2013

Colour = BLACK

\*\*\*\*\*

Car name which are manufactured after 2010 are:

\*\*\*\*\*

Brand = HYUNDAI

Model = CRETA

Year = 2013

Colour = BLACK

\*\*\*\*\*

Car details whose colour is Black are:

\*\*\*\*\*

Brand = HYUNDAI

Model = CRETA

Year = 2013

Colour = BLACK

**2. Create a Mobile array for the user specified size, insert different Mobile objects into it and perform the below operation:**

- a. display all the Mobile details
- b. display the Mobile name whose price is less than 9999.
- c. display the Mobile details which are manufactured between 2010 and 2020.
- d. display the Mobile name and price which are blue or black in colour.

```

public class Mobile {
    String brand;
    String model;
    double size;
    double price;
    int year;
    String color;
    public Mobile(String brand, String model, double size, double price, int year, String color) {
        this.brand = brand;
        this.model = model;
        this.size = size;
        this.price = price;
        this.year = year;
        this.color = color;
    }
    @Override
    public String toString() {
        return "=====\nBrand = "+brand+"\nModel = "+model+"\nSize(inch) = "+size+"\nPrice(INR) = "+price+"\nYear Of Manufacturing : "+year+"\nColour : "+color;
    }
}

```

```
import java.util.Scanner;
```

```

public class MainOfMobile {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of Mobile");
        System.out.println("*****");
        int no = sc.nextInt();
        Mobile[] m1 = new Mobile[no];
        for(int i=0;i<=m1.length-1;i++)
        {
            System.out.println("\nEnter Mobile" + (i+1) + " Details");
            System.out.println("*****");
            System.out.println("Brand Name :");
            String brand = sc.next();
            System.out.println("Model Name");
            String model = sc.next();
            System.out.println("Size(in Inches) :");
            double size = sc.nextDouble();
            System.out.println("Price in INR :");
            double price = sc.nextDouble();
            System.out.println("Year of manufacturing");
            int year = sc.nextInt();
            System.out.println("Color :");
            String color = sc.next();
            System.out.println("*****");

            m1[i] = new Mobile(brand,model,size,price,year,color);
            System.out.println("Mobile Details Inserted!!\n");
        }
        System.out.println("*****");
    }
}

```

```

System.out.println("Mobile Details are :");
for(int i=0;i<=m1.length-1;i++)
{
    System.out.println(m1[i]);
}
System.out.println("\nMobiles whose price is less than 9999 are");
for(int i=0;i<=m1.length-1;i++)
{
    if(m1[i].price<9999)
    {
        System.out.println(m1[i]);
    }
}
System.out.println("\nMobiles manufactured between 2010 and 2020 :");
for(int i=0;i<=m1.length-1;i++)
{
    if(m1[i].year>=2010 && m1[i].year<=2020)
    {
        System.out.println(m1[i]);
    }
}
System.out.println("\nMobile whose color is either black or blue:");
System.out.println("=====");
for(int i=0;i<=m1.length-1;i++)
{
    if(m1[i].color.equalsIgnoreCase("blue")||m1[i].color.equalsIgnoreCase("black"))
    {
        System.out.println(" Mobile Name : "+m1[i].brand+"
                           "+m1[i].model+"("+m1[i].color+")");
    }
}
}
}

```

## OUTPUT :

Enter the number of Mobile

\*\*\*\*\*

3

Enter Mobile1 Details

\*\*\*\*\*

Brand Name :

Oppo

Model Name

7f

Size(in Inches) :

5.1

Price in INR :

8999

Year of manufacturing

2012

Color :

Brown

\*\*\*\*\*

Mobile Details Inserted!!

### Enter Mobile2 Details

\*\*\*\*\*

Brand Name :

Iphone

Model Name

7s

Size(in Inches) :

5.6

Price in INR :

27999

Year of manufacturing

2009

Color :

Black

\*\*\*\*\*

Mobile Details Inserted!!

### Enter Mobile3 Details

\*\*\*\*\*

Brand Name :

Vivo

Model Name

z1x

Size(in Inches) :

6.2

Price in INR :

14999

Year of manufacturing

2014

Color :

Blue

\*\*\*\*\*

Mobile Details Inserted!!

\*\*\*\*\*

Mobile Details are :

=====

Brand = Oppo

Model = 7f

Size(inch) = 5.1

Price(INR) = 8999.0

Year Of Manufacturing : 2012

Colour : Brown

=====

Brand = Iphone

Model = 7s

Size(inch) = 5.6

Price(INR) = 27999.0

Year Of Manufacturing : 2009

Colour : Black

=====

Brand = Vivo

Model = z1x

Size(inch) = 6.2



Price(INR) = 14999.0  
Year Of Manufacturing : 2014  
Colour : Blue

Mobiles whose price is less than 9999 are

=====

Brand = Oppo  
Model = 7f  
Size(inch) = 5.1  
Price(INR) = 8999.0  
Year Of Manufacturing : 2012  
Colour : Brown

Mobiles manufactured between 2010 and 2020 :

=====

Brand = Oppo  
Model = 7f  
Size(inch) = 5.1  
Price(INR) = 8999.0  
Year Of Manufacturing : 2012  
Colour : Brown

=====

Brand = Vivo  
Model = z1x  
Size(inch) = 6.2  
Price(INR) = 14999.0  
Year Of Manufacturing : 2014  
Colour : Blue

Mobile whose color is either black or blue:

=====

Mobile Name : Iphone 7s(Black)  
Mobile Name : Vivo z1x(Blue)

## DRAWBACKS OF ARRAYS

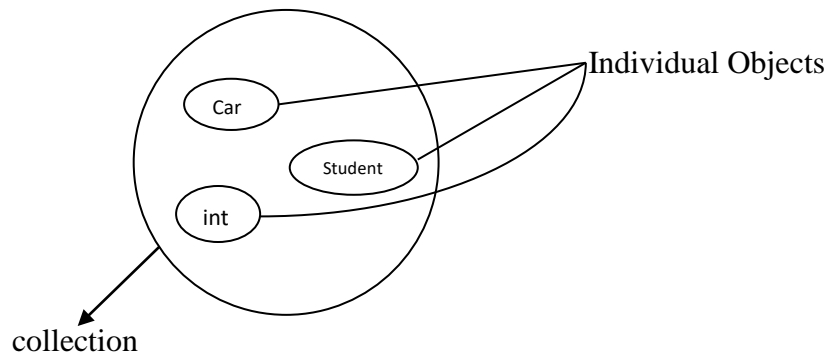
- =====
1. In an array, we can store only homogenous data(data of the same type).
  2. Arrays are fixed in size: if we go beyond the size, we will get `ArrayIndexOutOfBoundsException`.
  3. Arrays will always demand for continuous memory location.
  4. Inbuilt libraries are not present for arrays that means : for every task, the user must write the code explicitly.

**NOTE :** In order to overcome, all the above drawbacks of arrays , we make use of **collections**.



# COLLECTION

The process of combining multiple individual objects into a single entity is known as **collection**. If we want to represent group of individual objects into a single entity, we make use of collection.  
e.g.,



**Hence**, in order to overcome, all the above drawbacks of arrays , Java Developer introduced **collections**.

## Differences between arrays and collections.

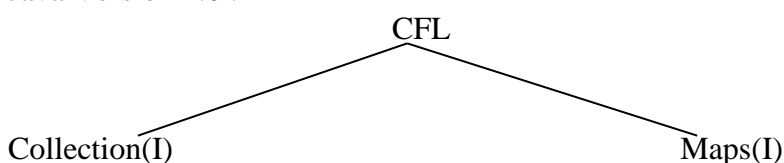
Arrays	Collection
1. It will allow only homogenous data.	1. It will allow both homogenous and heterogenous data.
2. It is fixed in size.	2. It is growable in nature.
3. It will demand for continuous memory location.	3. It will not demand for continuous memory location.
4. It allows both primitive & non-primitive data.	4. It allows only non-primitive data.(Objects)
5. It has no inbuilt methods.	5. It is rich in libraries.
6. It is better in performance(faster).	6. It is better in memory utilization.

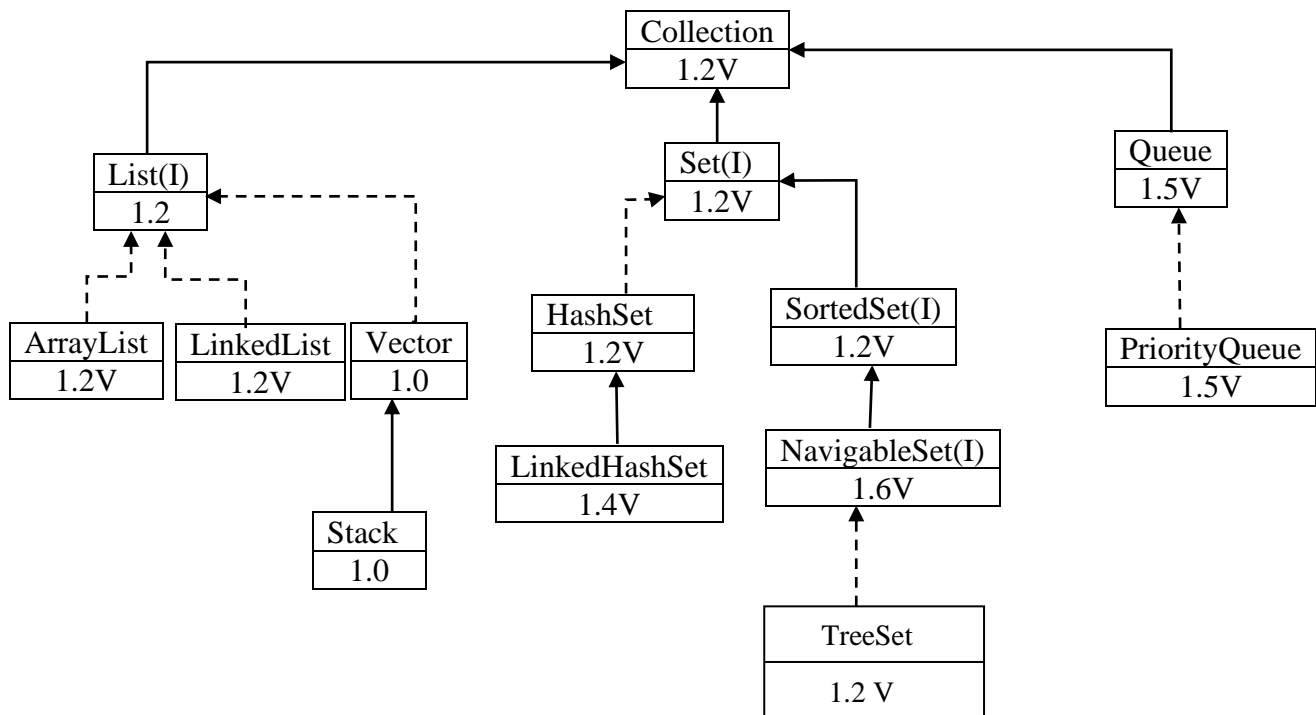
## Collection Framework Library(CFL) :

It is an inbuilt library which contains all the necessary classes and interfaces that are required to overcome the drawbacks of arrays. All the classes and interfaces of the collection framework library are present inside **java.util package**.

In C++, we call it as **Containers** where we call it as CFL.

Collection(I) & Maps(I) are root interfaces of collection framework library. Both of them were introduces in Java Version 1.0 .





## Wrapper Class :

Since we can store only objects inside a collection, wrapper classes are used to convert primitive type value to non-primitive type and non-primitive type value to primitive type.

Every primitive data type has a wrapper class association with it.

Primitive Datatype	Wrapper Class
byte	Byte.java
int	Integer.java
double	Double.java
boolean	Boolean.java
	TreeSet
	1.2V
char	Character.java
short	Short.java
long	Long.java
float	Float.java

All the wrapper classes are available in **java.lang** package.

All the wrapper classes are **final**.

Every wrapper class will override the following methods of object class:

**public String toString()**

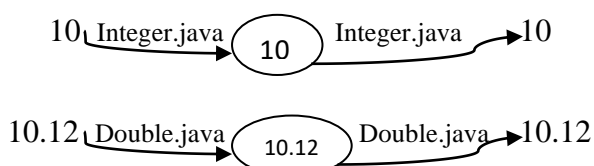
**public int hashCode()**

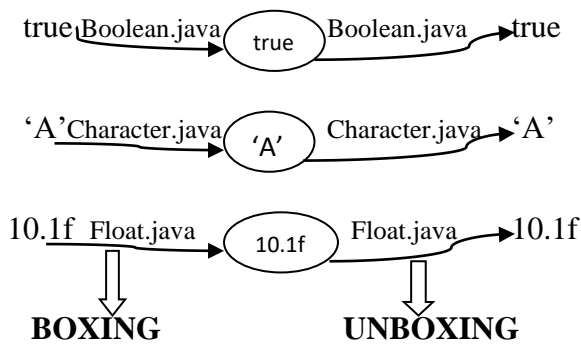
**public boolean equals(Object obj)**

Every wrapper class will implement the comparable interface.

The process of converting primitive type value to non-primitive type is known as **boxing**. Boxing can be done either implicitly by the JVM or explicitly by a user.

The process of converting non-primitive type value to primitive type is known as **unboxing**. Unboxing can be done either implicitly by the JVM or explicitly by a user.





Example :

```
package edu.jspider.collection.wrappers;
```

```
public class Demo {
    public static void main(String[] args) {
        //Auto-Boxing
        //Primitive type value given to reference variable
        Integer a = 123;
        System.out.println("a = "+a);
        Double b = 10.12;
        System.out.println("b = "+b);

        //Auto-unboxing
        //Object given to a variable
        int x = a;
        System.out.println("x = "+x);
        double y = b;
        System.out.println("y = "+y);

        //Explicit boxing
        //valueOf() is responsible for conversion from primitive to non-primitive
        Integer p = Integer.valueOf(123);
        System.out.println("p = "+p);
        Double q = Double.valueOf(10.12);
        System.out.println("q = "+q);

        //Explicit unboxing
        //intValue() will convert Integer obj into int value
        int r = p.intValue();
        System.out.println("r = "+r);
        double s = q.doubleValue();
        System.out.println("s = "+s);
    }
}
```

**OUTPUT :**

```
a = 123
b = 10.12
x = 123
y = 10.12
p = 123
q = 10.12
r = 123
s = 10.12
```

## COLLECTION(I) :-

It is one of the root interface in collection framework library which was introduced in the version 1.2. It will contain those methods which are commonly used by all the classes and interfaces under CFL.

The inbuilt methods of Collection(I) are :

```
public boolean add(Object obj)
public boolean addAll(Collection c)
public boolean contains(Object obj)
public boolean containsAll(Collection c)
public boolean removeAll(Collection c)
public boolean retainAll(Collection c)
public void clear()
public int size()
public boolean isEmpty()
public Object[] toArray()
public Iterator iterator()
```

## LIST(I) :

It is the sub-interface of collection which was introduced in the version 1.2. If we want to represent group of individual objects into a single entity where duplicate objects are allowed and insertion order is preserved, we can go for list.

In-built methods of List interface :

```
public void add(int index, Object obj)
public boolean addAll(int index, Collection c)
public Object get(int index)
public int indexOf(Object obj)
public int lastIndexOf(Object obj)
public Object set(int index, Object obj) //replace
public Object remove(Object obj)
public ListIterator listIterator()
```

## ArrayList (Implementation Class) :-

It is an implementation class of list which was introduced in the version 1.2. It will allow both homogenous object and heterogeneous objects.

- It will allow duplicate objects.
- The insertion order is preserved.
- Null Insertion is possible.
- The underlined data structure for array list is **growable Array** (Every collection class is implemented based on some data structure that means, they will follow the rules and regulation of this pre-defined data structure).
- ArrayList implements **RandomAccess(I), Cloneable(I) & Serializable(I)**.

## Constructors of ArrayList/ Different Ways of Creating ArrayList Collection :

### 1. ArrayList a1 = new ArrayList();

An ArrayList object will be created or a collection of type-ArrayList will be created with the initial capacity 10. Every time when the maximum capacity is reached, the JVM will expand the capacity of the collection automatically by using the below formula :

$$\begin{aligned}\text{New capacity} &= \text{Current Capacity} * (3/2) + 1 \\ &= 10 * (3/2) + 1\end{aligned}$$

= 16

First, a1 will point to ArrayList with capacity-10 which after breakpoint of maximum capacity of 10 will point now to ArrayList with capacity-16 and so on goes on next breakup of that maximum capacity.

## 2. ArrayList a2 = new ArrayList(int initial\_capacity) :-

A collection of type-ArrayList will be created with the initial capacity as specified by the user. Once maximum capacity is reached, the JVM will use the below formula in order to increase the capacity of collection.

New capacity = Current Capacity\*(3/2) + 1

## 3. ArrayList a3 = new ArrayList(Collection c) :-

It is used to convert other type of collection to ArrayList type.

e.g.;

```
package edu.jspiders.java.collection;
import java.util.ArrayList;
public class Demo1 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(10); //it is stored in object form through wrapper class implicitly executed by JVM.
        a1.add(10.12);
        a1.add(true);
        a1.add(20.13f); //different datatype allowed in a single array.
        a1.add('s');
        a1.add("Sagar"); //String is object itself.
        a1.add(10.12);
        a1.add(20.13f); //duplicate data allowed
        a1.add(null);

        System.out.println(a1);
    }
}
```

### OUTPUT :

[10, 10.12, true, 20.13, s, Sagar, 10.12, 20.13, null]

### Explanation for all the in-built methods :

```
package edu.jspiders.java.collection;
import java.util.ArrayList;
public class Demo1 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(10); //it is used to add individual object into the collection
        a1.add(10.12);
        a1.add('A');
        a1.add(10);

        System.out.println(a1); //[10, 10.12, A, 10]

        ArrayList a2 = new ArrayList();
        a2.add(100);
        a2.add('x');
        System.out.println(a2); //[100, x]

        //it will add one collection into another collection
        a1.addAll(a2);
    }
}
```

```

System.out.println(a1); //[10, 10.12, A, 10, 100, x]

//it checks whether object is present in collection or not
System.out.println(a1.contains(10.12)); //true
System.out.println(a2.contains(20)); //false

//to check if one collection is present in another one or not
System.out.println(a1.containsAll(a2)); //true

//to remove one collection objects from another collection
a1.removeAll(a2);
System.out.println(a1); //[10, 10.12, A, 10]

a1.remove(3);
System.out.println(a1); //[10, 10.12, A]

//remove all the objects from the collection
a2.clear();
System.out.println(a2); //[]

//to find out the number of objects in a collection
System.out.println(a1.size()); //3
System.out.println(a2.size()); //0

//used to check if a collection is empty or not
System.out.println(a1.isEmpty()); //false
System.out.println(a2.isEmpty()); //true

//to add an object at a specific index
a1.add(3, "Sagar");
System.out.println(a1); //[10, 10.12, A, Sagar]

a2.add("Sunar");
a2.add(510);
System.out.println(a2); //[Sunar, 510]

//to add one collection into another collection at specific index
a1.addAll(1, a2);
System.out.println(a1); //[10, Sunar, 510, 10.12, A, Sagar]

//to return object at the given index
System.out.println(a1.get(0)); //10
System.out.println(a1.get(4)); //A

//to return the first occurrence of the given object
System.out.println(a1.indexOf(10)); //0

//to return the last occurrence of the given object
System.out.println(a1.lastIndexOf(10)); //0

//replace
a1.set(4, "Prahlad");
System.out.println(a1); //[10, Sunar, 510, 10.12, Prahlad, Sagar]

//remove the first occurrence object

```



```

        a1.remove("Sagar");
        System.out.println(a1); //[10, Sunar, 510, 10.12, Prahlad]
    }
}

```

**Create a collection of type-ArrayList, insert different objects into it and display on the integer value.**

```

package edu.jspiders.java.collection;
import java.util.ArrayList;
public class Demo2 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(10.12);
        a1.add(111);
        a1.add(null);
        a1.add(123);

        System.out.println("Contents of a1 are :"+a1);
        System.out.println("Integer Value of a1 are");
        for(int i=0;i<=a1.size()-1;i++)
        {
            Object obj = a1.get(i);
            if(obj instanceof Integer)
            {
                System.out.println(obj);
            }
        }
    }
}

```

#### OUTPUT :

```

Contents of a1 are :[10, 10.12, 111, null, 123]
Integer Value of a1 are
10
111
123

```

**Create a collection of type-ArrayList, insert different objects into it and display on the double value.**

```

import java.util.ArrayList;
public class Demo3 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(10.12);
        a1.add(11.3f);
        a1.add(null);
        a1.add(12.5);

        System.out.println("Contents of a1 are :"+a1);
        System.out.println("Integer Value of a1 are");
        for(int i=0;i<=a1.size()-1;i++)
        {
            Object obj = a1.get(i);
            if(obj instanceof Double)
            {
                System.out.println(obj);
            }
        }
    }
}

```

```

    }
}
}

```

#### OUTPUT :

Contents of a1 are :[10, 10.12, 11.3, null, 12.5]  
 Double Value of a1 are  
 10.12  
 12.5

**Create a collection of type-ArrayList, insert different objects into it and display on the String value.**

```

import java.util.ArrayList;
public class Demo4 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add("Sag");
        a1.add(10.12);
        a1.add("Hello");
        a1.add(null);
        a1.add("A");

        System.out.println("Contents of a1 are :"+a1);
        System.out.println("Integer Value of a1 are");
        for(int i=0;i<=a1.size()-1;i++)
        {
            Object obj = a1.get(i);
            if(obj instanceof String)
            {
                System.out.println(obj);
            }
        }
    }
}

```

#### OUTPUT :

Contents of a1 are :[Sag, 10.12, Hello, null, A]  
 String Value of a1 are  
 Sag  
 Hello  
 A

**Create a collection of the type-Arraylist, insert different object into it and perform the operation:**

- Display all the values
- Display all odd int values
- Display the sum of all int values
- Display all the char value
- Display all String value that ends with 'i'
- Display all the double value between 10.1 and 20.1

```

import java.util.ArrayList;
public class Assignment1 {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(10);
        a1.add(10.12);
    }
}

```

```

a1.add(323);
a1.add('d');
a1.add(57);
a1.add("Luri");
a1.add(16.5);
a1.add(85.3);
a1.add('l');
a1.add("Bhumika");
System.out.println("All the values entered are :");
System.out.println("=====");
for(int i=0; i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    System.out.println(obj);
}
System.out.println("\nAll odd int values are:");
for(int i=0; i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    if(obj instanceof Integer)
    {
        Integer itr = (Integer)obj;
        if(itr%2!=0)
        {
            System.out.println(itr);
        }
    }
}
int sum=0;
for(int i=0; i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    if(obj instanceof Integer)
    {
        Integer itr = (Integer)obj;
        sum = sum+itr;
    }
}
System.out.println("\nThe sum of integer value is "+sum);

System.out.println("\nAll char values are:");
for(int i=0; i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    if(obj instanceof Character)
    {
        System.out.println(obj);
    }
}

System.out.println("\nString value which ends with 'i':");
for(int i=0; i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    if(obj instanceof String)

```

```

        {
            String s =(String)obj;
            if(s.endsWith("i"))
            {
                System.out.println(s);
            }
        }
    }
    System.out.println("\nDouble value between 10.1 and 20.1:");
    for(int i=0; i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Double)
        {
            Double d=(Double)obj;
            if(d>=10.1 && d<=20.1)
            {
                System.out.println(d);
            }
        }
    }
}

```

#### OUTPUT :

All the values entered are :

```

=====
10
10.12
323
d
57
Luri
16.5
85.3
l
Bhumika

```

All odd int values are:

```

323
57

```

The sum of integer value is 390

All char values are:

```

d
l

```

String value which ends with 'i':

```

Luri

```

Double value between 10.1 and 20.1:

```

10.12
16.5

```

Create a collection of type-Arraylist, insert different mobile and laptop objects into it and perform the below operation:

- Display all the mobile details.
- Display all the laptop details.
- Display all the mobile details which are manufactured before 2015.
- Display all the laptop details which are red in colour.
- Display the mobile details whose name starts with 'S'.

```
package edu.jspiders.collection.assignments;
public class Laptop {
    private String name;
    private int yom;
    private String color;
    private double price;
    public Laptop(String name, int yom, String color, double price) {
        this.name = name;
        this.yom = yom;
        this.color = color;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getYom() {
        return yom;
    }
    public void setYom(int yom) {
        this.yom = yom;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public String toString() {
        return this.name + "\t\t" + this.yom + "\t\t" + this.color +
            "\t\t" + this.price;
    }
}

import java.util.ArrayList;
public class MainOfMobileLaptop {
    public static void main(String[] args) {
        ArrayList al = new ArrayList();
        al.add(new Mobile("iphone se", 2021, "black", 34999));
    }
}
```

```

a1.add(new Mobile("samsung m20", 2011, "blue", 15499));
a1.add(new Mobile("Oneplus 6T", 2010, "red", 24999));

a1.add(new Laptop("Lenovo", 2022, "black", 36999));
a1.add(new Laptop("HP", 2011, "red", 56999));
a1.add(new Laptop("Macbook", 2019, "Golden", 96999));

System.out.println("All Mobile details are");
System.out.println("=====");
System.out.println("Name\t\tYOM\t\tColor\t\tPrice");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj=a1.get(i);
    if(obj instanceof Mobile)
    {
        System.out.println(obj);
    }
}

System.out.println("\nAll Laptop details are");
System.out.println("=====");
System.out.println("Name\t\tYOM\t\tColor\t\tPrice");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj=a1.get(i);
    if(obj instanceof Laptop)
    {
        System.out.println(obj);
    }
}

System.out.println("\nMobile details manufactured before 2015");
System.out.println("=====");
System.out.println("Name\t\tYOM\t\tColor\t\tPrice");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj=a1.get(i);
    if(obj instanceof Mobile)
    {
        Mobile m = (Mobile)obj;
        if(m.getYom()<2015)
        {
            System.out.println(obj);
        }
    }
}

System.out.println("\nLaptop details which is red in colour");
System.out.println("=====");
System.out.println("Name\t\tYOM\t\tColor\t\tPrice");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)

```

```

{
    Object obj=a1.get(i);
    if(obj instanceof Laptop)
    {
        Laptop l =(Laptop)obj;
        if(l.getColor().equalsIgnoreCase("red"))
        {
            System.out.println(obj);
        }
    }
}
System.out.println("\nMobile details whose name starts with 'S'");
System.out.println("=====");
System.out.println("Name\t\tYOM\t\tColor\t\tPrice");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj=a1.get(i);
    if(obj instanceof Mobile)
    {
        Mobile m1 = (Mobile)obj;
        if(m1.getName().startsWith("s"))
        {
            System.out.println(obj);
        }
    }
}
}
}

```

## OUTPUT :

All Mobile details are

Name	YOM	Color	Price
iphone se	34999.0	black	2021
samsung m20	15499.0	blue	2011
Oneplus 6T	24999.0	red	2010

All Laptop details are

Name	YOM	Color	Price
Lenovo	2022	black	36999.0
HP	2011	red	56999.0
Macbook	2019	Golden	96999.0

Mobile details manufactured before 2015

Name	YOM	Color	Price
samsung m20	15499.0	blue	2011
Oneplus 6T	24999.0	red	2010

Laptop details which is red in colour

Name	YOM	Color	Price
HP	2011	red	56999.0

Mobile details whose name starts with 'S'

Name	YOM	Color	Price
samsung m20	15499.0	blue	2011

Create a collection of type-Arraylist, insert different cricketer objects into it and perform the below operation:

1. Display all the cricketers' details.
2. Display the cricketer name whose total runs are more than 10000.
3. Display the cricketers' details whose age is between 25-35.
4. Display the cricketers' details whose total wicket is more than 300.
5. Display the cricketers' details whose name contains 'Singh' in it.

*package* edu.jspiders.collection.assignment2;

```
public class Cricketer {  
    private String name;  
    private int total_runs;  
    private int age;  
    private int total_wickets;  
    public Cricketer(String name, int total_runs, int age, int total_wickets) {  
        this.name = name;  
        this.total_runs = total_runs;  
        this.age = age;  
        this.total_wickets = total_wickets;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getTotal_runs() {  
        return total_runs;  
    }  
    public void setTotal_runs(int total_runs) {  
        this.total_runs = total_runs;  
    }  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public int getTotal_wickets() {  
        return total_wickets;  
    }  
    public void setTotal_wickets(int total_wickets) {
```



```

        this.total_wickets = total_wickets;
    }
    @Override
    public String toString() {

        return this.name + "\t\t" + this.age + "\t\t" +
            this.total_runs + "\t\t" + this.total_wickets;
    }

}

import java.util.ArrayList;

public class MainOfCricketer {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(new Cricketer("Navjot Singh", 9563, 55, 265));
        a1.add(new Cricketer("MS Dhoni", 11900, 35, 490));
        a1.add(new Cricketer("Virat Kohli", 12500, 29, 140));

        System.out.println("All Cricketer Details are:");
        System.out.println("=====");
        System.out.println("Name\t\t\tAge\t\tTotal Runs\tTotal Wickets");
        System.out.println("=====");
        for(int i=0; i<=a1.size()-1; i++)
        {
            Object obj = a1.get(i);
            System.out.println(obj);
        }
        System.out.println("\nCricketers whose age is between 25 and 35:");
        System.out.println("=====");
        for(int i=0; i<=a1.size()-1; i++)
        {
            Object obj = a1.get(i);
            Cricketer c = (Cricketer)obj;
            if(c.getAge()>=25 && c.getAge()<=35)
            {
                System.out.println(c.getName());
            }
        }

        System.out.println("\nCricketer details whose total wickets is more than 300:");
        System.out.println("=====");
        System.out.println("Name\t\t\tAge\t\tTotal Runs\tTotal Wickets");
        System.out.println("=====");
        for(int i=0; i<=a1.size()-1; i++)
        {
            Object obj = a1.get(i);
            Cricketer c = (Cricketer)obj;
            if(c.getTotal_wickets()>300)
            {
                System.out.println(obj);
            }
        }
        System.out.println("\nCricketer details whose total run is more than 10000:");
        System.out.println("=====");
    }
}

```

```

System.out.println("Name\t\tAge\t\tTotal Runs\tTotal Wickets");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    Cricketer c = (Cricketer)obj;
    if(c.getTotal_runs()>10000)
    {
        System.out.println(obj);
    }
}
System.out.println("\nCricketer details whose name contains 'Singh' :");
System.out.println("=====");
System.out.println("Name\t\tAge\t\tTotal Runs\tTotal Wickets");
System.out.println("=====");
for(int i=0;i<=a1.size()-1;i++)
{
    Object obj = a1.get(i);
    Cricketer c = (Cricketer)obj;
    if(c.getName().contains("Singh"))
    {
        System.out.println(obj);
    }
}
}
}

```

## OUTPUT :

All Cricketer Details are:

Name	Age	Total Runs	Total Wickets
Navjot Singh	55	9563	265
MS Dhoni	35	11900	490
Virat Kohli	29	12500	140

Cricketers whose age is between 25 and 35:

```

=====
MS Dhoni
Virat Kohli

```

Cricketer details whose total wickets is more than 300:

Name	Age	Total Runs	Total Wickets
MS Dhoni	35	11900	490

Cricketer details whose total run is more than 10000:

Name	Age	Total Runs	Total Wickets
MS Dhoni	35	11900	490
Virat Kohli	29	12500	140

Cricketer details whose name contains 'Singh' :

Name	Age	Total Runs	Total Wickets
Navjot Singh	55	9563	265

Create a collection of type-ArrayList, insert bike, car and truck objects into it and perform the below operation.

- Display all the bike details.
- Display all the car details.
- Display all the truck details.
- Display the bike details which are 'Black' or 'Silver' in color.
- Display the car details whose name starts with 'f' and ends with 'r'.
- Display the truck details whose mileage is more than 10.5 .

```
package edu.jspiders.collection.assignment3;
```

```
public class Car {  
    private String name;  
    private String color;  
    private double mileage;  
    public Car(String name, String color, double mileage) {  
        this.name = name;  
        this.color = color;  
        this.mileage = mileage;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public String getColor() {  
        return color;  
    }  
    public void setColor(String color) {  
        this.color = color;  
    }  
    public double getMileage() {  
        return mileage;  
    }  
    public void setMileage(double mileage) {  
        this.mileage = mileage;  
    }  
    @Override  
    public String toString() {  
        return this.name + "\t" + this.color + "\t" + this.mileage;  
    }  
}
```

[ Same type for Bike & Truck]

```
import java.util.ArrayList;  
public class MainOfAllVehicle {
```

```

public static void main(String[] args) {
    ArrayList a1 = new ArrayList();
    a1.add(new Car("Fisker", "White", 20));
    a1.add(new Car("Nissan", "Red", 25));

    a1.add(new Bike("Pulsar", "Black", 30));
    a1.add(new Bike("Honda", "Blue", 41));
    a1.add(new Bike("Apache", "Silver", 29));

    a1.add(new Truck("TATA", "Grey", 9.5));
    a1.add(new Truck("LEYLAND", "Green", 11.5));

    System.out.println("Car Details Are:");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Car)
        {
            System.out.println(obj);
        }
    }
    System.out.println("\nBike Details Are:");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Bike)
        {
            System.out.println(obj);
        }
    }
    System.out.println("\nTruck Details Are:");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Truck)
        {
            System.out.println(obj);
        }
    }
    System.out.println("\nBike Details whose color is Black or Silver:");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
    }
}

```

```

        if(obj instanceof Bike)
        {
            Bike b=(Bike)obj;
            if(b.getColor().equalsIgnoreCase("black")||b.getColor().equalsIgnoreCase("silver"))
            {
                System.out.println(obj);
            }
        }
    }
    System.out.println("\nCar Details which name starts with 'f' & ends with 'r':");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Car)
        {
            Car c = (Car)obj;
            if(c.getName().startsWith("F") && c.getName().endsWith("r"))
            {
                System.out.println(obj);
            }
        }
    }
    System.out.println("\nTruck Details which mileage is more than 10.5:");
    System.out.println("=====");
    System.out.println("Name\t\tColor\t\tMileage(km/ltr)");
    System.out.println("=====");
    for(int i=0;i<=a1.size()-1;i++)
    {
        Object obj = a1.get(i);
        if(obj instanceof Truck)
        {
            Truck t=(Truck)obj;
            if(t.getMileage(>10.5)
            {
                System.out.println(obj);
            }
        }
    }
}
}

```

## OUTPUT :

Car Details Are:

Name	Color	Mileage(km/ltr)
Fisker	White	20.0
Nissan	Red	25.0

Bike Details Are:

Name	Color	Mileage(km/ltr)
------	-------	-----------------

Pulsar	Black	30.0
Honda	Blue	41.0
Apache	Silver	29.0

Truck Details Are:

Name	Color	Mileage(km/ltr)
TATA	Grey	9.5
LEYLAND	Green	11.5

Bike Details whose color is Black or Silver:

Name	Color	Mileage(km/ltr)
Pulsar	Black	30.0
Apache	Silver	29.0

Car Details which name starts with 'f' & ends with 'r':

Name	Color	Mileage(km/ltr)
Fisker	White	20.0

Truck Details which mileage is more than 10.5:

Name	Color	Mileage(km/ltr)
LEYLAND	Green	11.5

## Generic Collection :

A collection which allows only homogenous objects into it is known as **generic collection**.

e.g.,

1. `ArrayList<Integer> a1 = new ArrayList<Integer>();`

***a1 collection will allow only Integer Object***

2. `ArrayList<Car> a2 = new ArrayList<Car>();`

***a2 collection will allow only Car object.***

3. `ArrayList<Faculty> a3 = new ArrayList<Faculty>();`

***a3 collection will allow only Faculty object.***

**Create a generic collection of type - ArrayList, insert different integer objects into it and perform the below operations :**

**a. Display all the values.**

**b. Display all the perfect number.**

***package edu.jspiders.GenericCollection.Demos;***

```
public class Operation {
    public void checkPerfect(int num)
    {
        int sum=0;
        for(int i=1;i<num;i++)
        {
```

```

        if(num%i==0)
        {
            sum=sum+i;
        }
    }
    if(sum==num)
    {
        System.out.println(num);
    }
}

import java.util.ArrayList;
public class Demo1 {
    public static void main(String[] args) {
        ArrayList<Integer> a1 = new ArrayList<Integer>();
        a1.add(12);
        a1.add(125);
        a1.add(28);
        a1.add(6);
        a1.add(121);
        a1.add(56);

        System.out.println("a1 = "+a1);

        Operation o = new Operation();
        System.out.println("\nAll the perfect numbers are:");
        for(int i=0;i<=a1.size()-1;i++)
        {
            int a=a1.get(i);
            o.checkPerfect(a);
        }
    }
}

```

#### OUTPUT :

a1 = [12, 125, 28, 6, 121, 56]

All the perfect numbers are:

28

6

**Create a generic collection of type-ArrayList, insert different Integer objects into it and perform the below operations:**

- a. Display all the values
- b. Display all the prime numbers
- c. Display all the number which are palindrome

```

public class Demo2 {
    public static void main(String[] args) {
        ArrayList<Integer> a1 = new ArrayList<Integer>();
        a1.add(17);
        a1.add(32523);
        a1.add(11);
        a1.add(13);
        a1.add(121);
        a1.add(56);
        a1.add(21);
    }
}

```

```

        System.out.println("a1 = "+a1);
        OperationDemo2 od = new OperationDemo2();
        System.out.println("\nAll the palindrome numbers are:");
        for(int i=0;i<=a1.size()-1;i++)
        {
            int a=a1.get(i);
            od.checkPalindrome(a);
        }

        System.out.println("\nAll the prime numbers are:");
        for(int i=0;i<=a1.size()-1;i++)
        {
            int a=a1.get(i);
            od.checkPrime(a);
        }
    }
}

import java.util.ArrayList;
public class Demo2 {
    public static void main(String[] args) {
        ArrayList<Integer> a1 = new ArrayList<Integer>();
        a1.add(17);
        a1.add(32523);
        a1.add(11);
        a1.add(13);
        a1.add(121);
        a1.add(56);
        a1.add(21);

        System.out.println("a1 = "+a1);
        OperationDemo2 od = new OperationDemo2();
        System.out.println("\nAll the palindrome numbers are:");
        for(int i=0;i<=a1.size()-1;i++)
        {
            int a=a1.get(i);
            od.checkPalindrome(a);
        }

        System.out.println("\nAll the prime numbers are:");
        for(int i=0;i<=a1.size()-1;i++)
        {
            int a=a1.get(i);
            od.checkPrime(a);
        }
    }
}

```

#### OUTPUT :

a1 = [17, 32523, 11, 13, 121, 56, 21]

All the palindrome numbers are:

32523

11

121



All the prime numbers are:

17

11

13

Create a generic collection of type-ArrayList, insert different Character objects into it and perform the below operations:

- a. Display all the values
- b. Display all the vowels
- c. Display all the lowercase character
- d. Count all the uppercase alphabets

**package** edu.jspider.generic.arraylist;

**import** java.util.ArrayList;

**public class** Demo1 {

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

        ArrayList<Character> a1 = **new** ArrayList<Character>();

        a1.add('a');

        a1.add('G');

        a1.add('o');

        a1.add('l');

        a1.add('J');

        a1.add('k');

        System.out.println("a1 = "+a1);

        System.out.println("\nVowel characters are:");

**for** (**int** i = 0; i < a1.size()-1; i++) {

**if** (a1.get(i)=='a' || a1.get(i)=='e' || a1.get(i)=='i' || a1.get(i)=='o' || a1.get(i)=='u') {

                System.out.println(a1.get(i));

            }

        }

        System.out.println("\nUppercase letters are:");

**for** (**int** i = 0; i < a1.size()-1; i++) {

**if** (a1.get(i).isUpperCase(a1.get(i))) {

                System.out.println(a1.get(i));

            }

        }

        System.out.println("\nLowercase letters are:");

**for** (**int** i = 0; i < a1.size()-1; i++) {

**if** (a1.get(i).isLowerCase(a1.get(i))) {

                System.out.println(a1.get(i));

            }

        }

    }

}

**OUTPUT :**

a1 = [a, G, o, l, J, k]

Vowel characters are:

a

o

Uppercase letters are:

G

J

Lowercase letters are:

a

o

l

Create a generic collection of type-ArrayList, insert different String objects into it and perform the below operations:

- Display all the values
- Display the String that starts with 's'.
- Display the String that ends with 'i'
- Display the String that contains 'a' in it.

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

```
public class Demo2 {
```

```
    public static void main(String[] args) {
        ArrayList<String> a1 = new ArrayList<String>();
        a1.add("Hi");
        a1.add("I");
        a1.add("am");
        a1.add("Sagar");
        a1.add("Suri");
        a1.add(".");

        System.out.println("a1 = "+a1);
        System.out.println("\nString that starts with 's': ");
        System.out.println("*****");
        for (int i = 0; i < a1.size()-1; i++) {
            if (a1.get(i).startsWith("s")||a1.get(i).startsWith("S")) {
                System.out.println(a1.get(i));
            }
        }
        System.out.println("\nString that ends with 'i': ");
        System.out.println("*****");
        for (int i = 0; i < a1.size()-1; i++) {
            if (a1.get(i).endsWith("i")||a1.get(i).endsWith("I")) {
                System.out.println(a1.get(i));
            }
        }
        System.out.println("\nString that contains 'a': ");
        System.out.println("*****");
        for (int i = 0; i < a1.size()-1; i++) {
            if (a1.get(i).contains("a")) {
                System.out.println(a1.get(i));
            }
        }
    }
}
```

**OUTPUT :**

a1 = [Hi, I, am, Sagar, Suri, .]

String that starts with 's':

\*\*\*\*\*

Sagar

Suri

String that ends with 'i':

\*\*\*\*\*

Hi

I

Suri

String that contains 'a':

\*\*\*\*\*

am

Sagar

### → When to go for ArrayList?

The underlined data structure for ArrayList is growable array that means: just like arrays, In ArrayList also, the object will be stored in continuous memory location. Whenever the frequent operation of the user is retrievable of data or fetching the data, they can go for ArrayList.

### LinkedList (Implementation Class) :

- It is the implementation class of list which was introduced in the version 1.2 .
- The underlined data structure for LinkedList is **doubly LinkedList**.
- It allows both homogenous objects and heterogenous object.
- It allows duplicate values.
- Insertion order is preserved.
- Null insertion is possible.
- It implements Serializable(I) and Clonnable(I) interface but not RandomAccess(I).

### Constructors of LinkedList/ Different ways of creating a collection of type-LinkedList

1. LinkedList l1 = new LinkedList();

An empty LinkedList object or a collection of type LinkedList will be created.

*import java.util.\*;*

*public class Demo1 {*

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

*LinkedList l1 = new LinkedList<>();*

*l1.add(10);*

*l1.add('a');*

*l1.add(12.1);*

*System.out.println("l1 = "+l1);*

*l1.add(1, 'x');*

*System.out.println("l1 = "+l1);*

*}*

*}*

### OUTPUT :

l1 = [10, a, 12.1]

l1 = [10, x, a, 12.1]

## 2. LinkedList l2 = new LinkedList(Collection c);

It is used to convert other type of collections to LinkedList type.

### When to go for LinkedList?

→ Whenever the frequent operation of the user is **insertion & deletion**, it is recommended to go for LinkedList.

**NOTE :** *LinkedList is exactly same like ArrayList except the following:*

ArrayList	LinkedList
<i>The underlined data structure is growable array.</i>	<i>The underlined data structure is doubly LinkedList.</i>
<i>It implements RandomAccess(I) interface.</i>	<i>It does not implement RandomAccess(I) interface.</i>
<i>Objects are stored in continuous memory location.</i>	<i>Objects are stored in non-continuous memory location but internally, they are linked with each other as shown in the above example.</i>
<i>Retrieval Operation is faster.</i>	<i>Retrieval operation is slower than ArrayList. However, insertion and deletion operations are faster.</i>

### Collections(Utility Class) :

It is a utility class which will contain predefined utility methods that are used to perform operations on an existing collection.

The collection class is available in **java.util package**. All the utility methods present inside the collections class is **static method**. They should be called using the class name.

The collections class is applicable only the generic collection.

#### Example :-

```
package edu.jspiders.collection.demos;
```

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

```
public class Demo1 {
```

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

```
        ArrayList<Integer> a1 = new ArrayList<Integer>();
```

```
        a1.add(10);
```

```
        a1.add(1);
```

```
        a1.add(5);
```

```
        a1.add(100);
```

```
        a1.add(87);
```

```
        a1.add(45);
```

```
        a1.add(23);
```

```
        a1.add(34);
```

```
        a1.add(1);
```

```
        System.out.println("a1 = "+a1);
```

```
        Collections.sort(a1); //sort the collection in ascending order
```

```
        System.out.println("Sorted value of a1:"+a1);
```

```
        System.out.println("Min. value = "+Collections.min(a1));
```

```
        //return minimum value of collection
```

```
        System.out.println("Max. value = "+Collections.max(a1));
```

```
        //return maximum value of collection
```

```
        //shuffle all the objects present in the collection
```

```
        Collections.shuffle(a1);
```

```
        System.out.println("a1 after shuffle : "+a1);
```

```

//reverse the objects present in collection
Collections.reverse(a1);
System.out.println("a1 after reverse : "+a1);

//to perform the replace operation
Collections.replaceAll(a1, 1, 11);
System.out.println("a1 = "+a1);

```

```

}
}

```

#### OUTPUT :

```

a1 = [10, 1, 5, 100, 87, 45, 23, 34, 1]
Sorted value of a1:[1, 1, 5, 10, 23, 34, 45, 87, 100]
Min. value = 1
Max. value = 100
a1 after shuffle : [34, 1, 100, 1, 10, 23, 87, 5, 45]
a1 after reverse : [45, 5, 87, 23, 10, 1, 100, 1, 34]
a1 = [45, 5, 87, 23, 10, 11, 100, 11, 34]

```

**Create a generic collection of type-LinkedList, store double objects into it and display all the values present in the collection in descending order:**

```

package edu.jspiders.collection.demos;
import java.util.*;
public class Demo2 {
    public static void main(String[] args) {
        LinkedList<Double> d1 = new LinkedList<Double>();
        d1.add(10.12);
        d1.add(5.65);
        d1.add(7.89);
        d1.add(15.12);
        d1.add(75.65);
        d1.add(87.89);
        d1.add(122.12);
        d1.add(7.89);
        d1.add(7.89);

        System.out.println("d1 before sorting");
        System.out.println("=====");
        System.out.println(d1);

        Collections.sort(d1);
        System.out.println("\nd1 after sorting in ascending order");
        System.out.println(d1);

        Collections.reverse(d1);
        System.out.println("\nd1 after sorting in descending order");
        System.out.println(d1);
    }
}

```

#### OUTPUT :

```

d1 before sorting
=====
[10.12, 5.65, 7.89, 15.12, 75.65, 87.89, 122.12, 7.89, 7.89]

d1 after sorting in ascending order
[5.65, 7.89, 7.89, 10.12, 15.12, 75.65, 87.89, 122.12]

```

d1 after sorting in descending order  
[122.12, 87.89, 75.65, 15.12, 10.12, 7.89, 7.89, 7.89, 5.65]

### **Important Note :**

*If a user wants to sort a collection which contains custom object inside it, the below two mandatory rules must be followed :*

- 1) The collection must be generic collection(Homogenous Object).*
- 2) The class whose object we are inserting into a collection must implement **Comparable(I)**.*

### **Comparable(I) :**

- ✓ It is a functional interface(only one abstract method).
- ✓ It is present in **java.lang package**.
- ✓ It is responsible for sorting the collection.

Syntax:

```
interface Comparable
{
    int compareTo(Object obj);
}
```

### **Working of compareTo(Object obj) :**

```
public int compareTo(Object obj)
{
    return CurrentObject.attribute-GivenObject.attribute; //ascending order sorting
    or
    return GivenObject.attribute-CurrentObject.attribute; //descending order sorting
}
```

### **example:**

```
package edu.jspiders.collection.compareTo;
public class Bag implements Comparable
{
    private String name;
    private String color;
    private double price;
    private int yom;
    public Bag(String name, String color, double price, int yom) {
        this.name = name;
        this.color = color;
        this.price = price;
        this.yom = yom;
    }
    @Override
    public String toString() {
        return this.name + "\t\t" + this.color + "\t\t" + this.price + "\t\t" + this.yom;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

```

    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public int getYom() {
        return yom;
    }
    public void setYom(int yom) {
        this.yom = yom;
    }
    @Override
    public int compareTo(Object o) {
        Bag b =(Bag)o;
        return (int)(this.price-b.price);
    }
}

import java.util.LinkedList;
public class MainOfBag {
    public static void main(String[] args) {
        LinkedList<Bag> a1 = new LinkedList<Bag>();
        a1.add(new Bag("Artifox", "Black", 989, 2010));
        a1.add(new Bag("SkyBag", "Blue", 1459, 2017));
        a1.add(new Bag("Aristo", "Red", 1299, 2004));
        a1.add(new Bag("Vip", "Black", 1299, 2022));
        a1.add(new Bag("Data", "Brown", 1999, 2019));
        a1.add(new Bag("Adidas", "black", 2499, 2005));

        System.out.println("Name\t\tColor\t\tPrice\t\tYoM");
        System.out.println("=====");
        for (int i = 0; i < a1.size(); i++) {
            System.out.println(a1.get(i));
        }
        System.out.println("Details according to increasing price:");
        Collections.sort(a1);

        System.out.println("Name\t\tColor\t\tPrice\t\tYoM");
        System.out.println("=====");
        for (int i = 0; i < a1.size(); i++) {
            System.out.println(a1.get(i));
        }
    }
}

```

**OUTPUT :**

Name	Color	Price	YoM
Artifox	Black	989.0	2010
SkyBag	Blue	1459.0	2017
Aristo	Red	1299.0	2004
Vip	Black	1299.0	2022
Data	Brown	1999.0	2019
Adidas	black	2499.0	2005

Details according to increasing price:

Name	Color	Price	YoM
Artifox	Black	989.0	2010
Aristo	Red	1299.0	2004
Vip	Black	1299.0	2022
SkyBag	Blue	1459.0	2017
Data	Brown	1999.0	2019
Adidas	black	2499.0	2005

Create a generic collection of type-ArrayList, insert mobile objects into it and perform below operation:

- Display all the mobile details in increasing order of their price.
- Display all the mobile details whose price is between 15000 and 20000.
- Display all the mobile details in decreasing order of their name.

```

package edu.jspiders.collection.demos;
import java.lang.*;
public class Mobile implements Comparable{
    private String name;
    private String color;
    private double price;
    private double size;
    public Mobile(String name, String color, double price, double size) {
        this.name = name;
        this.color = color;
        this.price = price;
        this.size = size;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public double getPrice() {
        return price;
    }
}

```



```

    public void setPrice(double price) {
        this.price = price;
    }
    public double getSize() {
        return size;
    }
    public void setSize(double size) {
        this.size = size;
    }
    @Override
    public String toString() {
        return this.name + "\t\t" + this.color + "\t\t" + this.price + "\t\t" + this.size;
    }
    @Override
    public int compareTo(Object obj){
        Mobile m = (Mobile)obj;
        return (int)(this.price-m.price);
    }
}

import java.util.ArrayList;
import java.util.Collections;

public class MainOfMobile {
    public static void main(String[] args) {
        ArrayList a1 = new ArrayList();
        a1.add(new Mobile("Iphone", "Red", 44999, 6.1));
        a1.add(new Mobile("redmi", "Blue", 15999, 5.6));
        a1.add(new Mobile("Mi", "Black", 19999, 6.1));

        System.out.println("Name\t\tColour\t\tPrice\t\tSize(inches)");
        System.out.println("=====");
        for (int i = 0; i < a1.size(); i++) {
            System.out.println(a1.get(i));
        }

        Collections.sort(a1);
        System.out.println("\nMobile details according to their increasing price");
        System.out.println("=====");
        System.out.println("Name\t\tColour\t\tPrice\t\tSize(inches)");
        System.out.println("=====");
        for (int i = 0; i < a1.size(); i++) {
            System.out.println(a1.get(i));
        }

        System.out.println("\nMobile details whose price is between 15000-20000");
        System.out.println("=====");
        System.out.println("Name\t\tColour\t\tPrice\t\tSize(inches)");
        System.out.println("=====");
        for (int i = 0; i < a1.size(); i++) {
            Object o = a1.get(i);
            Mobile m = (Mobile)o;
            if (m.getPrice() >= 15000 && m.getPrice() <= 20000) {
                System.out.println(o);
            }
        }
    }
}

```

```

    }
}
}

```

### OUTPUT :

Name	Colour	Price	Size(inches)
Iphone	Red	44999.0	6.1
redmi	Blue	15999.0	5.6
Mi	Black	19999.0	6.1

Mobile details according to their increasing price

Name	Colour	Price	Size(inches)
redmi	Blue	15999.0	5.6
Mi	Black	19999.0	6.1
Iphone	Red	44999.0	6.1

Mobile details whose price is between 15000-20000

Name	Colour	Price	Size(inches)
redmi	Blue	15999.0	5.6
Mi	Black	19999.0	6.1

Create a generic collection of type-LinkedList, insert book objects into it and display all the book details in decreasing order of their year of publication.

**import** edu.jspiders.Iterator.Assignment1.Shoes;

```

public class Book implements Comparable {
    private String name;
    private String author;
    private int yom;
    public Book(String name, String author, int yom) {
        this.name = name;
        this.author = author;
        this.yom = yom;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getAuthor() {
        return author;
    }
    public void setAuthor(String author) {
        this.author = author;
    }
    public int getYom() {
        return yom;
    }
}

```

```

    public void setYom(int yom) {
        this.yom = yom;
    }
    @Override
    public String toString() {
        return this.name + "\t\t" + this.author + "\t\t" + this.yom;
    }
    public int compareTo(Object o) {
        Book b = (Book)o;
        return (int)(this.yom - b.yom);
    }
}
import java.util.*;
public class MainOfBook {
    public static void main(String[] args) {
        LinkedList ll = new LinkedList();
        ll.add(new Book("Alchemy", "Coehlo", 1999));
        ll.add(new Book("Archer", "Rowling", 2002));
        ll.add(new Book("Riot", "Tharoor", 2006));
        ll.add(new Book("KGB", "Mitrokhin", 1991));

        Collections.sort(ll);
        System.out.println("Book details according to year of publication");
        System.out.println("=====");
        System.out.println("Name\t\tAuthor\t\tYear_Of_Publication");
        System.out.println("=====");
        for (int i = 1; i < ll.size(); i++) {
            System.out.println(ll.get(i));
        }
    }
}

```

#### OUTPUT :

Book details according to year of publication

=====		
Name	Author	Year_Of_Publication
=====		
Alchemy	Coehlo	1999
Archer	Rowling	2002
Riot	Tharoor	2006

#### NOTE :

We can not compare two string using '-' operator. In order to compare two String values, we make use of compareTo(String s) which is an in-built method of String class.

```

public int compareTo(Object obj)
{
    return CurrentObject.compareTo(GivenObject.attribute); //ascending order sorting
    or
    return GivenObject.attribute.compareTo(CurrentObject.attribute); //descending order sorting
}

```

*Collections class(Utility class) can be used only for List implementation classes(ArrayList, LinkedList, Vector and Stack)*

## Set(I) :

It is a sub-interface of collection interface which was introduced in version 1.2 .

If we want to represent group of individual objects into a single entity where duplicate objects are not allowed and insertion order is not preserved. Hence, we go for Set(I).

List(I)	Set(I)
It is introduced in 1.2 version.	It is introduced in 1.2 version
Duplicate objects are allowed.	Duplicate objects are not allowed.
Insertion order is preserved.	Insertion order is not preserved.

## HashSet(Implementation Class) :

It is an implementation class of **Set(I)** which was introduced in version 1.2 .

- It allowed both homogenous objects and heterogenous objects.
- Duplicate objects are not allowed.
- Insertion order is not preserved.
- Null insertion is possible.
- The underlined data structure for HashSet is **HashTable** .
- It implements **Cloneable(I)** and **Serializable(I)** but not **RandomAccess(I)**.

## Constructors of HashSet/ Different ways of creating a collection of type-HashSet

### 1. HashSet h1 = new HashSet();

An empty HashSet object or a collection of type-HashSet will be created with the initial capacity 16 and fill ratio = 0.75 or 75 %.(Once 75% of the collection is occupied, the JVM will expand the capacity of the collection.)

### 2. HashSet h2 = new HashSet(int initial\_capacity);

A collection of type-HashSet will be created with the initial capacity as specified by a user and with the fill ratio 75%.

### 3. HashSet h3 = new HashSet(int initial\_capacity, float fill\_ratio);

A collection of type-HashSet will be created with the initial capacity and fill ratio as specified by a user.

### 4. HashSet h4 = new HashSet(Collection c);

It is used to convert other type of collection to HashSet type.

Example;

```
package edu.jspiders.hashset.demos;
```

```
import java.util.HashSet;
```

```
public class Demo1 {  
    public static void main(String[] args) {  
        HashSet h = new HashSet();  
        h.add(10);  
        h.add(10.12);  
        h.add('a');  
        h.add("Dinga");  
        h.add(null);  
        h.add(10);  
        h.add(10);  
        h.add(10.12);  
    }  
}
```

```

        System.out.println("h = "+h);
    }
}

```

#### OUTPUT :

h = [null, a, 10, Dinga, 10.12]

Create a generic collection of type-HashSet, insert Integer objects into it and display all the values in descending order. *(Very very Important)*

*package* edu.jspiders.hashset.demos;

*import* java.util.\*;

```

public class Demo2 {
    public static void main(String[] args) {
        HashSet<Integer> h = new HashSet();
        h.add(10);
        h.add(10);
        h.add(52);
        h.add(125);
        h.add(101);
        h.add(190);
        h.add(121);

        System.out.println("h = "+h);

        ArrayList a1 = new ArrayList(h);
        /* Since Collections class methods can be used only for List implementation classes,
        that is why we are converting HashSet
        Collection to ArrayList collection.
        */

        Collections.sort(a1); // sorting the objects present in the collection in ascending order.
        System.out.println("a1 = "+a1);

        //In descending order
        Collections.reverse(a1);
        System.out.println("a1 = "+a1);
    }
}

```

#### OUTPUT :

h = [52, 101, 121, 10, 125, 190]

a1 = [10, 52, 101, 121, 125, 190]

a1 = [190, 125, 121, 101, 52, 10]

#### LinkedHashSet (Sub-class) :

It is a sub-class of HashSet which was introduced in the version 1.4 .

- LinkedHashSet is exactly same like HashSet except the below features:
  - The underlined data structure for LinkedHashSet is **DoublyLinkedList** & **HashTable**.
  - Insertion order is preserved.

## Differences between HashSet and LinkedHashSet.

HashSet	LinkedHashSet
It is an implementation class.	It is a sub-class of HashSet.
Underlined data structure is HashTable.	Underlined data structure is DoublyLinkedList & HashTable.
Insertion order is not preserved.	Insertion order is preserved.

### Example :

```
package edu.jspiders.hashset.demos;
import java.util.*;
public class Demo3 {
    public static void main(String[] args) {
        LinkedHashSet h = new LinkedHashSet();
        h.add(10);
        h.add(10);
        h.add(52);
        h.add(50.32);
        h.add("a");
        h.add(null);
        h.add(true);
        h.add("Hello");

        System.out.println("h = "+h);
    }
}
```

### OUTPUT :

h = [10, 52, 50.32, a, null, true, Hello]

### Important Note :

Whenever a frequent operation of user is **searching of data**, they can make use of either **HashSet** or **LinkedHashSet**.

### NOTE :

**public Object get(int index)** is present in List(I). That is why, it is applicable only for List(I) implementation classes (ArrayList, LinkedList, Vector and Stack).

### Cursors :

It will do the same task as that of **get(int index)** that is to fetch the object one by one from a given collection.

Since **get(int index)** can be used only for List(I) implementation classes (ArrayList, LinkedList, Vector and Stack) that is why, in order to fetch the object one by one from **HashSet/LinkedHashSet/Trees/**

**PriorityQueue** Collection, we make use of **Cursors**.

It will work exactly same like **get(int index)** of List(I).

**\*\*Every cursor in java is a Interface\*\***

### Types of Cursor :

#### a. Enumeration(I) :

- ✓ It is also known as Legacy cursor.
- ✓ It can be used only for legacy classes (Stack and Vector).
- ✓ Using Enumeration(I) cursor, we can perform only read operation.

### **b. Iterator(I)\*\*\***

- ✓ It is known as Universal cursor.
- ✓ It can be used for any collection class (ArrayList, LinkedList, Vector, Stack, HashSet, LinkedHashSet, TreeSet and PriorityQueue).
- ✓ Using Iterator(I) cursor, we can perform read and remove operation.

### **c. ListIterator(I) :**

- ✓ It is also known as the powerful cursor.
- ✓ It can be used only for List implementation classes (ArrayList, LinkedList, Stack and Vector)
- ✓ Using ListIterator(I) cursor, we can perform read, remove and replace operation.

### **Iterator(I) :**

Syntax :

```
package java.util.*;
interface Iterator
{
    Object next();
    boolean hasNext();
    Object remove();
}
```

### **public Iterator iterator() :**

- ✓ It is a helper method which will return the implementation class object of **Iterator(I)**.
- ✓ In java, every Collection class will have a private inner class which implements **Iterator(I)**. Since, private inner class object cannot be created directly, that is why, we make use of this **helper** method in order to get the object of the class which implements the iterator interface.
- ✓ The return type of the helper method is interface type that means it can return its implementation class object.

Structure/Example :

```
public class AL/LL/S/V/HS/LHS/TS/PQ
{
    =====
    =====
    =====
    =====
    public Iterator iterator()
    {
        return new xyz();
    }
    private class xyz implements Iterator
    {
        public Object next()
        {
            -----
        }
        public boolean hasNext()
        {
            -----
        }
        public Object remove()
        {
```

```

    }
    }
}

```

Iterator itr = c.iterator(); *helper method to which returns the implementation class object of Iterator(I)*  
*Interface type reference variable to achieve loose coupling*  
 collectionType(AL/LL/V/S/HS/LHS/TS/PQ)

#### Note :

**next()** :- It will return the obj to which it is pointing to. (initial value=0).

**hasNext()** :- It will check if **next()** is pointing to some object. If yes, it returns true else false. (initial value = -1)

### Working of Iterator Cursor :

h1 → [10,10.12, "hello", 'a', null, 123]

```
Iterator itr = h1.iterator();
```

```

while(itr.hasNext())
{
    Object obj = itr.next();
    if(obj instanceof Integer)
    {
        sop(obj);
    }
}

```

Create a collection of type-HashSet, insert different objects into it and perform the below operation:

- Display all the values
- Display the int values which are between 100 & 200.
- Display the String value that ends with 'r'.
- Display the Double value which are less than 10.1 .

```

package edu.jspiders.hashset.iterator;
import java.util.*;
public class Demo1 {
    public static void main(String[] args) {
        HashSet h1 = new HashSet();
        h1.add(10);
        h1.add(10.12);
        h1.add('a');
        h1.add(null);
        h1.add(101);
        h1.add(9.1);
        h1.add("sagar");

        System.out.println("h1 = "+h1);
    }
}

```



```

System.out.println("\nint value b/w 100 & 200");
System.out.println("=====");
Iterator itr = h1.iterator();
while (itr.hasNext()) {
    Object obj = itr.next(); //upcasting
    if (obj instanceof Integer) {
        Integer i = (Integer)obj; //downcasting
        if (i>=100 && i<=200) {
            System.out.println(i);
        }
    }
}
System.out.println("\nString value ending with 'r'");
System.out.println("=====");
Iterator itr1 = h1.iterator();
while (itr1.hasNext()) {
    Object obj = itr1.next(); //upcasting
    if (obj instanceof String) {
        String i = (String)obj; //downcasting
        if (i.endsWith("r")) {
            System.out.println(i);
        }
    }
}
System.out.println("\nDouble value less than 10.1");
System.out.println("=====");
Iterator itr2 = h1.iterator();
while (itr2.hasNext()) {
    Object obj = itr2.next(); //upcasting
    if (obj instanceof Double) {
        Double i = (Double)obj; //downcasting
        if (i<10.1) {
            System.out.println(i);
        }
    }
}
}
}
}

```

## OUTPUT :

h1 = [null, a, 9.1, 101, sagar, 10, 10.12]

int value b/w 100 & 200

=====

101

String value ending with 'r'

=====

sagar

Double value less than 10.1

=====

9.1

Create a generic collection of type-LinkedHashSet, insert different shoes object into it and perform the below operation :

- i. Display all the shoes details.
- ii. Display the shoe details in increasing order of their price.
- iii. Display the shoe details which are white or black in color.
- iv. Display the shoe name whose price is less than 1000.

```
package edu.jspiders.Iterator.Assignment1;
```

```
public class Shoes implements Comparable{
    private String name;
    private double price;
    private String color;
    public Shoes(String name, double price, String color) {
        this.name = name;
        this.price = price;
        this.color = color;
    }
    @Override
    public String toString() {
        return this.name + "\t" + this.price + "\t" + this.color;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public int compareTo(Object o) {
        Shoes s =(Shoes)o;
        return (int)(this.price-s.price);
    }
}
import java.util.*;
public class MainOfShoes {
    public static void main(String[] args) {
        LinkedHashSet<Shoes> h1 = new LinkedHashSet<Shoes>();
        h1.add(new Shoes("Adidas", 1499, "black"));
        h1.add(new Shoes("Nike", 999, "blue"));
        h1.add(new Shoes("AIR", 799, "white"));

        System.out.println("Shoes Details:");
        System.out.println("=====");
    }
}
```

```

System.out.println("Name\tPrice\tColour");
System.out.println("=====");
Iterator itr = h1.iterator();
while (itr.hasNext()) {
    Object obj = itr.next(); //upcasting
    System.out.println(obj);
}

LinkedList a1 = new LinkedList(h1);
Collections.sort(a1);
System.out.println("\nShoes Details sorting according to price");
System.out.println("=====");
System.out.println("Name\tPrice\tColour");
System.out.println("=====");
Iterator itr3 = a1.iterator();
while (itr3.hasNext()) {
    Object obj = itr3.next(); //upcasting
    Shoes s = (Shoes)obj;
    System.out.println(s);
}

System.out.println("\nShoes Details which are black or white:");
System.out.println("=====");
System.out.println("Name\tPrice\tColour");
System.out.println("=====");
Iterator itr1 = h1.iterator();
while (itr1.hasNext()) {
    Object obj = itr1.next(); //upcasting
    Shoes s = (Shoes)obj;
    if (s.getColor().equalsIgnoreCase("Blue") ||
        s.getColor().equalsIgnoreCase("black"))
    {
        System.out.println(obj);
    }
}

System.out.println("\nShoes Details which price is less than 1000");
System.out.println("=====");
System.out.println("Name\tPrice\tColour");
System.out.println("=====");
Iterator itr2 = h1.iterator();
while (itr2.hasNext()) {
    Object obj = itr2.next(); //upcasting
    Shoes s = (Shoes)obj;
    if (s.getPrice() < 1000)
    {
        System.out.println(obj);
    }
}
}
}

```

## OUTPUT :

Shoes Details:

Name	Price	Colour
Adidas	1499.0	black
Nike	999.0	blue
AIR	799.0	white

Shoes Details sorting according to price

Name	Price	Colour
AIR	799.0	white
Nike	999.0	blue
Adidas	1499.0	black

Shoes Details which are black or white:

Name	Price	Colour
Adidas	1499.0	black
Nike	999.0	blue

Shoes Details which price is less than 1000

Name	Price	Colour
Nike	999.0	blue
AIR	799.0	white

## Sorted Set(I):

It is the sub-interface of set which was introduced in the version 1.2 .

If we want to represent group of individual objects into a single entity where the objects are stored in some sorting order then we can make use of SortedSet();

## TreeSet:

- It is an implementation class of Navigable Set which was introduced in the version 1.2 .
- The underlined data structure for TreeSet is **Binary Search Tree** .
- It allows only homogenous object.
- Null insertion is not allowed.
- Duplicate objects are not allowed.
- Insertion order is not preserved because the objects will be stored in some sorting order.
- It implements **Clonnable(I)** & **Serializable(I)** but not **RandomAccess(I)**.

In a TreeSet collection, we can store only those class objects which are eligible for sorting, an object is said to be eligible for sorting only if it implements **Comparator(I)** or **Comparable(I)**.

If a class implements Comparable(I), it is known as **Default Natural Sorting Order(DNSO)**.

If a class implements Comparator(I), it is known as **Customized Sorting Order(CSO)**.

## Constructors of TreeSet/Different ways of creating a collection of type-TreeSet.

TreeSet t1 = new TreeSet(); //DNSO

which ever class object we add in t1 collection must implement **Comparable(I)**.

TreeSet t2 = new TreeSet(Comparator c); //CSO

which ever class object we add in t2 collection must implement **Comparator(I)** and the class object should be passed as an argument to the constructor.

e.g.;

-----

TreeSet t2 = new TreeSet(new Student());

in t2, we can add Student objects and Student class must implement **Comparator(I)**.

TreeSet t1 = new TreeSet(Collection c);

it is used to convert other type of collection to TreeSet type.

### Example :

**package** edu.jspider.TreeSet.Demos;

**import** java.util.TreeSet;

**public class** Demo1 {

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

        TreeSet<Integer> t1 = **new** TreeSet<Integer>();

        t1.add(100);

        t1.add(100);

        t1.add(99);

        t1.add(78);

        t1.add(89);

        t1.add(1);

        t1.add(67);

        System.out.println("t1 = "+t1);

    }

}

### OUTPUT :

t1 = [1, 67, 78, 89, 99, 100]

### in STANDARD IT METHOD :

**package** edu.jspider.TreeSet.Demos;

**import** java.util.TreeSet;

**public class** Operations {

**public** TreeSet<Integer> createCollection()

    {

        TreeSet t1 = **new** TreeSet<Integer>();

        t1.add(123);

        t1.add(132);

**return** t1;

    }

**public void** Display(TreeSet<Integer> t1)

    {

        System.out.println(t1);

    }

}

```

import java.util.TreeSet;
public class Demo2 {
    public static void main(String[] args) {
        Operations o = new Operations();
        TreeSet<Integer> t1 = o.createCollection();
        o.Display(t1);
    }
}

```

**OUTPUT:**

[123, 132]

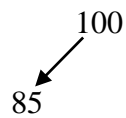
### Working of compareTo(Object obj) :

Example :-

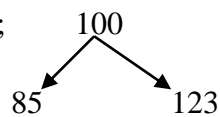
t1 = [100, 85, 123, 79, 119, 82, 126];

1) 100

2) 85.compareTo(100);

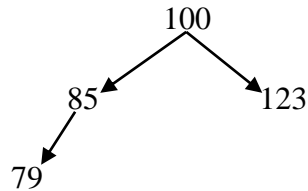


3) 123.compareTo(100);



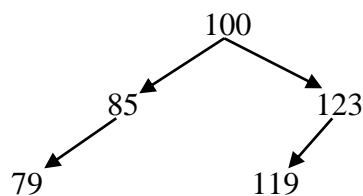
4) 79.compareTo(100);

79.compareTo(85);



5) 119.compareTo(100);

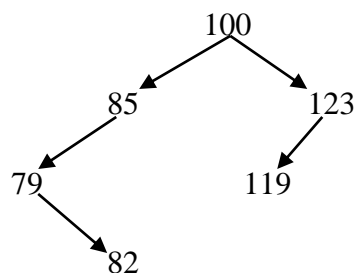
119.compareTo(123);



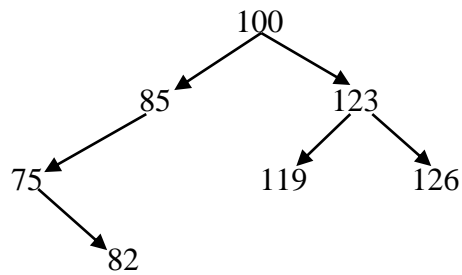
6) 82.compareTo(100);

82.compareTo(85);

82.compareTo(79);



7) 126.compareTo(100);  
126.compareTo(123);



### When to go for TreeSet?

→ Whenever a user wants to store all the data in some **sorting order**, the user can make use of **TreeSet**.

### Queue(I) :

It is a sub-interface of collection which was introduced in the version 1.5. If we want to represent group of individual objects into a single entity where the objects must be processed in **FIFO**(First-In First-Out), we go for queue.

### PriorityQueue (Implementation class) :

It is an implementation class of queue which was introduced in the version 1.5 .

- The underlined data structure for priority queue is queue.
- It allows only homogenous objects.
- Null insertion is not allowed.
- Duplication objects are allowed.
- Insertion order is not preserved.
- It implements **Clonable(I)**, **Serializable(I)** but not **RandomAccess(I)**.

### Constructors of PriorityQueue / Different ways of creating a collection of type- PriorityQueue:

```
PriorityQueue p1 = new PriorityQueue();
```

An empty collection of type-PriorityQueue will be created and whichever class object, we are adding in p1, must implement Comparable(I).

```
PriorityQueue p1 = new PriorityQueue(int initial_capacity, Comparator c);
```

An empty collection of type-PriorityQueue with user specified size will be created and whichever class object, we are adding in p1, must implement Comparator(I).

```
PriorityQueue p1 = new PriorityQueue(Collection c);
```

It is used to convert other type of collection to PriorityQueue type.

### Important Note :

*In PriorityQueue collection, the objects will be stored in some random order but the processing of the details from the collection will happen in sorting order.*

### Example ;

```
package edu.jspider.PriorityQueue.Demos;  
import java.util.PriorityQueue;  
  
public class Demo1 {
```

```

public static void main(String[] args) {
    PriorityQueue<Integer> p1 = new PriorityQueue<Integer>();
    p1.add(100);
    p1.add(121);
    p1.add(78);
    p1.add(45);
    p1.add(1);
    p1.add(11);

    System.out.println("p1 = "+p1); //p1 = [1, 45, 11, 121, 78, 100]

    p1.remove(); //an inbuilt method of PriorityQueue which will remove the first object

    System.out.println("p1 = "+p1); //p1 = [11, 45, 100, 121, 78]

    p1.remove();
    System.out.println("p1 = "+p1); //p1 = [45, 78, 100, 121]

    p1.remove();

    System.out.println("p1 = "+p1); //p1 = [78, 121, 100]

}
}

```

Comparable	Comparator
1) Comparable provides a <b>single sorting sequence</b> . In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides <b>multiple sorting sequences</b> . In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable <b>affects the original class</b> , i.e., the actual class is modified.	Comparator <b>doesn't affect the original class</b> , i.e., the actual class is not modified.
3) Comparable provides <b>compareTo() method</b> to sort elements.	Comparator provides <b>compare() method</b> to sort elements.
4) Comparable is present in <b>java.lang</b> package.	A Comparator is present in the <b>java.util</b> package.
5) We can sort the list elements of Comparable type by <b>Collections.sort(List)</b> method.	We can sort the list elements of Comparator type by <b>Collections.sort(List, Comparator)</b> method.



## COLLECTION COMPARISON

	Homogenous Object	Heterogenous Object	Duplicate	Insertion Order Preserved	Null Insertion	Underlined Data Structure	Implements Clonable(I)	Implements Serializable(I)	Implements RandomAccess(I)	When Recommended
<b>ArrayList</b>	Yes	Yes	Yes	Yes	Yes	Growable Array	Yes	Yes	Yes	Retrieval
<b>LinkedList</b>	Yes	Yes	Yes	Yes	Yes	Doubly LinkedList	Yes	Yes	No	Insertion or Deletion
<b>HashSet</b>	Yes	Yes	No	No	Yes	HashTable	Yes	Yes	No	Searching
<b>LinkedHashSet</b>	Yes	Yes	No	Yes	Yes	DoublyLink edList & HashTable	Yes	Yes	No	Searching
<b>TreeSet</b>	Yes	No	No	No	No	Binary Search Tree	Yes	Yes	No	Sorting
<b>PriorityQueue</b>	Yes	No	Yes	No	No	Queue	Yes	Yes	No	FIFO



# MULTI-THREADING

A thread is a lightweight sub-process, the smallest unit of processing. It is a separate path of execution. Multiprocessing and multithreading, both are used to achieve multitasking.

**Multithreading in Java** is a process of executing multiple threads simultaneously. We use multithreading than multiprocessing because threads use a shared memory area.

They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process which helps to attain maximum CPU utilization.

Java Multithreading is mostly used in games, animation, etc.

## 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.

## Java Thread class

Java provides **Thread class** to achieve thread programming. Thread class provides **constructors** and methods to create and perform operations on a thread. Thread class extends **Object class** and implements Runnable interface.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

## Thread creation by extending the Thread class :

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}
```

```

    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}

```

### Output :

```

Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running

```

## Thread creation by implementing the Runnable Interface :

We create a new class which implements java.lang.Runnable interface and override run() method. then we instantiate a Thread object and call start() method on this object.

```

// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

```

```

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {

```

```
Thread object
    = new Thread(new MultithreadingDemo());
object.start();
}
}
}
```

**Output :**

Thread 13 is running  
Thread 11 is running  
Thread 12 is running  
Thread 15 is running  
Thread 14 is running  
Thread 18 is running  
Thread 17 is running  
Thread 16 is running

**Thread Class vs Runnable Interface :**

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.





