

Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- A Java library example is, [Comparator Interface](#). If a class implements this interface, then it can be used to sort a collection.

Syntax :

```
interface <interface_name> {  
  
    // declare constant fields  
  
    // declare methods that abstract  
  
    // by default.  
  
}
```

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Why do we use interface ?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

```
// A simple interface

interface Player

{

    final int id = 10;

    int move();

}
```

To implement an interface we use keyword: implements

```
// Java program to demonstrate working of
// interface.

import java.io.*;

// A simple interface

interface In1

{

    // public, static and final

    final int a = 10;

    // public and abstract

    void display();

}
```

```

// A class that implements the interface.

class TestClass implements In1
{
    // Implementing the capabilities of
    // interface.

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

    // Driver Code

    public static void main (String[] args)
    {
        TestClass t = new TestClass();

        t.display();

        System.out.println(a);
    }
}

```

Output:

Geek

10

A real-world example:

Let's consider the example of vehicles like bicycle, car, bike....., they

have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, caretc implement all these functionalities in their own class in their own way.

```
import java.io.*;

interface Vehicle {

    // all are the abstract methods.

    void changeGear(int a);

    void speedUp(int a);

    void applyBrakes(int a);

}

class Bicycle implements Vehicle{

    int speed;

    int gear;

    // to change gear

    @Override

    public void changeGear(int newGear) {
```

```
        gear = newGear;

    }

    // to increase speed

    @Override

    public void speedUp(int increment){

        speed = speed + increment;

    }

    // to decrease speed

    @Override

    public void applyBrakes(int decrement){

        speed = speed - decrement;

    }

    public void printStates() {

        System.out.println("speed: " + speed

            + " gear: " + gear);

    }
```

```
}
```

```
class Bike implements Vehicle {
```

```
    int speed;
```

```
    int gear;
```

```
    // to change gear
```

```
    @Override
```

```
    public void changeGear(int newGear) {
```

```
        gear = newGear;
```

```
    }
```

```
    // to increase speed
```

```
    @Override
```

```
    public void speedUp(int increment) {
```

```
        speed = speed + increment;
```

```
    }
```

```
// to decrease speed

@Override

public void applyBrakes(int decrement){

    speed = speed - decrement;

}

public void printStates() {

    System.out.println("speed: " + speed

        + " gear: " + gear);

}

}

class GFG {

    public static void main (String[] args) {

        // creating an inatance of Bicycle

        // doing some operations

        Bicycle bicycle = new Bicycle();

        bicycle.changeGear(2);
```

```

        bicycle.speedUp(3);

        bicycle.applyBrakes(1);

        System.out.println("Bicycle present state :");

        bicycle.printStates();

    }

    // creating instance of the bike.

    Bike bike = new Bike();

    bike.changeGear(1);

    bike.speedUp(4);

    bike.applyBrakes(3);

    System.out.println("Bike present state :");

    bike.printStates();

}

}

```

Output;

Bicycle present state :

speed: 2 gear: 2

Bike present state :

speed: 1 gear: 1

New features added in interfaces in JDK 8

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

```
// An example to show that interfaces can
```

```
// have methods from JDK 1.8 onwards
```

```
interface In1
```

```
{
```

```
    final int a = 10;
```

```
    default void display()
```

```
    {
```

```
        System.out.println("hello");
```

```
    }
```

```
}
```

```
// A class that implements the interface.
```

```
class TestClass implements In1
```

```
{
```

```
    // Driver Code
```

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

```
    {
```

```
        TestClass t = new TestClass();
```

```
        t.display();  
    }  
  
}
```

Output :

hello

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

```
// An example to show that interfaces can  
  
// have methods from JDK 1.8 onwards  
  
interface In1  
  
{  
  
    final int a = 10;  
  
    static void display()  
  
    {  
  
        System.out.println("hello");  
  
    }  
  
}
```

```
// A class that implements the interface.  
  
class TestClass implements In1  
  
{  
  
    // Driver Code
```

```

        public static void main (String[] args)

        {

            In1.display();

        }

    }

```

3. Output :

4. hello

Important points about interface or summary of article:

- We can't create instance(interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface or interfaces (more than one interface) .
- A class that implements interface must implements all the methods in interface.
- All the methods are public and abstract. And all the fields are public, static, and final.
- It is used to achieve multiple inheritance.
- It is used to achieve loose coupling.

New features added in interfaces in JDK 9

From Java 9 onwards, interfaces can contain following also

1. Static methods
2. Private methods
3. Private Static methods

Access specifier of methods in interfaces

In Java, all methods in an interface are *public* even if we do not specify *public* with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

Consider the following example, x is by default *public static final* and *foo()* is *public* even if there are no specifiers.

```
interface Test {  
  
    int x = 10;    // x is public static final and must be initialized  
    here  
  
    void foo();    // foo() is public  
  
}
```

Access specifiers for classes or interfaces in Java

In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

- 1) private (accessible within the class where defined)
- 2) default or package private (when no access specifier is specified)
- 3) protected
- 4) public (accessible from any class)

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.

- 1) public
- 2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
//filename: Main.java  
  
protected class Test {}  
  
  
  
public class Main {  
  
    public static void main(String args[]) {
```

```
}  
  
}
```

Note : Nested interfaces and classes can have all access specifiers.

Comparator Interface in Java with Examples

A comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of two different classes. Following function compare obj1 with obj2

Syntax:

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an Array/ArrayList of our own class type, containing fields like roll no, name, address, DOB, etc, and we need to sort the array based on Roll no or name?

Method 1: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criteria like Roll No. and Name.

Method 2: Using comparator interface- Comparator interface is used to order the objects of a user-defined class. This interface is present in java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using a comparator, we can sort the elements based on data members. For instance, it may be on roll no, name, age, or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

```
// To sort a given list. ComparatorClass must implement  
// Comparator interface.  
public void sort(List list, ComparatorClass c)
```

How does Collections.Sort() work?

Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0, or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for their sort.

Working Program:

Java

```
// Java program to demonstrate working of Comparator

// interface

import java.io.*;

import java.lang.*;

import java.util.*;

// A class to represent a student.

class Student {

    int rollno;

    String name, address;

    // Constructor

    public Student(int rollno, String name, String address)

    {

        this.rollno = rollno;

        this.name = name;

        this.address = address;

    }

}
```

```

    }

    // Used to print student details in main()

    public String toString()

    {

        return this.rollno + " " + this.name + " "

            + this.address;

    }

}

```

```

class Sortbyroll implements Comparator<Student> {

    // Used for sorting in ascending order of

    // roll number

    public int compare(Student a, Student b)

    {

        return a.rollno - b.rollno;

    }

}

```

```

class Sortbyname implements Comparator<Student> {

    // Used for sorting in ascending order of

```

```
// name

public int compare(Student a, Student b)

{

    return a.name.compareTo(b.name);

}

}

// Driver class

class Main {

    public static void main(String[] args)

    {

        ArrayList<Student> ar = new ArrayList<Student>();

        ar.add(new Student(111, "bbbb", "london"));

        ar.add(new Student(131, "aaaa", "nyc"));

        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");

        for (int i = 0; i < ar.size(); i++)

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

        Collections.sort(ar, new Sortbyroll());
```



```
        System.out.println("\nSorted by rollno");

        for (int i = 0; i < ar.size(); i++)

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

        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");

        for (int i = 0; i < ar.size(); i++)

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

    }

}
```

Output

Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur

Sorted by rollno

111 bbbb london

121 cccc jaipur

131 aaaa nyc

Sorted by name

131 aaaa nyc

111 bbbb london

121 cccc jaipur

By changing the return value inside the compare method, you can sort in any order that you wish to. Eg. for descending order just change the positions of 'a' and 'b' in the above compare method.

Sort collection by more than one field:

In previous articles, we have discussed how to sort the list of objects on the basis of a single field using Comparable and Comparator interface. But, what if we have a requirement to sort ArrayList objects in accordance with more than one fields like firstly, sort according to the student name and secondly, sort according to student age. Below is the implementation of the above approach:

Java

```
// Java program to demonstrate working of Comparator

// interface more than one field


import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.Iterator;

import java.util.List;


class Student {

    // instance member variables

    String Name;

    int Age;
```

```
// parameterized constructor

public Student(String Name, Integer Age)

{

    this.Name = Name;

    this.Age = Age;

}


public String getName() { return Name; }


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


public Integer getAge() { return Age; }


public void setAge(Integer Age) { this.Age = Age; }


// overriding toString() method

@Override public String toString()

{

    return "Customer{"

        + "Name=" + Name + ", Age=" + Age + '}';

}
```

```
}
```

```
static class CustomerSortingComparator
```

```
    implements Comparator<Student> {
```

```
        @Override
```

```
        public int compare(Student customer1,
```

```
                               Student customer2)
```

```
        {
```

```
            // for comparison
```

```
            int NameCompare = customer1.getName().compareTo(
```

```
                customer2.getName());
```

```
            int AgeCompare = customer1.getAge().compareTo(
```

```
                customer2.getAge());
```

```
            // 2-level comparison
```

```
            return (NameCompare == 0) ? AgeCompare
```

```
                : NameCompare;
```

```
        }
```

```
    }
```

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

{

    // create ArrayList to store Student

    List<Student> al = new ArrayList<>();

    // create customer objects using constructor

    // initialization

    Student obj1 = new Student("Ajay", 27);

    Student obj2 = new Student("Sneha", 23);

    Student obj3 = new Student("Simran", 37);

    Student obj4 = new Student("Ajay", 22);

    Student obj5 = new Student("Ajay", 29);

    Student obj6 = new Student("Sneha", 22);

    // add customer objects to ArrayList

    al.add(obj1);

    al.add(obj2);

    al.add(obj3);

    al.add(obj4);
```

```
al.add(obj5);

al.add(obj6);


// before Sorting arraylist: iterate using Iterator

Iterator<Student> custIterator = al.iterator();


System.out.println("Before Sorting:\n");

while (custIterator.hasNext()) {

    System.out.println(custIterator.next());

}


// sorting using Collections.sort(al, comparator);

Collections.sort(al,

                new CustomerSortingComparator());


// after Sorting arraylist: iterate using enhanced

// for-loop

System.out.println("\n\nAfter Sorting:\n");

for (Student customer : al) {

    System.out.println(customer);

}
```

```
}
```

```
}
```

Output

Before Sorting:

```
Customer{Name=Ajay, Age=27}  
Customer{Name=Sneha, Age=23}  
Customer{Name=Simran, Age=37}  
Customer{Name=Ajay, Age=22}  
Customer{Name=Ajay, Age=29}  
Customer{Name=Sneha, Age=22}
```

After Sorting:

```
Customer{Name=Ajay, Age=22}  
Customer{Name=Ajay, Age=27}  
Customer{Name=Ajay, Age=29}  
Customer{Name=Simran, Age=37}  
Customer{Name=Sneha, Age=22}  
Customer{Name=Sneha, Age=23}
```

Java Interface methods

There is a rule that [every member of interface is only and only public whether you define or not](#). So when we define the method of the interface in a class implementing the interface, we have to give it public access as [child class can't assign the weaker access to the methods](#).

As defined, every method present inside interface is always public and abstract whether we are declaring or not. Hence inside interface the following methods declarations are equal.

```
void methodOne();
```

```
public Void methodOne();  
abstract Void methodOne();  
public abstract Void methodOne();
```

public : To make this method available for every implementation class.

abstract : Implementation class is responsible to provide implementation.

Also, We can't use the following modifiers for interface methods.

- Private
- protected
- final
- static
- synchronized
- native
- strictfp

```
// A Simple Java program to demonstrate that
```

```
// interface methods must be public in
```

```
// implementing class
```

```
interface A
```

```
{
```

```
    void fun();
```

```
}
```

```
class B implements A
```

```
{
```

```
    // If we change public to anything else,
```

```
    // we get compiler error
```

```
    public void fun()
```

```
    {
```



```

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

    }

}

class C

{

    public static void main(String[] args)

    {

        B b = new B();

        b.fun();

    }

}

```

Output:

fun()

If we change fun() to anything other than public in class B, we get compiler error “attempting to assign weaker access privileges; was public”

Nested Interface in Java

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

Interface in a class

Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer [this](#) for details). This interface declared in a class can either be default, public, protected not private. While implementing the interface, we mention the interface as **c_name.i_name** where **c_name** is the name of the class in which it is nested and **i_name** is the name of the interface itself.

Let us have a look at the following code:-

```
// Java program to demonstrate working of
// interface inside a class.

import java.util.*;

class Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
```

```

public static void main(String[] args)

{

    Test.Yes obj;

    Testing t = new Testing();

    obj=t;

    obj.show();

}

}

```

show method of interface

The access specifier in above example is default. We can assign public, protected or private also. Below is an example of protected. In this particular example, if we change access specifier to private, we get compiler error because a derived class tries to access it.

```

// Java program to demonstrate protected

// specifier for nested interface.

import java.util.*;

class Test

{

    protected interface Yes

    {

        void show();

    }

}

```

```

class Testing implements Test.Yes

{

    public void show()

    {

        System.out.println("show method of interface");

    }

}

```

```

class A

{

    public static void main(String[] args)

    {

        Test.Yes obj;

        Testing t = new Testing();

        obj=t;

        obj.show();

    }

}

```

show method of interface

Interface in another Interface

An interface can be declared inside another interface also. We mention the interface as **i_name1.i_name2** where **i_name1** is the name of the interface

in which it is nested and **i_name2** is the name of the interface to be implemented.

```
// Java program to demonstrate working of
// interface inside another interface.

import java.util.*;

interface Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}
```

```

class A

{

    public static void main(String[] args)

    {

        Test.Yes obj;

        Testing t = new Testing();

        obj = t;

        obj.show();

    }

}

```

show method of interface

Note: In the above example, access specifier is public even if we have not written public. If we try to change access specifier of interface to anything other than public, we get compiler error. Remember, [interface members can only be public.](#)

```

// Java program to demonstrate an interface cannot
// have non-public member interface.

import java.util.*;

interface Test

{

    protected interface Yes

    {

        void show();
    }
}

```

```
    }  
}
```

```
class Testing implements Test.Yes  
{  
    public void show()  
    {  
        System.out.println("show method of interface");  
    }  
}
```

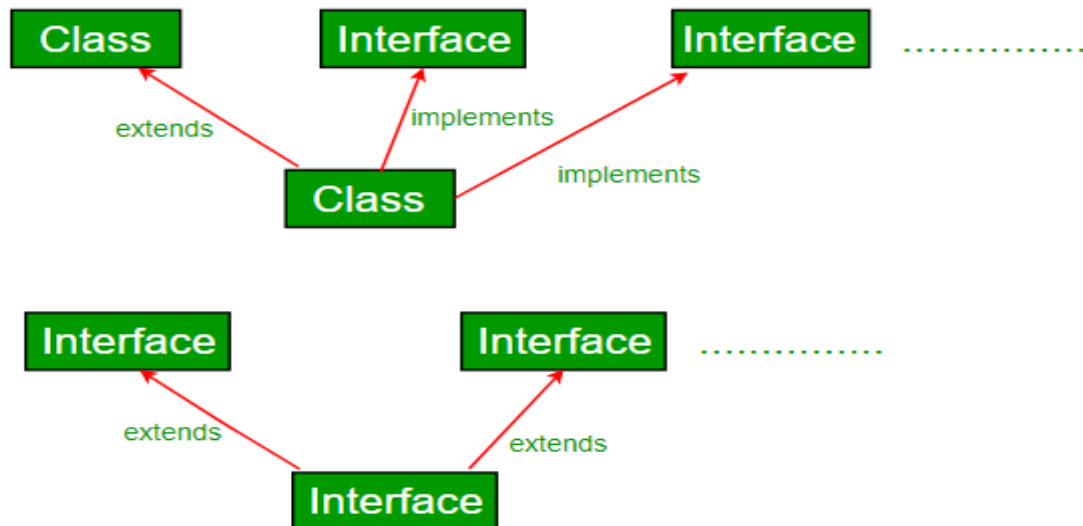
```
class A  
{  
    public static void main(String[] args)  
    {  
        Test.Yes obj;  
        Testing t = new Testing();  
        obj = t;  
        obj.show();  
    }  
}
```

illegal combination of modifiers: public and protected
protected interface Yes

Interfaces and Inheritance in Java

Prerequisites: [Interfaces in Java](#), [Java and Multiple Inheritance](#)

A class can extend another class and/ can implement one and more than one interface.



```
// Java program to demonstrate that a class can
```

```
// implement multiple interfaces
```

```
import java.io.*;
```

```
interface intfA
```

```
{
```

```
    void m1();
```

```
}
```

```
interface intfB
```

```
{
```



```
        void m2();
    }

    // class implements both interfaces
    // and provides implementation to the method.

    class sample implements intfA, intfB
    {
        @Override

        public void m1()
        {
            System.out.println("Welcome: inside the method m1");
        }

        @Override

        public void m2()
        {
            System.out.println("Welcome: inside the method m2");
        }
    }

    class GFG
```

```

{

    public static void main (String[] args)

    {

        sample ob1 = new sample();

        // calling the method implemented

        // within the class.

        ob1.m1();

        ob1.m2();

    }

}

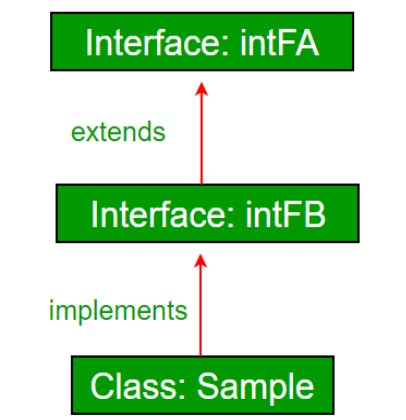
```

Output;

Welcome: inside the method m1

Welcome: inside the method m2

Interface inheritance : An Interface can extend other interface.



// Java program to demonstrate inheritance in

```
// interfaces.

import java.io.*;

interface intfA

{

    void geekName();

}


interface intfB extends intfA

{

    void geekInstitute();

}


// class implements both interfaces and provides
// implementation to the method.

class sample implements intfB

{

    @Override

    public void geekName()

    {

        System.out.println("Rohit");

    }

}
```

```

@Override

public void geekInstitute()

{

    System.out.println("JIIT");

}


public static void main (String[] args)

{

    sample ob1 = new sample();


    // calling the method implemented

    // within the class.

    ob1.geekName();

    ob1.geekInstitute();

}

}

```

Output:

Rohit
JIIT

An interface can also extend multiple interfaces.

```
// Java program to demonstrate multiple inheritance

// in interfaces

import java.io.*;

interface intfA

{

    void geekName();

}


interface intfB

{

    void geekInstitute();

}


interface intfC extends intfA, intfB

{

    void geekBranch();

}
```

```
// class implements both interfaces and provides  
  
// implementation to the method.  
  
class sample implements intfC  
{  
  
    public void geekName()  
  
    {  
  
        System.out.println("Rohit");  
  
    }  
  
  
    public void geekInstitute()  
  
    {  
  
        System.out.println("JIIT");  
  
    }  
  
  
    public void geekBranch()  
  
    {  
  
        System.out.println("CSE");  
  
    }  
  
  
    public static void main (String[] args)  
  
    {
```

```

        sample obl = new sample();

        // calling the method implemented

        // within the class.

        obl.geekName();

        obl.geekInstitute();

        obl.geekBranch();

    }

}

```

Output:

Rohit

JIIIT

CSE

Why Multiple Inheritance is not supported through a class in Java, but it can be possible through the interface?

Multiple Inheritance is not supported by class because of ambiguity. In case of interface, there is no ambiguity because implementation to the method(s) is provided by the implementing class up to Java 7. From Java 8, interfaces also have implementations of methods. So if a class implementing two or more interfaces having the same method signature with implementation, it is mandated to implement the method in class also. Refer [Java and Multiple Inheritance](#) for details.

Exceptions in Java

What is an Exception?

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

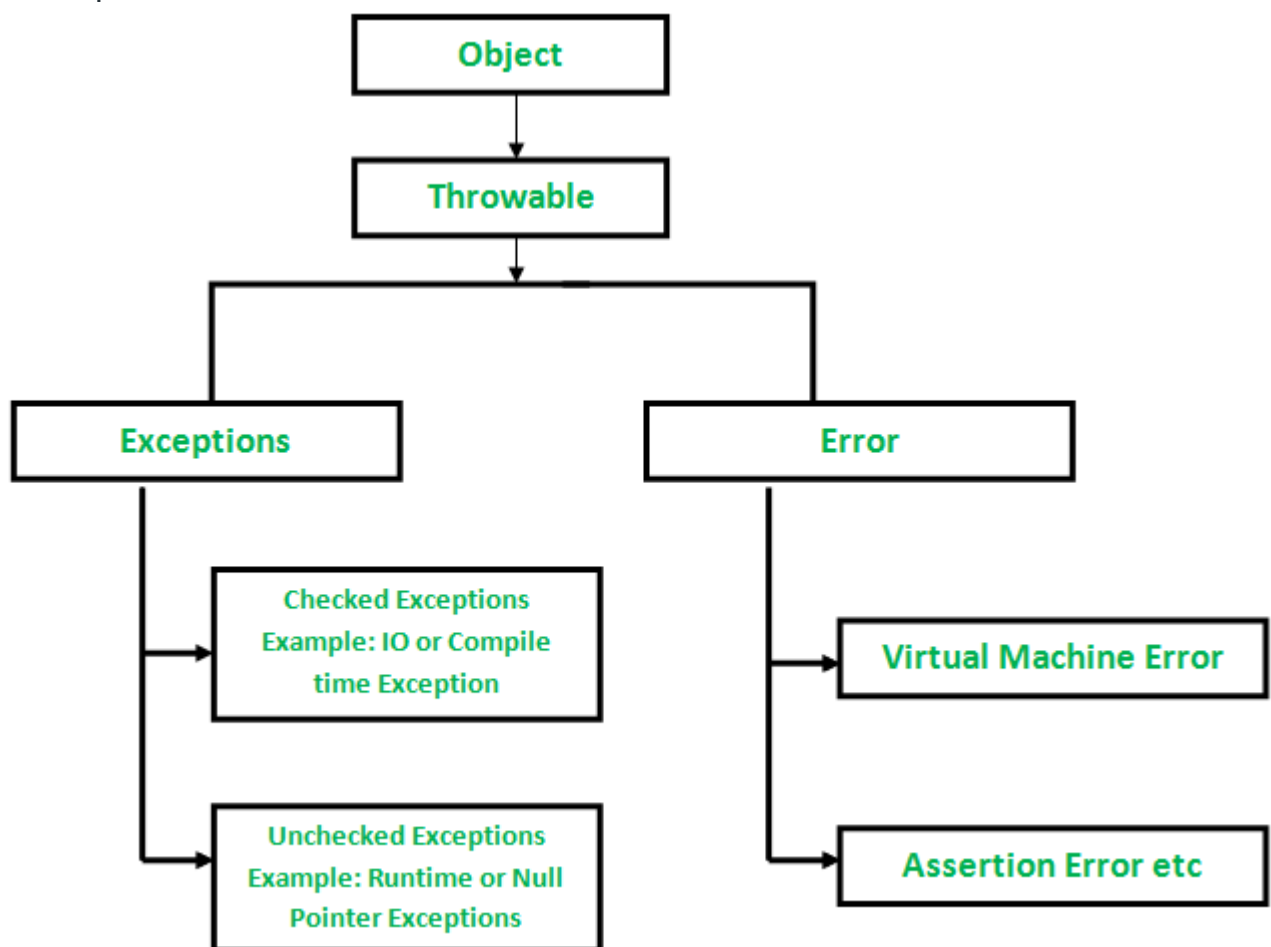
Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

All exception and errors types are sub classes of class **Throwable**, which is base class of hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. `NullPointerException` is an example of such an exception. Another branch, **Error** are used by the Java run-time system ([JVM](#)) to indicate errors having to do with the run-time environment itself (JRE). `StackOverflowError` is an example of such an error.



For checked vs unchecked exception, see [Checked vs Unchecked Exceptions](#)

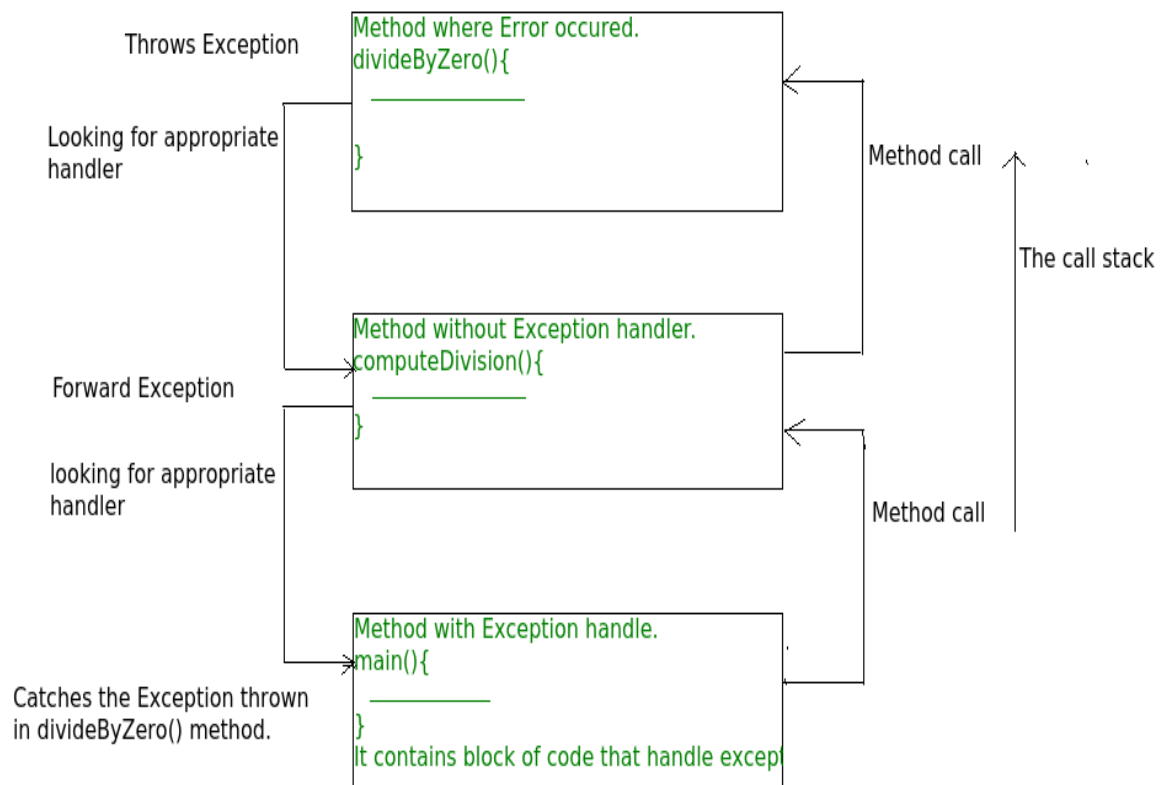
How JVM handle an Exception?

Default Exception Handling : Whenever inside a method, if an exception has occurred, the method creates an Object known as Exception Object and hands it off to the run-time system (JVM). The exception object contains name and description of the exception, and current state of the program where exception has occurred. Creating the Exception Object and handling it to the run-time system is called throwing an Exception. There might be the list

of the methods that had been called to get to the method where exception was occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

- The run-time system searches the call stack to find the method that contains block of code that can handle the occurred exception. The block of the code is called **Exception handler**.
- The run-time system starts searching from the method in which exception occurred, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occurred exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If run-time system searches all the methods on call stack and couldn't have found the appropriate handler then run-time system handover the Exception Object to **default exception handler**, which is part of run-time system. This handler prints the exception information in the following format and terminates program **abnormally**.
- Exception in thread "xxx" Name of Exception : Description
- // Call Stack

See the below diagram to understand the flow of the call stack.



The call stack and searching the call stack for exception handler.

Example :

```
// Java program to demonstrate how exception is thrown.

class ThrowsExecp{

    public static void main(String args[]){

        String str = null;

        System.out.println(str.length());

    }

}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
    at ThrowsExecp.main(File.java:8)
```

Let us see an example that illustrate how run-time system searches appropriate exception handling code on the call stack :

```
// Java program to demonstrate exception is thrown

// how the runTime system searches th call stack

// to find appropriate exception handler.

class ExceptionThrown

{

    // It throws the Exception(ArithmeticException).
```

```

// Appropriate Exception handler is not found within this method.

static int divideByZero(int a, int b){

    // this statement will cause ArithmeticException(/ by zero)

    int i = a/b;

    return i;

}

// The runTime System searches the appropriate Exception handler
// in this method also but couldn't have found. So looking forward
// on the call stack.

static int computeDivision(int a, int b) {

    int res =0;

    try

    {

        res = divideByZero(a,b);

    }

    // doesn't matches with ArithmeticException

```

```
        catch(NumberFormatException ex)

        {

            System.out.println("NumberFormatException is occurred");

        }

        return res;

    }
}
```

```
// In this method found appropriate Exception handler.
```

```
// i.e. matching catch block.
```

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

```
    int a = 1;
```

```
    int b = 0;
```

```
    try
```

```
    {
```

```
        int i = computeDivision(a,b);
```

```
    }
```

```
// matching ArithmeticException
```

```

        catch(ArithmeticException ex)

        {

            // getMessage will print description of exception(here /
by zero)

            System.out.println(ex.getMessage());

        }

    }

}

```

Output :

/ by zero.

How Programmer handles an exception?

Customized Exception Handling : Java exception handling is managed via five keywords: **try**, **catch**, [throw](#), [throws](#), and **finally**. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword [throw](#). Any exception that is thrown out of a method must be specified as such by a [throws](#) clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

Detailed Article: [Control flow in try catch finally block](#)

Need of try-catch clause(Customized Exception Handling)

Consider the following java program.

```

// java program to demonstrate

// need of try-catch clause

```

```

class GFG {

    public static void main (String[] args) {

        // array of size 4.

        int[] arr = new int[4];

        // this statement causes an exception

        int i = arr[4];

        // the following statement will never execute

        System.out.println("Hi, I want to execute");

    }

}

```

Output :

```

Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 4
    at GFG.main(GFG.java:9)

```

Explanation : In the above example an array is defined with size i.e. you can access elements only from index 0 to 3. But you trying to access the elements at index 4 (by mistake) that's why it is throwing an exception. In this case, JVM terminates the program **abnormally**. The statement *System.out.println("Hi, I want to execute");* will never execute. To execute it, we must handle the exception using try-catch. Hence to continue normal flow of the program, we need try-catch clause.

How to use try-catch clause

```

try {
    // block of code to monitor for errors
    // the code you think can raise an exception

```

```

}
catch (ExceptionType1 ex0b) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 ex0b) {
// exception handler for ExceptionType2
}
// optional
finally {
// block of code to be executed after try block ends
}

```

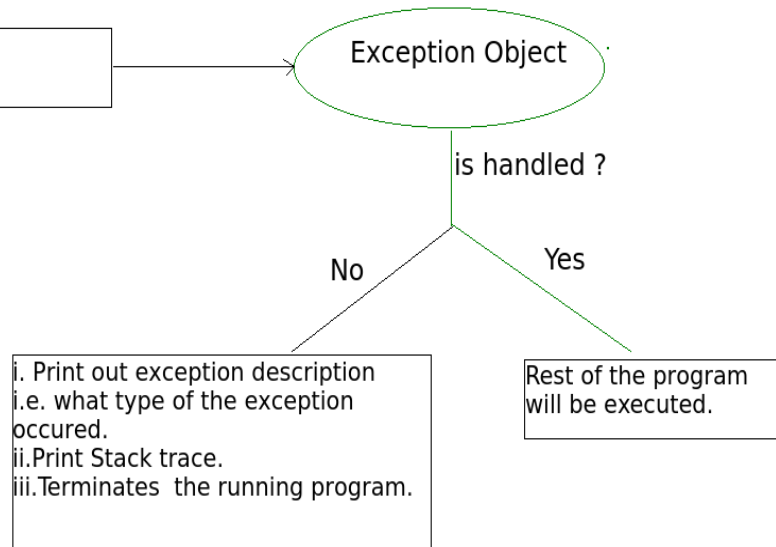
Points to remember :

- In a method, there can be more than one statements that might throw exception, So put all these statements within its own **try** block and provide separate exception handler within own **catch** block for each of them.
- If an exception occurs within the **try** block, that exception is handled by the exception handler associated with it. To associate exception handler, we must put **catch** block after it. There can be more than one exception handlers. Each **catch** block is a exception handler that handles the exception of the type indicated by its argument. The argument, `ExceptionType` declares the type of the exception that it can handle and must be the name of the class that inherits from **Throwable** class.
- For each try block there can be zero or more catch blocks, but **only one** finally block.
- The finally block is optional. It always gets executed whether an exception occurred in try block or not . If exception occurs, then it will be executed after **try and catch blocks**. And if exception does not occur then it will be executed after the **try** block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

Summary

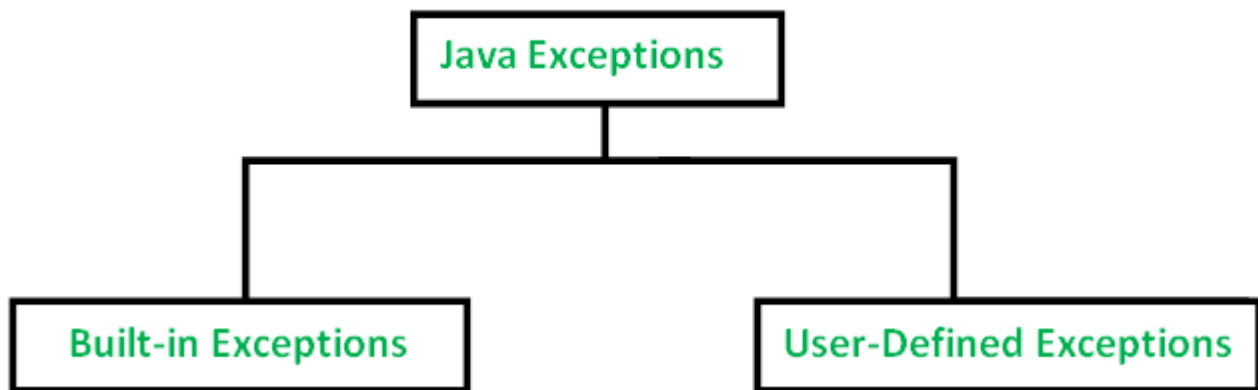
An Exception Object is created and thrown.

```
int a = 10/0;
```



Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



Built-in Exceptions

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **ArithmeticException**
It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException**
It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
This Exception is raised when a file is not accessible or does not open.
5. **IOException**
It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException**
It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
It is thrown when accessing a method which is not found.
9. **NullPointerException**
This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException**
This exception is raised when a method could not convert a string into a numeric format.

11. **RuntimeException**

This represents any exception which occurs during runtime.

12. **StringIndexOutOfBoundsException**

It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string

Examples of Built-in Exception:

- **Arithmetic exception**

- **Java**

```
// Java program to demonstrate ArithmeticException

class ArithmeticException_Demo

{

    public static void main(String args[])

    {

        try {

            int a = 30, b = 0;

            int c = a/b;    // cannot divide by zero

            System.out.println ("Result = " + c);

        }

        catch(ArithmeticException e) {

            System.out.println ("Can't divide a number by 0");

        }

    }

}
```

Output:

Can't divide a number by 0

- **NullPointerException**

- Java

```
//Java program to demonstrate NullPointerException

class NullPointerException_Demo

{

    public static void main(String args[])

    {

        try {

            String a = null; //null value

            System.out.println(a.charAt(0));

        } catch (NullPointerException e) {

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

        }

    }

}
```

Output:

NullPointerException..

- **StringIndexOutOfBoundsException**

- Java

```
// Java program to demonstrate StringIndexOutOfBoundsException

class StringIndexOutOfBounds_Demo
```

```

{

    public static void main(String args[])

    {

        try {

            String a = "This is like chipping "; // length is 22

            char c = a.charAt(24); // accessing 25th element

            System.out.println(c);

        }

        catch (StringIndexOutOfBoundsException e) {

            System.out.println("StringIndexOutOfBoundsException");

        }

    }

}

```

Output:

StringIndexOutOfBoundsException

- **FileNotFoundException**

- **Java**

```

//Java program to demonstrate FileNotFoundException

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

class File_notFound_Demo {

```

```

public static void main(String args[]) {

    try {

        // Following file does not exist

        File file = new File("E://file.txt");

        FileReader fr = new FileReader(file);

    } catch (FileNotFoundException e) {

        System.out.println("File does not exist");

    }

}
}

```

Output:

File does not exist

• NumberFormat Exception

• Java

```

// Java program to demonstrate NumberFormatException

class NumberFormat_Demo

{

    public static void main(String args[])

```

```

{

    try {

        // "akki" is not a number

        int num = Integer.parseInt ("akki") ;

        System.out.println(num) ;

    } catch (NumberFormatException e) {

        System.out.println("Number format exception");

    }

}

}

```

Output:

Number format exception

• **ArrayIndexOutOfBoundsException**

• Java

```

// Java program to demonstrate ArrayIndexOutOfBoundsException

class ArrayIndexOutOfBounds_Demo

{

    public static void main(String args[])

    {

        try{

            int a[] = new int[5];

```

```

        a[6] = 9; // accessing 7th element in an array of

                // size 5

    }

    catch (ArrayIndexOutOfBoundsException e) {

        System.out.println ("Array Index is Out Of Bounds");

    }

}

}

```

Output:

Array Index is Out Of Bounds

User-Defined Exceptions

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called 'user-defined Exceptions'.

Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter.

We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
```

```
{
    super(str);
}
```

- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

```
MyException me = new MyException("Exception details");
```

```
throw me;
```

- The following program illustrates how to create own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be apt in the account.
- If it is so, then MyException is raised and a message is displayed "Balance amount is less".

• Java

```
// Java program to demonstrate user defined exception

// This program throws an exception whenever balance

// amount is below Rs 1000

class MyException extends Exception

{

    //store account information

    private static int accno[] = {1001, 1002, 1003, 1004};

    private static String name[] =

        {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

    private static double bal[] =

        {10000.00, 12000.00, 5600.0, 999.00, 1100.55};
```



```

// default constructor

MyException() {

}

// parameterized constructor

MyException(String str) { super(str); }

// write main()

public static void main(String[] args)

{

    try {

        // display the heading for the table

        System.out.println("ACCNO" + "\t" + "CUSTOMER" +

                               "\t" + "BALANCE");

        // display the actual account information

        for (int i = 0; i < 5; i++)

        {

            System.out.println(accno[i] + "\t" + name[i] +

                               "\t" + bal[i]);

            // display own exception if balance < 1000

```

```

        if (bal[i] < 1000)
        {
            MyException me =
                new MyException("Balance is less than 1000");

            throw me;
        }
    }

} //end of try

catch (MyException e) {

    e.printStackTrace();

}

}

}

```

RunTime Error

```

MyException: Balance is less than 1000
    at MyException.main(fileProperty.java:36)

```

Output:

ACCNO	CUSTOMER	BALANCE
1001	Nish	10000.0
1002	Shubh	12000.0
1003	Sush	5600.0
1004	Abhi	999.0

Catching base and derived classes as exceptions

- Difficulty Level : [Easy](#)
- Last Updated : 29 Jun, 2021

Exception Handling – catching base and derived classes as exceptions:

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints “*Caught Base Exception*”

- C

```
#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main()

{

    Derived d;

    // some other stuff

    try {

        // Some monitored code

        throw d;

    }

    catch(Base b) {
```

```

        cout<<"Caught Base Exception";

    }

    catch(Derived d) { //This catch block is NEVER executed

        cout<<"Caught Derived Exception";

    }

    getchar();

    return 0;

}

```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modified program and it prints “*Caught Derived Exception*”

- C

```

#include<iostream>

using namespace std;

class Base {};

class Derived: public Base {};

int main()

{

    Derived d;

    // some other stuff

```

```

try {

    // Some monitored code

    throw d;

}

catch(Derived d) {

    cout<<"Caught Derived Exception";

}

catch(Base b) {

    cout<<"Caught Base Exception";

}

getchar();

return 0;

}

```

In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code. For example, following Java code fails in compilation with error message “*exception Derived has already been caught*”

- Java

```

//filename Main.java

class Base extends Exception {}

class Derived extends Base {}

public class Main {

```

```

public static void main(String args[]) {

    try {

        throw new Derived();

    }

    catch(Base b) {}

    catch(Derived d) {}

}

}

```

Checked vs Unchecked Exceptions in Java

- Difficulty Level : [Easy](#)
- Last Updated : 24 Sep, 2018

In Java, there are two types of exceptions:

1) Checked: are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

For example, consider the following Java program that opens file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

```

import java.io.*;

class Main {

    public static void main(String[] args) {

        FileReader file = new FileReader("C:\\test\\a.txt");

```

```

        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"

        for (int counter = 0; counter < 3; counter++)

            System.out.println(fileInput.readLine());

        fileInput.close();

    }

}

```

Output:

Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -

unreported exception java.io.FileNotFoundException; must be caught or declared to be

thrown

at Main.main(Main.java:5)

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program.

Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

```

import java.io.*;

```

```

class Main {

    public static void main(String[] args) throws IOException {

        FileReader file = new FileReader("C:\\test\\a.txt");

        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\\test\\a.txt"

        for (int counter = 0; counter < 3; counter++)

            System.out.println(fileInput.readLine());

        fileInput.close();

    }

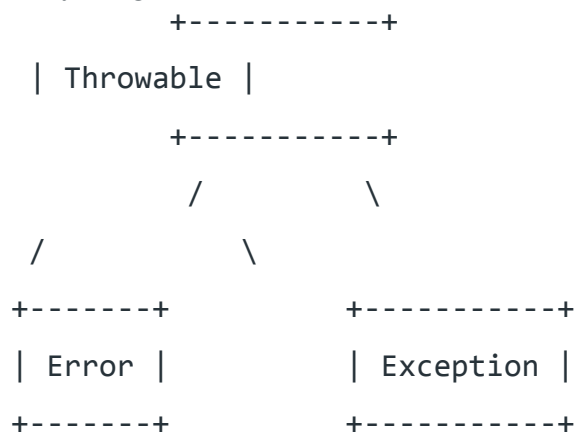
}

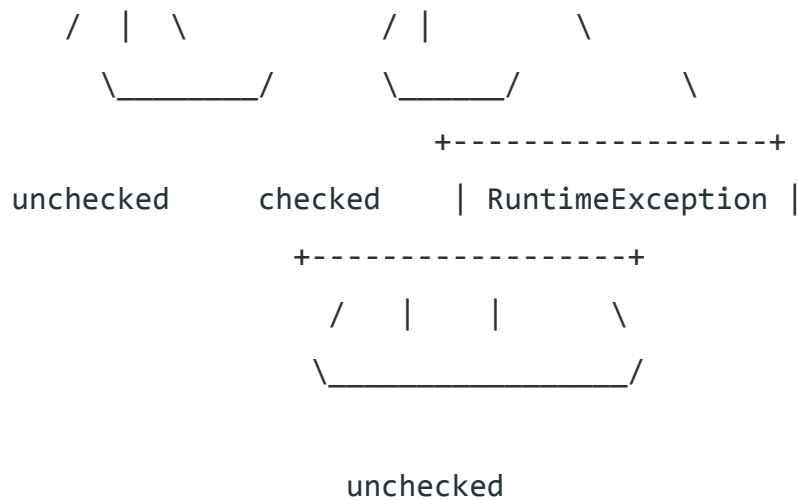
```

Output: First three lines of file “C:\\test\\a.txt”

2) Unchecked are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.

In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.





Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```

class Main {

    public static void main(String args[]) {

        int x = 0;

        int y = 10;

        int z = y/x;

    }

}

```

Output:

```

Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Main.main(Main.java:5)

```

Java Result: 1

Why two types?

See [Unchecked Exceptions — The Controversy](#) for details.

Should we make our exceptions checked or unchecked?

Following is the bottom line from [Java documents](#)

If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception

throw and throws in Java

throw

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either [checked or unchecked exception](#). The throw keyword is mainly used to throw custom exceptions.

Syntax:

throw *Instance*

Example:

```
throw new ArithmeticException("/ by zero");
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and [user defined exceptions typically extend Exception class](#). Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

Java

```
// Java program that demonstrates the use of throw

class ThrowExcep

{

    static void fun()

    {

        try

        {

            throw new NullPointerException("demo");
```

```

    }

    catch (NullPointerException e)

    {

        System.out.println("Caught inside fun().");

        throw e; // rethrowing the exception

    }

}

public static void main (String args[])

{

    try

    {

        fun();

    }

    catch (NullPointerException e)

    {

        System.out.println("Caught in main.");

    }

}

}

```

Output:

Caught inside fun().

Caught in main.

Another Example:

Java

```
// Java program that demonstrates the use of throw

class Test

{

    public static void main(String[] args)

    {

        System.out.println(1/0);

    }

}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

throws

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax:

type method_name(parameters) throws exception_list

exception_list is a comma separated list of all the exceptions which a method might throw.

In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying **unreported exception XXX must be caught or declared to be thrown**. To prevent this compile time error we can handle the exception in two ways:

1. By using [try catch](#)
2. By using **throws** keyword

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

Java

```
// Java program to illustrate error in case
// of unhandled exception

class tst

{

    public static void main(String[] args)

    {

        Thread.sleep(10000);

        System.out.println("Hello Geeks");

    }

}
```

Output:

error: unreported exception InterruptedException; must be caught or declared to be thrown

Explanation: In the above program, we are getting compile time error because there is a chance of exception if the main thread is going to sleep, other threads get the chance to execute main() method which will cause InterruptedException.

Java

```
// Java program to illustrate throws

class tst

{

    public static void main(String[] args) throws InterruptedException

    {
```

```
        Thread.sleep(10000);

        System.out.println("Hello Geeks");

    }

}
```

Output:

Hello Geeks

Explanation: In the above program, by using throws keyword we handled the InterruptedException and we will get the output as **Hello Geeks**

Another Example:

Java

```
// Java program to demonstrate working of throws

class ThrowsExcep

{

    static void fun() throws IllegalAccessException

    {

        System.out.println("Inside fun(). ");

        throw new IllegalAccessException("demo");

    }

    public static void main(String args[])

    {

        try

        {

            fun();

        }

    }

}
```

```

    }

    catch(IllegalAccessException e)

    {

        System.out.println("caught in main.");

    }

}

```

Output:

Inside fun().

caught in main.

Important points to remember about throws keyword:

- throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
- throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
- By the help of throws keyword we can provide information to the caller of the method about the exception.

User-defined Custom Exception in Java

Java provides us facility to create our own exceptions which are basically derived classes of [Exception](#). For example MyException in below code extends the Exception class.

We pass the string to the constructor of the super class- Exception which is obtained using “getMessage()” function on the object created.

```

// A Class that represents use-defined expception

class MyException extends Exception

```

```
{

    public MyException(String s)

    {

        // Call constructor of parent Exception

        super(s);

    }

}


// A Class that uses above MyException

public class Main

{

    // Driver Program

    public static void main(String args[])

    {

        try

        {

            // Throw an object of user defined exception

            throw new MyException("GeeksGeeks");

        }

        catch (MyException ex)

        {

            System.out.println("Caught");

        }

    }

}
```



```

        // Print the message from MyException object

        System.out.println(ex.getMessage());

    }

}

}

```

Output:

Caught

GeeksGeeks

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of [Exception](#) class can also be called without a parameter and call to super is not mandatory.

```

// A Class that represents use-defined expception

class MyException extends Exception

{

}

// A Class that uses above MyException

public class setText

{

    // Driver Program

```

```
public static void main(String args[])
{
    try
    {
        // Throw an object of user defined exception

        throw new MyException();
    }

    catch (MyException ex)
    {
        System.out.println("Caught");

        System.out.println(ex.getMessage());
    }
}
```

Output:

```
Caught
null
```

Java 7 try-with-resources example

Before Java 7, we can close a resource with finally :

```
try{  
    //open resources  
}catch(Exception){  
    //handle exception  
}finally{  
    //close resources  
}
```

In Java 7, a new try-with-resources approach is introduced, it helps to close resources automatically.

```
try(open resources, one or more  
resources){
```

```
//...  
}
```

//after try block, the resource will be closed automatically.

1. BufferedReader

1.1 Before Java 7, we have to close the BufferedReader manually.

```
package com.mkyong.io;
```

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

```
import java.io.FileReader;
```

```
import java.io.IOException;
```

```
public class TestApp {
```

```
public static void main(String[] args) {  
  
    BufferedReader br = null;  
    String line;  
  
    try {  
  
        br = new BufferedReader(new  
FileReader("C:\\testing.txt"));  
        while ((line = br.readLine()) != null)  
{  
  
            System.out.println(line);  
        }  
  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {
```

```
        try {  
            if (br != null) br.close();  
        } catch (IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
  
}
```

1.2 In Java 7, with try-with-resources, the `BufferedReader` will be closed automatically after try block.

```
package com.mkyong.io;
```

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class TestApp {

    public static void main(String[] args) {

        String line;

        try (BufferedReader br = new
BufferedReader(
            new
FileReader("C:\\testing.txt"))) {

            while ((line = br.readLine()) != null)
{
                System.out.println(line);
            }
        }
    }
}
```

```
    }

    } catch (IOException e) {
        e.printStackTrace();
    }

    // br will be closed automatically
}

}
```

2. JDBC

Java example to auto close 2 resources in a try-with-resources statement.

2.1 Before Java 7

@Override

```
public void update(int postId, String  
metaValue) {
```

```
    Connection connection = null;
```

```
    PreparedStatement ps = null;
```

```
    try {
```

```
        connection =  
dataSource.getConnection();
```

```
        ps =  
connection.prepareStatement(SQL_UPDA  
TE_POST_META);
```

```
        ps.setString(1, metaValue);
```

```
        ps.setInt(2, postId);
```

```
        ps.setString(3,  
GlobalUtils.META_KEY);
```

```
        ps.executeUpdate();
```

```
    } catch (Exception e) {
```

```
        //
```

```
    } finally {
```

```
        if (ps != null) {
```

```
            ps.close();
```

```
        }
```

```
        if (connection != null) {
```

```
            connection.close();
```

```
        }
```

```
    }
```

```
}
```

2.2 Java 7 try-with-resources

```
@Override
```

```
    public void update(int postId, String  
    metaValue) {
```

```
        try (Connection connection =  
        dataSource.getConnection());
```

```
            PreparedStatement ps =  
            connection.prepareStatement(SQL_UPDA  
            TE_POST_META)) {
```

```
                ps.setString(1, metaValue);
```

```
                ps.setInt(2, postId);
```

```
                ps.setString(3,  
                GlobalUtils.META_KEY);
```

```
        ps.executeUpdate()

    } catch (Exception e) {

        //...

    }

}
```

Generics in Java

Generics mean **parameterized types**. The idea is to allow type (Integer, String, ... etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types.

An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Why Generics?

The **Object** is the superclass of all other classes and Object reference can refer to any type object. These features lack type safety. Generics add that type safety feature. We will discuss that type of safety feature in later examples.

Generics in Java is similar to [templates in C++](#). For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. There are some fundamental differences between the two approaches to generic types.

Generic Class

Like C++, we use <> to specify parameter types in generic class creation. To create objects of a generic class, we use the following syntax.

```
// To create an instance of generic class
```

```
BaseType <Type> obj = new BaseType <Type>()
```

Note: In Parameter type we can not use primitives like 'int', 'char' or 'double'.

- Java

```
// A Simple Java program to show working of user defined
```

```
// Generic classes
```

```
// We use < > to specify Parameter type
```

```
class Test<T>
```

```
{
```

```
    // An object of type T is declared
```

```
    T obj;
```

```
    Test(T obj) { this.obj = obj; } // constructor
```

```
    public T getObject() { return this.obj; }
```

```
}
```

```
// Driver class to test above
```

```
class Main
```

```
{
```

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

```
    {
```

```
        // instance of Integer type
```

```

Test <Integer> iObj = new Test<Integer>(15);

System.out.println(iObj.getObject());

// instance of String type

Test <String> sObj =

                                new Test<String>("GeeksForGeeks");

System.out.println(sObj.getObject());

}

}

```

Output:

15

GeeksForGeeks

We can also pass multiple Type parameters in Generic classes.

- Java

```

// A Simple Java program to show multiple

// type parameters in Java Generics


// We use < > to specify Parameter type

class Test<T, U>

{

```

```
T obj1; // An object of type T

U obj2; // An object of type U


// constructor

Test(T obj1, U obj2)

{

    this.obj1 = obj1;

    this.obj2 = obj2;

}


// To print objects of T and U

public void print()

{

    System.out.println(obj1);

    System.out.println(obj2);

}

}


// Driver class to test above

class Main

{

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

```

{

    Test <String, Integer> obj =

        new Test<String, Integer>("GfG", 15);


    obj.print();

}

}

```

Output:

GfG

15

Generic Functions:

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method, the compiler handles each method.

- Java

```

// A Simple Java program to show working of user defined

// Generic functions


class Test

{

    // A Generic method example

    static <T> void genericDisplay (T element)

```



```

{

    System.out.println(element.getClass().getName() +

                        " = " + element);

}


// Driver method

public static void main(String[] args)

{

    // Calling generic method with Integer argument

    genericDisplay(11);


    // Calling generic method with String argument

    genericDisplay("GeeksForGeeks");


    // Calling generic method with double argument

    genericDisplay(1.0);

}

}

```

Output :

```

java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0

```

Generics work only with Reference Types:

When we declare an instance of a generic type, the type argument passed to the type parameter must be a reference type. We cannot use primitive data types like **int**, **char**.

- Java

```
Test<int> obj = new Test<int>(20);
```

The above line results in a compile-time error, that can be resolved by using type wrappers to encapsulate a primitive type.

But primitive type array can be passed to the type parameter because arrays are reference type.

- Java

```
ArrayList<int[]> a = new ArrayList<>();
```

Generic Types Differ Based on Their Type Arguments:

Consider the following Java code.

- Java

```
// A Simple Java program to show working
```

```
// of user-defined Generic classes
```

```
// We use < > to specify Parameter type
```

```
class Test<T>
```

```
{
```

```

// An object of type T is declared

T obj;

Test(T obj) { this.obj = obj; } // constructor

public T getObject() { return this.obj; }

}


// Driver class to test above

class Main

{

    public static void main (String[] args)

    {

        // instance of Integer type

        Test <Integer> iObj = new Test<Integer>(15);

        System.out.println(iObj.getObject());


        // instance of String type

        Test <String> sObj =

                                new Test<String>("GeeksForGeeks");

        System.out.println(sObj.getObject());

        iObj = sObj; //This results an error

    }

}

```

Output:

error:

incompatible types:

Test cannot be converted to Test

Even though iObj and sObj are of type Test, they are the references to different types because their type parameters differ. Generics add type safety through this and prevent errors.

Advantages of Generics:

Programs that use Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use it for any type we want.

2. Type Safety: Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of a string, the compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

- Java

```
// A Simple Java program to demonstrate that NOT using  
  
// generics can cause run time exceptions  
  
import java.util.*;  
  
class Test  
{  
  
    public static void main(String[] args)  
  
    {  
  
        // Creating a an ArrayList without any type specified
```

```

    ArrayList al = new ArrayList();

    al.add("Sachin");

    al.add("Rahul");

    al.add(10); // Compiler allows this

    String s1 = (String)al.get(0);

    String s2 = (String)al.get(1);

    // Causes Runtime Exception

    String s3 = (String)al.get(2);

}

}

```

Output :

```

Exception in thread "main" java.lang.ClassCastException:
    java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)

```

How generics solve this problem?

At the time of defining ArrayList, we can specify that this list can take only String objects.

- Java

```

// Using generics converts run time exceptions into

```

```
// compile time exception.

import java.util.*;

class Test

{

    public static void main(String[] args)

    {

        // Creating a an ArrayList with String specified

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

        al.add("Sachin");

        al.add("Rahul");

        // Now Compiler doesn't allow this

        al.add(10);

        String s1 = (String)al.get(0);

        String s2 = (String)al.get(1);

        String s3 = (String)al.get(2);

    }

}
```

Output:

15: error: no suitable method found for add(int)

```
    al.add(10);
```

^

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example every time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not typecast it every time.

- Java

```
// We don't need to typecast individual members of ArrayList
```

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

```
class Test
```

```
{
```

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

```
    {
```

```
        // Creating a an ArrayList with String specified
```

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

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

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

```
        // Typecasting is not needed
```

```
        String s1 = al.get(0);
```

```
String s2 = al.get(1);  
  
}  
  
}
```

4. Generics promotes code reusability.

5. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same, they are type safe too.

Generics in Java with collection classes is very easy but it provides a lot more features than just creating the type of collection.

We will try to learn the features of generics in this article. Understanding generics can become confusing sometimes if we go with jargon words, so I would try to keep it simple and easy to understand.

We will look into below topics of generics in java.

1. [Generics in Java](#)
2. [Java Generic Class](#)
3. [Java Generic Interface](#)
4. [Java Generic Type](#)
5. [Java Generic Method](#)
6. [Java Generics Bounded Type Parameters](#)
7. [Java Generics and Inheritance](#)
8. [Java Generic Classes and Subtyping](#)
9. [Java Generics Wildcards](#)
1. [Java Generics Upper Bounded Wildcard](#)
2. [Java Generics Unbounded Wildcard](#)
3. [Java Generics Lower bounded Wildcard](#)
10. [Subtyping using Generics Wildcard](#)
11. [Java Generics Type Erasure](#)
12. [Generics FAQs](#)

1. Generics in Java

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of `ClassCastException` that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety. Let's see how generics help us using collection classes safely.

```
List list = new ArrayList();
list.add("abc");
list.add(new Integer(5)); //OK

for(Object obj : list){
    //type casting leading to ClassCastException at runtime
    String str=(String) obj;
}
```

Above code compiles fine but throws `ClassCastException` at runtime because we are trying to cast `Object` in the list to `String` whereas one of the element is of type `Integer`. After Java 5, we use collection classes like below.

```
List<String> list1 = new ArrayList<String>(); // java 7 ?
List<String> list1 = new ArrayList<>();
list1.add("abc");
//list1.add(new Integer(5)); //compiler error

for(String str : list1){
    //no type casting needed, avoids ClassCastException
}
```

Notice that at the time of list creation, we have specified that the type of elements in the list will be `String`. So if we try to add any other type of object in the list, the program will throw compile-time error. Also notice that in for loop, we don't need typecasting of the element in the list, hence removing the `ClassCastException` at runtime.

2. Java Generic Class

We can define our own classes with generics type. A generic type is a class or interface that is parameterized over types. We use angle brackets (`<>`) to specify the type parameter.

To understand the benefit, let's say we have a simple class as:

```

package com.journaldev.generics;

public class GenericsTypeOld {

    private Object t;

    public Object get() {
        return t;
    }

    public void set(Object t) {
        this.t = t;
    }

    public static void main(String args[]){
        GenericsTypeOld type = new GenericsTypeOld();
        type.set("Pankaj");
        String str = (String) type.get(); //type casting,
error prone and can cause ClassCastException
    }
}

```

Notice that while using this class, we have to use type casting and it can produce `ClassCastException` at runtime. Now we will use java generic class to rewrite the same class as shown below.

```

package com.journaldev.generics;

public class GenericsType<T> {

    private T t;

    public T get(){
        return this.t;
    }

    public void set(T t1){
        this.t=t1;
    }

    public static void main(String args[]){
        GenericsType<String> type = new GenericsType<>();
        type.set("Pankaj"); //valid

        GenericsType type1 = new GenericsType(); //raw
type
        type1.set("Pankaj"); //valid
    }
}

```

```

        type1.set(10); //valid and autoboxing support
    }
}

```

Notice the use of `GenericType` class in the main method. We don't need to do type-casting and we can remove `ClassCastException` at runtime. If we don't provide the type at the time of creation, the compiler will produce a warning that "GenericType is a raw type."

References to generic type `GenericType<T>` should be parameterized". When we don't provide the type, the type becomes `Object` and hence it's allowing both `String` and `Integer` objects. But, we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

Tip: We can use `@SuppressWarnings("rawtypes")` annotation to suppress the compiler warning, check out [java annotations tutorial](#).

Also notice that it supports [java autoboxing](#).

3. Java Generic Interface

`Comparable` interface is a great example of Generics in interfaces and it's written as:

```

package java.lang;
import java.util.*;

public interface Comparable<T> {
    public int compareTo(T o);
}

```

In similar way, we can create generic interfaces in java. We can also have multiple type parameters as in `Map` interface. Again we can provide parameterized value to a parameterized type also, for example `new HashMap<String, List<String>>()` is valid.

4. Java Generic Type

Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language. So generics also comes with its own naming conventions. Usually, type

parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

5. Java Generic Method

Sometimes we don't want the whole class to be parameterized, in that case, we can create java generics method. Since the [constructor](#) is a special kind of method, we can use generics type in constructors too.

Here is a class showing an example of a java generic method.

```
package com.journaldev.generics;

public class GenericsMethods {

    //Java Generic Method
    public static <T> boolean isEqual(GenericsType<T> g1,
GenericsType<T> g2) {
        return g1.get().equals(g2.get());
    }

    public static void main(String args[]){
        GenericsType<String> g1 = new GenericsType<>();
        g1.set("Pankaj");

        GenericsType<String> g2 = new GenericsType<>();
        g2.set("Pankaj");

        boolean isEqual =
GenericsMethods.<String>isEqual(g1, g2);
        //above statement can be written simply as
        isEqual = GenericsMethods.isEqual(g1, g2);
        //This feature, known as type inference, allows
you to invoke a generic method as an ordinary method, without
specifying a type between angle brackets.
        //Compiler will infer the type that is needed
    }
```

```
}
```

Notice the *isEqual* method signature showing syntax to use generics type in methods. Also, notice how to use these methods in our java program. We can specify type while calling these methods or we can invoke them like a normal method. Java compiler is smart enough to determine the type of variable to be used, this facility is called **type inference**.

6. Java Generics Bounded Type Parameters

Suppose we want to restrict the type of objects that can be used in the parameterized type, for example in a method that compares two objects and we want to make sure that the accepted objects are Comparables. To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, similar like below method.

```
public static <T extends Comparable<T>> int compare(T t1, T
t2) {
    return t1.compareTo(t2);
}
```

The invocation of these methods is similar to unbounded method except that if we will try to use any class that is not Comparable, it will throw compile-time error.

Bounded type parameters can be used with methods as well as classes and interfaces.

Java Generics supports multiple bounds also, i.e <T extends A & B & C>. In this case, A can be an interface or class. If A is class then B and C should be an interface. We can't have more than one class in multiple bounds.

7. Java Generics and Inheritance

We know that [Java inheritance](#) allows us to assign a variable A to another variable B if A is subclass of B. So we might think that any generic type of A can be assigned to generic type of B, but it's not the case. Let's see this with a simple program.

```

package com.journaldev.generics;

public class GenericsInheritance {

    public static void main(String[] args) {
        String str = "abc";
        Object obj = new Object();
        obj=str; // works because String is-a Object,
inheritance in java
        MyClass<String> myClass1 = new MyClass<String>();
        MyClass<Object> myClass2 = new MyClass<Object>();
        //myClass2=myClass1; // compilation error since
MyClass<String> is not a MyClass<Object>
        obj = myClass1; // MyClass<T> parent is Object
    }

    public static class MyClass<T>{}
}

```

We are not allowed to assign `MyClass<String>` variable to `MyClass<Object>` variable because they are not related, in fact `MyClass<T>` parent is `Object`.

8. Java Generic Classes and Subtyping

We can subtype a generic class or interface by extending or implementing it. The relationship between the type parameters of one class or interface and the type parameters of another are determined by the extends and implements clauses.

For example, `ArrayList<E>` implements `List<E>` that extends `Collection<E>`, so `ArrayList<String>` is a subtype of `List<String>` and `List<String>` is subtype of `Collection<String>`.

The subtyping relationship is preserved as long as we don't change the type argument, below shows an example of multiple type parameters.

```

interface MyList<E,T> extends List<E>{
}

```

The subtypes of `List<String>` can be `MyList<String,Object>`, `MyList<String,Integer>` and so on.

9. Java Generics Wildcards

Question mark (?) is the wildcard in generics and represent an unknown type. The wildcard can be used as the type of a parameter, field, or local variable and sometimes as a return type. We can't use wildcards while invoking a generic method or instantiating a generic class. In the following sections, we will learn about upper bounded wildcards, lower bounded wildcards, and wildcard capture.

9.1) Java Generics Upper Bounded Wildcard

Upper bounded wildcards are used to relax the restriction on the type of variable in a method. Suppose we want to write a method that will return the sum of numbers in the list, so our implementation will be something like this.

```
public static double sum(List<Number> list) {  
    double sum = 0;  
    for (Number n : list) {  
        sum += n.doubleValue();  
    }  
    return sum;  
}
```

Now the problem with above implementation is that it won't work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when an upper bounded wildcard is helpful. We use generics wildcard with **extends** keyword and the **upper bound** class or interface that will allow us to pass argument of upper bound or it's subclasses types.

The above implementation can be modified like the below program.

```
package com.journaldev.generics;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class GenericsWildcards {  
    public static void main(String[] args) {  
        List<Integer> ints = new ArrayList<>();  
        ints.add(3); ints.add(5); ints.add(10);  
        double sum = sum(ints);  
    }  
}
```

```

        System.out.println("Sum of ints="+sum);
    }

    public static double sum(List<? extends Number> list) {
        double sum = 0;
        for (Number n : list) {
            sum += n.doubleValue();
        }
        return sum;
    }
}

```

It's similar like writing our code in terms of interface, in the above method we can use all the methods of upper bound class Number. Note that with upper bounded list, we are not allowed to add any object to the list except null. If we will try to add an element to the list inside the sum method, the program won't compile.

9.2) Java Generics Unbounded Wildcard

Sometimes we have a situation where we want our generic method to be working with all types, in this case, an unbounded wildcard can be used. Its same as using `<? extends Object>`.

```

public static void printData(List<?> list) {
    for (Object obj : list) {
        System.out.print(obj + "::");
    }
}

```

We can provide `List<String>` or `List<Integer>` or any other type of Object list argument to the *printData* method. Similar to upper bound list, we are not allowed to add anything to the list.

9.3) Java Generics Lower bounded Wildcard

Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as `List<Integer>` but it will be tied up with Integers whereas `List<Number>` and `List<Object>` can also hold integers, so we can use a lower bound wildcard to achieve this. We use generics wildcard (?) with **super** keyword and lower bound class to achieve this.

We can pass lower bound or any supertype of lower bound as an argument, in this case, java compiler allows to add lower bound object types to the list.

```
public static void addIntegers(List<? super Integer> list) {  
    list.add(new Integer(50));  
}
```

10. Subtyping using Generics Wildcard

```
List<? extends Integer> intList = new ArrayList<>();  
List<? extends Number> numList = intList; // OK. List<?  
extends Integer> is a subtype of List<? extends Number>
```

11. Java Generics Type Erasure

Generics in Java was added to provide type-checking at compile time and it has no use at run time, so java compiler uses **type erasure** feature to remove all the generics type checking code in byte code and insert type-casting if necessary. Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

For example, if we have a generic class like below;

```
public class Test<T extends Comparable<T>> {  
  
    private T data;  
    private Test<T> next;  
  
    public Test(T d, Test<T> n) {  
        this.data = d;  
        this.next = n;  
    }  
  
    public T getData() { return this.data; }  
}
```

The Java compiler replaces the bounded type parameter T with the first bound interface, Comparable, as below code:

```
public class Test {
```

```

private Comparable data;
private Test next;

public Node(Comparable d, Test n) {
    this.data = d;
    this.next = n;
}

public Comparable getData() { return data; }
}

```

12. Generics FAQs

12.1) Why do we use Generics in Java?

Generics provide strong compile-time type checking and reduces risk of ClassCastException and explicit casting of objects.

12.2) What is T in Generics?

We use <T> to create a generic class, interface, and method. The T is replaced with the actual type when we use it.

12.3) How does Generics work in Java?

Generic code ensures type safety. The compiler uses type-erasure to remove all type parameters at the compile time to reduce the overload at runtime.

13. Generics in Java – Further Readings

- Generics doesn't support sub-typing, so `List<Number> numbers = new ArrayList<Integer>();` will not compile, learn [why generics doesn't support sub-typing](#).
- We can't create generic array, so `List<Integer>[] array = new ArrayList<Integer>[10]` will not compile, read [why we can't create generic array?](#).

That's all for **generics in java**, java generics is a really vast topic and requires a lot of time to understand and use it effectively. This post here is an attempt to provide

basic details of generics and how can we use it to extend our program with type-safety.

ArrayList vs LinkedList in Java

An [array](#) is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. However, the limitation of the array is that the size of the array is predefined and fixed. There are multiple ways to solve this problem. In this article, the difference between two [classes](#) which are implemented to solve this problem named [ArrayList](#) and [LinkedList](#) is discussed.

[ArrayList](#)

ArrayList is a part of the [collection framework](#). It is present in the [java.util](#) package and provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed. We can dynamically add and remove items. It automatically resizes itself. The following is an example to demonstrate the implementation of the ArrayList.

```
// Java program to demonstrate the  
  
// working of an ArrayList  
  
import java.io.*;  
  
import java.util.*;  
  
class GFG {  
  
    public static void main(String[] args)  
  
    {
```

```
// Declaring an ArrayList

ArrayList<Integer> arrli

    = new ArrayList<Integer>();


// Appending the new elements

// at the end of the list

for (int i = 1; i <= 5; i++)

    arrli.add(i);


// Printing the ArrayList

System.out.println(arrli);


// Removing an element from the

// List

arrli.remove(3);


// Printing the ArrayList after

// removing the element

System.out.println(arrli);

}
```

```
}
```

Output:

```
[1, 2, 3, 4, 5]
```

```
[1, 2, 3, 5]
```

LinkedList

This class implements the [LinkedList Data Structure](#). Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate [object](#) with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the [arrays](#). The following is an example to demonstrate the implementation of the LinkedList.

```
// Java program to demonstrate the  
  
// working of a LinkedList  
  
import java.util.*;  
  
public class Test {  
  
    public static void main(String args[])  
  
    {  
  
        // Creating an object of the  
  
        // class linked list  
  
        LinkedList<String> object  
  
            = new LinkedList<String>();  
  
  
  
        // Adding the elements to the
```

```

// linked list

object.add("A");

object.add("B");

object.addLast("C");


System.out.println(object);


// Removing elements from the

// list

object.remove("B");

object.removeFirst();


System.out.println("Linked list after "

                    + "deletion: " + object);

}

}

```

Output:

[A, B, C]

Linked list after deletion: [C]

Difference Between ArrayList and LinkedList in Java

ArrayList	LinkedList
-----------	------------

This class uses a dynamic array to store the elements in it. With the introduction of generics , this class supports the storage of all types of objects.	This class uses a doubly linked list to store the elements in it. Similar to the ArrayList, this class also supports the storage of all types of objects.
Manipulating ArrayList takes more time due to the internal implementation. Whenever we remove an element, internally, the array is traversed and the memory bits are shifted.	Manipulating LinkedList takes less time compared to ArrayList because, in a doubly-linked list, there is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
This class implements a List interface . Therefore, this acts as a list.	This class implements both the List interface and the Deque interface . Therefore, it can act as a list and a deque.
This class works better when the application demands storing the data and accessing it.	This class works better when the application demands manipulation of the stored data.

Difference between TreeSet, LinkedHashSet and HashSet in Java with Example

TreeSet, LinkedHashSet and HashSet all are implementation of Set interface and by virtue of that, they follow contract of Set interface i.e. they do not allow duplicate elements. Despite being from same type hierarchy, there are a lot of differences between them; which is important to understand, so that you can choose most appropriate [Set](#) implementation based upon your requirement. By the way, difference between TreeSet and HashSet or LinkedHashSet is also one of the [popular Java Collection interview question](#), not as popular as [Hashtable vs HashMap](#) or [ArrayList vs Vector](#) but still appears in various Java interviews.

Difference between HashMap, TreeMap, and LinkedHashMap in Java

major difference between `HashMap`, `TreeMap` and `LinkedHashMap` classes in Java.

We know that a [Map](#) is an object that represents a mapping from unique keys to values. Java offers several useful implementations of the `java.util.Map` interface, such as [HashMap](#), [TreeMap](#) and [LinkedHashMap](#), which are more or less similar in functionality. This post provides an overview of some of the major differences between these implementations.

1. Implementation Details

The `HashMap` and `LinkedHashMap` classes implement the `Map` interface, whereas `TreeMap` implements the `Map`, `NavigableMap`, and `SortedMap` interface. A `HashMap` is implemented as a Hash table, a `TreeMap` is implemented as a Red-Black Tree, and `LinkedHashMap` is implemented as a doubly-linked list buckets in Java.

2. Iteration Order of mappings

We know that a map's contents can be viewed as a set of keys, collection of values, or set of key-value mappings. The most important difference between the `HashMap`, `TreeMap` and `LinkedHashMap` class lies in the order in which their iterators return the map's contents.

1. `HashMap` makes no guarantees on the iteration order of the map. Also, the addition and removal of any element might change its iteration order.
2. `TreeMap`, on the other hand, is iterated according to the natural ordering of its keys or according to the `Comparator` specified at the map's creation time.
3. `LinkedHashMap` maintains a doubly-linked list through all of its entries. This linked list defines the iteration order, which is the order in which keys were inserted into the map.

Here's a simple Java program to illustrate the iteration order of `HashMap`, `TreeMap` and `LinkedHashMap`.

```
1 import java.util.HashMap;

2 import java.util.LinkedHashMap;

3 import java.util.Map;

4 import java.util.TreeMap;

5
```



```

6 class Main

7 {

8     public static void main(String[] args)

9     {

10         Map<String, String> hm = new LinkedHashMap<>();

11

12         hm.put("USA", "Washington");

13         hm.put("United Kingdom", "London");

14         hm.put("India", "New Delhi");

15

16         System.out.println("LinkedHashMap : " + hm);

17

18         hm = new TreeMap<>(hm);

19         System.out.println("TreeMap : " + hm);

20

21         hm = new HashMap<>(hm);

22         System.out.println("HashMap : " + hm);

23     }

24 }

```

```

LinkedHashMap : {USA=Washington, United Kingdom=London, India=New
Delhi}
TreeMap : {India=New Delhi, USA=Washington, United Kingdom=London}
HashMap : {United Kingdom=London, USA=Washington, India=New Delhi}

```

3. Performance

Assuming the hash method disperses the elements properly among the buckets, `HashMap` and `LinkedHashMap` offers $O(1)$ time performance for the basic operations such as `get`, `put`, `containsKey`, `remove`, etc., On the other hand, `TreeMap` guarantees $O(\log(n))$ time cost for these operations. Please note that due to the added expense of maintaining the doubly-linked list, `LinkedHashMap`'s performance is slightly lower than that of `HashMap`. So if performance is an issue, `HashMap` is preferred.

Now coming to the space complexity, `HashMap` requires less memory than `TreeMap` and `LinkedHashMap` since it uses a hash table to store the mappings. `LinkedHashMap` has the extra overhead of a doubly-linked list, and `TreeMap` is implemented as a Red-black tree, which takes more memory.

4. Null values/keys

`HashMap` and `LinkedHashMap` permits null values and null key, whereas `TreeMap` permits only null values (not null keys) if the natural ordering of keys is used. It supports null keys only if its `Comparator` supports comparison on null keys.

Which Implementation to use?

Use `HashMap` when performance is critical, and the ordering of keys doesn't matter.

Use `TreeMap` when keys need to be ordered using their natural ordering or by a `Comparator`.

Use `LinkedHashMap` if the insertion order of the keys should be preserved.

Difference between TreeSet, LinkedHashSet and HashSet in Java

`TreeSet`, `LinkedHashSet` and `HashSet` in Java are three Set implementation in collection framework and like many others they are also used to store objects. Main feature of `TreeSet` is sorting, `LinkedHashSet` is insertion order and `HashSet` is just general

purpose collection for storing object.

HashSet is implemented using [HashMap in Java](#) while TreeSet is implemented using [TreeMap](#). TreeSet is a SortedSet implementation which allows it to keep elements in the sorted order defined by either [Comparable or Comparator interface](#).

Comparable is used for natural order sorting and Comparator for [custom order sorting](#) of objects, which can be provided while creating instance of TreeSet.

Anyway before seeing difference between TreeSet, LinkedHashSet and HashSet, let's see some similarities between them:

- 1) **Duplicates** : All three implements Set interface means they are not allowed to store duplicates.
- 2) **Thread safety** : HashSet, TreeSet and LinkedHashSet are not [thread-safe](#), if you use them in multi-threading environment where at least one Thread modifies Set you need to externally synchronize them.
- 3) **Fail-Fast Iterator** : Iterator returned by TreeSet, LinkedHashSet and HashSet are fail-fast Iterator. i.e. If Iterator is modified after its creation by any way other than Iterators `remove()` method, it will throw `ConcurrentModificationException` with best of effort. read more about [fail-fast vs fail-safe Iterator](#) here

Now let's see the **difference between HashSet, LinkedHashSet and TreeSet in Java** :

1. Performance and Speed

First difference between them comes in terms of speed. HashSet is fastest, LinkedHashSet is second on performance or almost similar to HashSet but TreeSet is bit slower because of sorting operation it needs to perform on each insertion. TreeSet provides guaranteed $O(\log(n))$ time for common operations like

add, remove and contains, while `HashSet` and `LinkedHashSet` offer constant time performance e.g. $O(1)$ for add, contains and remove given hash function uniformly distribute elements in bucket.

2. Ordering

`HashSet` does not maintain any order while `LinkedHashSet` maintains **insertion order of elements** much like `List` interface and `TreeSet` maintains sorting order of elements.

3. Internal Implementation

`HashSet` is backed by an `HashMap` instance, `LinkedHashSet` is implemented using `HashSet` and `LinkedList` while `TreeSet` is backed up by `NavigableMap` in Java and by default it uses `TreeMap`.

4. null

Both `HashSet` and `LinkedHashSet` allows null but `TreeSet` doesn't allow null but `TreeSet` doesn't allow null and throw [java.lang.NullPointerException](#) when you will insert null into `TreeSet`. Since `TreeSet` uses [compareTo\(\) method](#) of respective elements to compare them which throws `NullPointerException` while comparing with null, here is an example:

```
TreeSet cities
```

```
Exception in thread "main" java.lang.NullPointerException
```

```
    at java.lang.String.compareTo(String.java:1167)
```

```
    at java.lang.String.compareTo(String.java:92)
```

```
    at java.util.TreeMap.put(TreeMap.java:545)
```

```
    at java.util.TreeSet.add(TreeSet.java:238)
```

5. Comparison

`HashSet` and `LinkedHashSet` uses [equals\(\) method in Java](#) for comparison but `TreeSet` uses [compareTo\(\) method](#) for maintaining ordering. That's why `compareTo()` should be consistent to `equals` in Java. failing to do so break general contract of `Set` interface i.e. it can

permit duplicates.

TreeSet vs HashSet vs LinkedHashSet - Example

Let's compare all these Set implementation on some points by writing Java program. In this example we are demonstrating difference in ordering, time taking while inserting 1M records among [TreeSet](#), HashSet and LinkedHashSet in Java. This will help to solidify some points which discussed in earlier section and help to decide when to use HashSet, LinkedHashSet or TreeSet in Java.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

/**
 * Java program to demonstrate difference between TreeSet, HashSet
 * and LinkedHashSet
 *
 * in Java Collection.
 * @author
 */
public class SetComparision {

    public static void main(String args[]){
        HashSet<String> fruitsStore = new HashSet<String>();
        LinkedHashSet<String> fruitMarket
= new LinkedHashSet<String>();
        TreeSet<String> fruitBuzz = new TreeSet<String>();

        for(String fruit: Arrays.asList("mango", "apple", "banana"))
        {
            fruitsStore.add(fruit);
            fruitMarket.add(fruit);
            fruitBuzz.add(fruit);
        }
    }
}
```

```

    }

    //no ordering in HashSet - elements stored in random order
    System.out.println("Ordering in HashSet :" + fruitsStore);

    //insertion order or elements - LinkedHashSet stores
elements as insertion
    System.err.println("Order of element in LinkedHashSet :" +
fruitMarket);

    //should be sorted order - TreeSet stores element in sorted
order
    System.out.println("Order of objects in TreeSet :" +
fruitBuzz);

    //Performance test to insert 10M elements in HashSet,
LinkedHashSet and TreeSet
    Set<Integer> numbers = new HashSet<Integer>();
    long startTime = System.nanoTime();
    for(int i =0; i<10000000; i++){
        numbers.add(i);
    }

    long endTime = System.nanoTime();
    System.out.println("Total time to insert 10M elements in
HashSet in sec : "

        + (endTime - startTime));

    // LinkedHashSet performance Test - inserting 10M objects
    numbers = new LinkedHashSet<Integer>();
    startTime = System.nanoTime();

```

```

        for(int i =0; i<10000000; i++){
            numbers.add(i);
        }
        endTime = System.nanoTime();
        System.out.println("Total time to insert 10M elements in
LinkedHashSet in sec : "

                                + (endTime - startTime));

        // TreeSet performance Test - inserting 10M objects
        numbers = new TreeSet<Integer>();
        startTime = System.nanoTime();
        for(int i =0; i<10000000; i++){
            numbers.add(i);
        }
        endTime = System.nanoTime();
        System.out.println("Total time to insert 10M elements in
TreeSet in sec : "

                                + (endTime - startTime));
    }
}

```

Output

```

Ordering in HashSet :[banana, apple, mango]
Order of element in LinkedHashSet :[mango, apple, banana]
Order of objects in TreeSet :[apple, banana, mango]
Total time to insert 10M elements in HashSet in sec : 3564570637
Total time to insert 10M elements in LinkedHashSet in sec
: 3511277551
Total time to insert 10M elements in TreeSet in sec : 10968043705

```

When to use HashSet, TreeSet and LinkedHashSet in Java

Since all three implements Set [interface](#) they can be used for common Set operations like not allowing duplicates but since HashSet, TreeSet and LinkedHashSet has there

special feature which makes them appropriate in certain scenario.

Because of sorting order provided by `TreeSet`, use `TreeSet` when you need a collection where elements are sorted without duplicates. [HashSet](#) are rather general purpose Set implementation, Use it as default Set implementation if you need a fast, duplicate free collection.

While `LinkedHashSet` is extension of `HashSet` and its more suitable where you need to maintain **insertion order** of elements, similar to [List](#) without compromising performance for costly `TreeSet`.

Another use of `LinkedHashSet` is for creating copies of existing Set, Since `LinkedHashSet` preserves insertion order, it returns Set which contains same elements in same order like exact copy.

In short, although all three are Set interface implementation they offer distinctive feature, `HashSet` is a general purpose Set while `LinkedHashSet` provides insertion order guarantee and `TreeSet` is a `SortedSet` which stores elements in sorted order specified by [Comparator or Comparable in Java](#).

How to copy object from one Set to other

Here is code example of `LinkedHashSet` which demonstrate How `LinkedHashSet` can be used to copy objects from one Set to another without losing order. You will get exact replica of source Set, in terms of contents and order. Here static method `copy(Set source)` is written using [Generics](#), This kind of [parameterized method](#) provides type-safety and help to avoid `ClassCastException` at runtime.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

/**
 * Java program to copy object from one HashSet to another using
 * LinkedHashSet.
```



```

    * HashSet preserves order of element while copying elements.
    *

    */
public class SetUtils{

    public static void main(String args[]) {

        HashSet<String> source
= new HashSet<String>(Arrays.asList("Set, List, Map"));
        System.out.println("source : " + source);
        Set<String> copy = SetUtils.copy(source);
        System.out.println("copy of HashSet using HashSet: " +
copy);
    }

    /*
    * Static utility method to copy Set in Java
    */
    public static <T> Set<T> copy(Set<T> source){
        return new HashSet<T>(source);
    }
}

```

Output:

```

source : [Set, List, Map]
copy of HashSet using HashSet: [Set, List, Map]

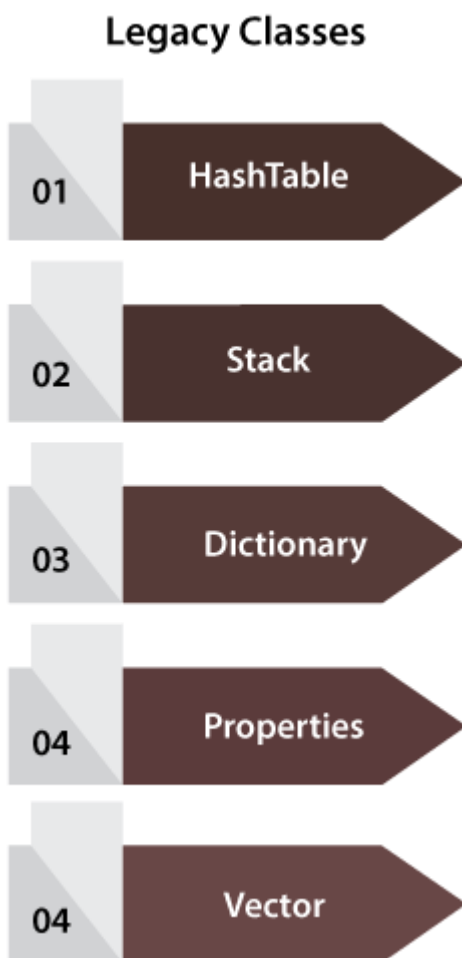
```

Legacy Class in Java

In the past decade, the **Collection** framework didn't include in Java. In the early version of Java, we have several classes and interfaces which allow us to store objects. After adding the [Collection framework](#) in **JSE 1.2**, for supporting the collections framework, these classes were re-engineered. So, classes and interfaces that formed the collections framework in the older version of [Java](#) are known as **Legacy classes**. For supporting generic in JDK5, these classes were re-engineered.

All the legacy classes are synchronized. The **java.util** package defines the following **legacy** classes:

1. HashTable
2. Stack
3. Dictionary
4. Properties
5. Vector



Vector Class

[Vector](#) is a special type of ArrayList that defines a dynamic array. ArrayList is not synchronized while **vector** is synchronized. The vector class has several legacy methods that are not present in the collection framework. Vector implements Iterable after the release of JDK 5 that defines the vector is fully compatible with collections, and vector elements can be iterated by the for-each loop.

Vector class provides the following four constructors:

1) Vector()

It is used when we want to create a default vector having the initial size of 10.

2) Vector(int size)

It is used to create a vector of specified capacity. It accepts size as a parameter to specify the initial capacity.

3) Vector(int size, int incr)

It is used to create a vector of specified capacity. It accepts two parameters size and increment parameters to specify the initial capacity and the number of elements to allocate each time when a vector is resized for the addition of objects.

4) Vector(Collection c)

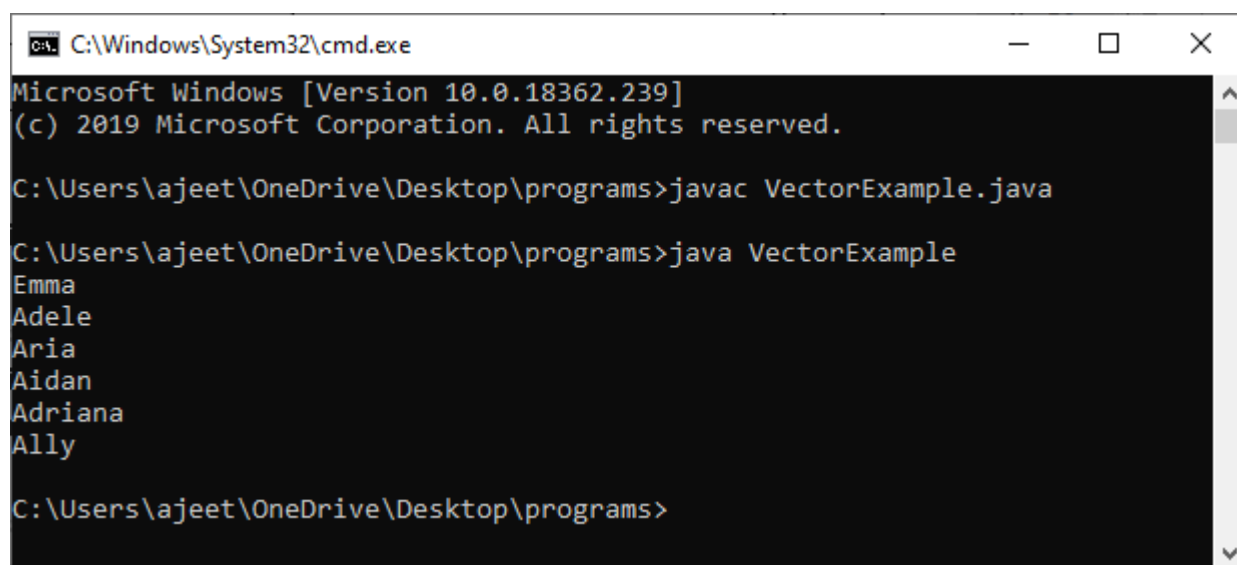
It is used to create a vector with the same elements which are present in the collection. It accepts the collection as a parameter.

VectorExample.java

```
1. import java.util.*;
2. public class VectorExample
3. {
4.     public static void main(String[] args)
5.     {
6.         Vector<String> vec = new Vector<String>();
7.         vec.add("Emma");
8.         vec.add("Adele");
9.         vec.add("Aria");
10.        vec.add("Aidan");
11.        vec.add("Adriana");
12.        vec.add("Ally");
13.        Enumeration<String> data = vec.elements();
14.        while(data.hasMoreElements())
15.        {
16.            System.out.println(data.nextElement());
```

```
17.    }
18. }
19. }
```

Output:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\ajeet\OneDrive\Desktop\programs>javac VectorExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java VectorExample
Emma
Adele
Aria
Aidan
Adriana
Ally

C:\Users\ajeet\OneDrive\Desktop\programs>
```

Hashtable Class

The [Hashtable class](#) is similar to [HashMap](#). It also contains the data into key/value pairs. It doesn't allow to enter any null key and value because it is synchronized. Just like Vector, Hashtable also has the following four constructors.

1) Hashtable()

It is used when we need to create a default Hashtable having size 11.

2) Hashtable(int size)

It is used to create a Hashtable of the specified size. It accepts size as a parameter to specify the initial size of it.

3) Hashtable(int size, float fillratio)

It creates the **Hashtable** of the specified size and fillratio. It accepts two parameters, size (of type int) and fillratio (of type float). The fillratio must be between 0.0 and 1.0. The **fillratio** parameter determines how full the hash table can be before it is resized upward. It means when we enter more elements than its capacity or size than the Hashtable is expended by multiplying its size with the fullratio.

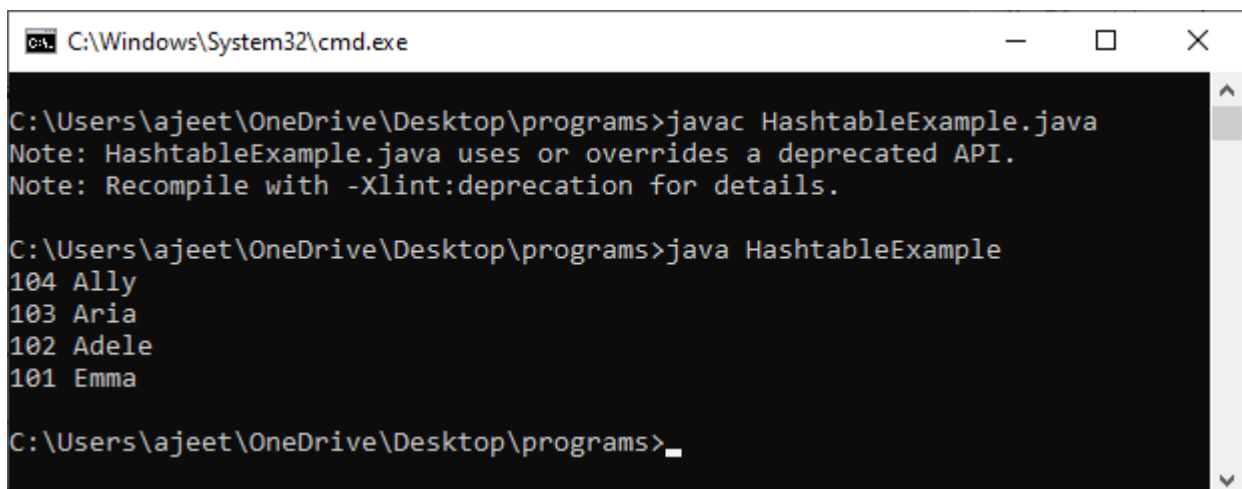
4) Hashtable(Map< ? extends K, ? extends V> m)

It is used to create a Hashtable. The Hashtable is initialized with the elements present in **m**. The capacity of the Hashtable is the twice the number elements present in **m**.

HashtableExample.java

```
1. import java.util.*;
2. class HashtableExample
3. {
4.     public static void main(String args[])
5.     {
6.         Hashtable<Integer,String> student = new Hashtable<Integer, String>();
7.         student.put(new Integer(101), "Emma");
8.         student.put(new Integer(102), "Adele");
9.         student.put(new Integer(103), "Aria");
10.        student.put(new Integer(104), "Ally");
11.        Set dataset = student.entrySet();
12.        Iterator iterate = dataset.iterator();
13.        while(iterate.hasNext())
14.        {
15.            Map.Entry map=(Map.Entry)iterate.next();
16.            System.out.println(map.getKey()+" "+map.getValue());
17.        }
18.    }
19.}
```

Output:



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac HashtableExample.java
Note: HashtableExample.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java HashtableExample
104 Ally
103 Aria
102 Adele
101 Emma

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Properties Class

Properties class extends Hashtable class to maintain the list of values. The list has both the key and the value of type string. The **Property** class has the following two constructors:

1) Properties()

It is used to create a Properties object without having default values.

2) Properties(Properties propdefault)

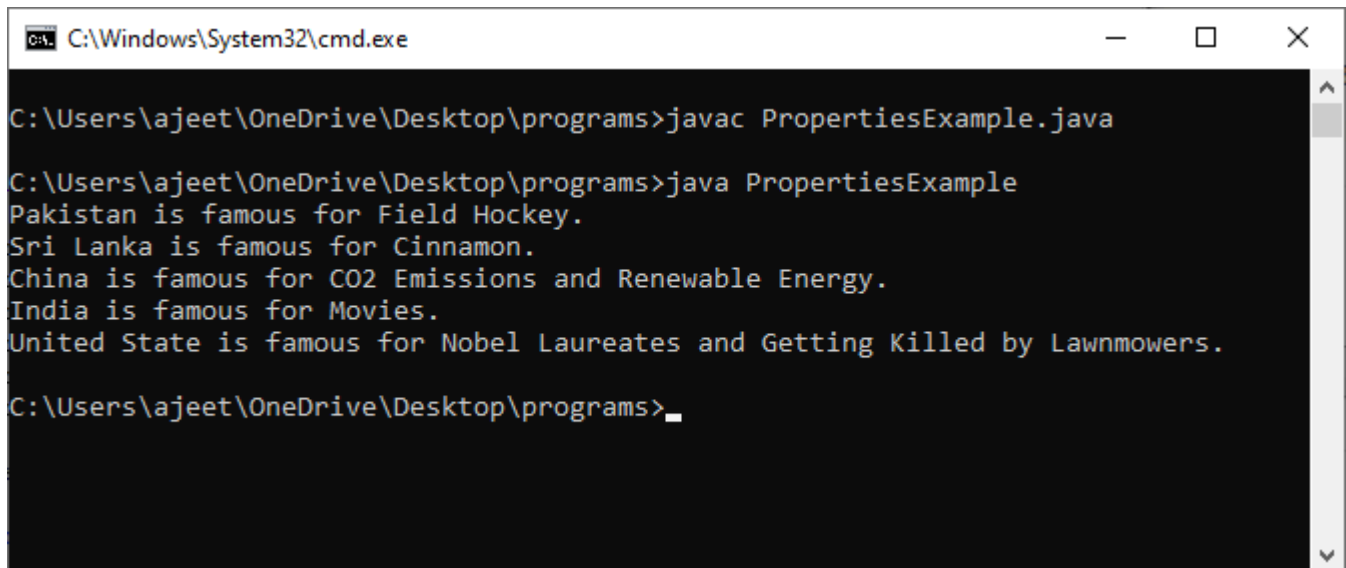
It is used to create the Properties object using the specified parameter of properties type for its default value.

The main difference between the Hashtable and Properties class is that in Hashtable, we cannot set a default value so that we can use it when no value is associated with a certain key. But in the Properties class, we can set the default value.

PropertiesExample.java

```
1. import java.util.*;
2. public class PropertiesExample
3. {
4.     public static void main(String[] args)
5.     {
6.         Properties prop_data = new Properties();
7.         prop_data.put("India", "Movies.");
8.         prop_data.put("United State", "Nobel Laureates and Getting Killed by Lawnmower
9. s.");
10.        prop_data.put("Pakistan", "Field Hockey.");
11.        prop_data.put("China", "CO2 Emissions and Renewable Energy.");
12.        prop_data.put("Sri Lanka", "Cinnamon.");
13.        Set< ?> set_data = prop_data.keySet();
14.        for(Object obj: set_data)
15.        {
16.            System.out.println(obj+" is famous for "+ prop_data.getProperty((String)obj));
17.        }
18.    }
```

Output:



```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac PropertiesExample.java

C:\Users\ajeet\OneDrive\Desktop\programs>java PropertiesExample
Pakistan is famous for Field Hockey.
Sri Lanka is famous for Cinnamon.
China is famous for CO2 Emissions and Renewable Energy.
India is famous for Movies.
United State is famous for Nobel Laureates and Getting Killed by Lawnmowers.

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

Stack Class

[Stack class](#) extends Vector class, which follows the LIFO(LAST IN FIRST OUT) principal for its elements. The stack implementation has only one default constructor, i.e., Stack().

1) Stack()

It is used to create a stack without having any elements.

There are the following methods can be used with Stack class:

1. The **push()** method is used to add an object to the stack. It adds an element at the top of the stack.
2. The **pop()** method is used to get or remove the top element of the stack.
3. The **peek()** method is similar to the pop() method, but it can't remove the stack's top element using it.
4. The **empty()** method is used to check the emptiness of the stack. It returns true when the stack has no elements in it.
5. The **search()** method is used to ensure whether the specified object exists on the stack or not.

StackExample.java

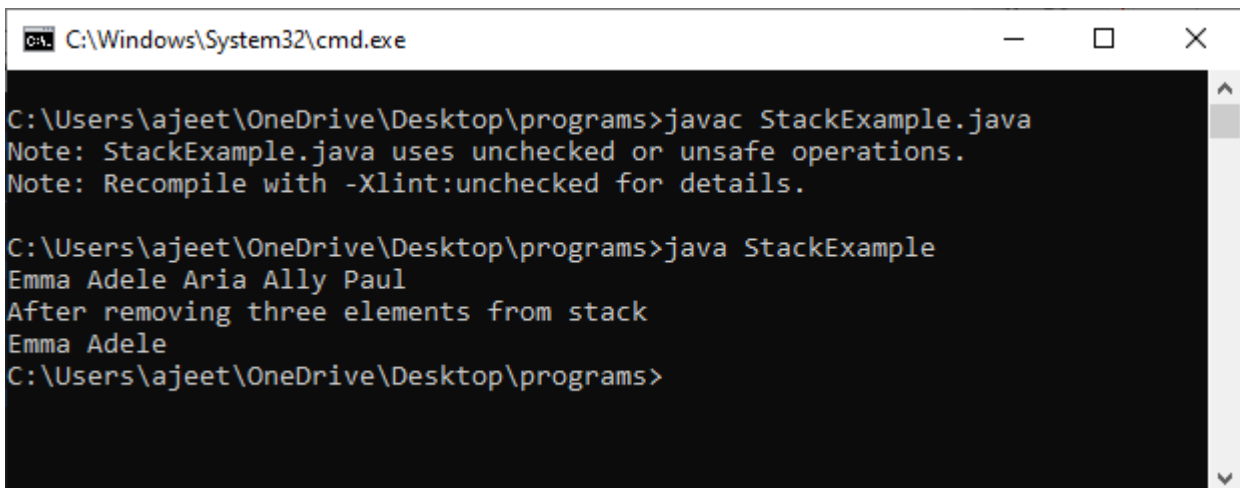
1. **import** java.util.*;
2. **class** StackExample {
3. **public static void** main(String args[]) {

```

4.      Stack stack = new Stack();
5.      stack.push("Emma");
6.      stack.push("Adele");
7.      stack.push("Aria");
8.      stack.push("Ally");
9.      stack.push("Paul");
10.     Enumeration enum1 = stack.elements();
11.     while(enum1.hasMoreElements())
12.         System.out.print(enum1.nextElement()+" ");
13.     stack.pop();
14.     stack.pop();
15.     stack.pop();
16.     System.out.println("\nAfter removing three elements from stack");
17.     Enumeration enum2 = stack.elements();
18.     while(enum2.hasMoreElements())
19.         System.out.print(enum2.nextElement()+" ");
20.     }
21. }

```

Output:



```

C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac StackExample.java
Note: StackExample.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java StackExample
Emma Adele Aria Ally Paul
After removing three elements from stack
Emma Adele
C:\Users\ajeet\OneDrive\Desktop\programs>

```

Dictionary Class

The **Dictionary** class operates much like Map and represents the **key/value** storage repository. The **Dictionary class** is an abstract class that stores the data into the key/value pair. We can define the dictionary as a list of key/value pairs.

The dictionary class provides the following methods:

1. The **put(K key, V value)** method is used to add a key-value pair to the dictionary.
2. The **elements()** method is used to get the value representation in the dictionary.
3. The **get(Object key)** method is used to get the value mapped with the argumented key in the dictionary.
4. The **isEmpty()** method is used to check whether the dictionary is empty or not.
5. The **keys()** method is used to get the key representation in the dictionary.
6. The **remove(Object key)** method removes the data from the dictionary.
7. The **size()** method is used to get the size of the dictionary.

DictionaryExample.java

```
1. import java.util.*;
2. public class DictionaryExample
3. {
4.     public static void main(String[] args)
5.     {
6.         // Initializing Dictionary object
7.         Dictionary student = new Hashtable();
8.
9.         // Using put() method to add elements
10.        student.put("101", "Emma");
11.        student.put("102", "Adele");
12.        student.put("103", "Aria");
13.        student.put("104", "Ally");
14.        student.put("105", "Paul");
15.
16.        //Using the elements() method
17.        for (Enumeration enum1 = student.elements(); enum1.hasMoreElements();)
18.        {
19.            System.out.println("The data present in the dictionary : " + enum1.nextElement());
20.        }
21.
22.        // Using the get() method
```

```
23.     System.out.println("\nName of the student 101 : " + student.get("101"));
24.     System.out.println("Name of the student 102 : " + student.get("102"));
25.
26.     //Using the isEmpty() method
27.     System.out.println("\n Is student dictionary empty? : " + student.isEmpty() + "\n");
28.
29.     // Using the keys() method
30.     for (Enumeration enum2 = student.keys(); enum2.hasMoreElements();)
31.     {
32.         System.out.println("Ids of students: " + enum2.nextElement());
33.     }
34.
35.     // Using the remove() method
36.     System.out.println("\nDelete : " + student.remove("103"));
37.     System.out.println("The name of the deleted student : " + student.get("123"));
38.
39.     System.out.println("\nThe size of the student dictionary is : " + student.size());
40.
41. }
42. }
```

Output:

```
C:\Windows\System32\cmd.exe

C:\Users\ajeet\OneDrive\Desktop\programs>javac DictionaryExample.java
Note: DictionaryExample.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\ajeet\OneDrive\Desktop\programs>java DictionaryExample
The data present in the dictionary : Paul
The data present in the dictionary : Ally
The data present in the dictionary : Aria
The data present in the dictionary : Adele
The data present in the dictionary : Emma

Name of the student 101 : Emma
Name of the student 102 : Adele

Is student dictionary empty? : false

Ids of students: 105
Ids of students: 104
Ids of students: 103
Ids of students: 102
Ids of students: 101

Delete : Aria
The name of the deleted student : null

The size of the student dictionary is : 4

C:\Users\ajeet\OneDrive\Desktop\programs>_
```

All the classes that we have discussed above are known as legacy classes. For supporting generic in JDK5, the above-discussed classes were re-engineered.

Collections.sort() in Java with Examples

java.util.Collections.sort() method is present in java.util.Collections class. It is used to sort the elements present in the specified [list](#) of Collection in ascending order.

It works similar to [java.util.Arrays.sort\(\)](#) method but it is better then as it can sort the elements of Array as well as linked list, queue and many more present in it.

```
public static void sort(List myList)
```

myList : A List type object we want to sort.

This method doesn't return anything

Example:

Let us suppose that our list contains

```
{"Geeks For Geeks", "Friends", "Dear", "Is", "Superb"}
```

After using `Collection.sort()`, we obtain a sorted list as

```
{"Dear", "Friends", "Geeks For Geeks", "Is", "Superb"}
```

Sorting an ArrayList in ascending order

```
// Java program to demonstrate working of Collections.sort()

import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings

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

        al.add("Geeks For Geeks");

        al.add("Friends");

        al.add("Dear");

        al.add("Is");

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

```

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */

        Collections.sort(al);

        // Let us print the sorted list

        System.out.println("List after the use of" +

                           " Collection.sort() :\n" + al);

    }

}

```

Output:

```

List after the use of Collection.sort() :
[Dear, Friends, Geeks For Geeks, Is, Superb]

```

Sorting an ArrayList in descending order

```

// Java program to demonstrate working of Collections.sort()
// to descending order.

import java.util.*;

public class Collectionsorting
{

    public static void main(String[] args)

```

```

{

    // Create a list of strings

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

    al.add("Geeks For Geeks");

    al.add("Friends");

    al.add("Dear");

    al.add("Is");

    al.add("Superb");


    /* Collections.sort method is sorting the

    elements of ArrayList in ascending order. */

    Collections.sort(al, Collections.reverseOrder());


    // Let us print the sorted list

    System.out.println("List after the use of" +

                        " Collection.sort() :\n" + al);

}

}

```

Output:

```

List after the use of Collection.sort() :
[Superb, Is, Geeks For Geeks, Friends, Dear]

```

Sorting an ArrayList according to user defined criteria.

We can use [Comparator Interface](#) for this purpose.

```
// Java program to demonstrate working of Comparator

// interface and Collections.sort() to sort according
// to user defined criteria.

import java.util.*;

import java.lang.*;

import java.io.*;


// A class to represent a student.

class Student

{

    int rollno;

    String name, address;


    // Constructor

    public Student(int rollno, String name,

                    String address)

    {

        this.rollno = rollno;

        this.name = name;

        this.address = address;

    }

}
```

```
// Used to print student details in main()

public String toString()

{

    return this.rollno + " " + this.name +

        " " + this.address;

}

}
```

```
class Sortbyroll implements Comparator<Student>

{

    // Used for sorting in ascending order of

    // roll number

    public int compare(Student a, Student b)

    {

        return a.rollno - b.rollno;

    }

}
```

```
// Driver class

class Main

{
```



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

{

    ArrayList<Student> ar = new ArrayList<Student>();

    ar.add(new Student(111, "bbbb", "london"));

    ar.add(new Student(131, "aaaa", "nyc"));

    ar.add(new Student(121, "cccc", "jaipur"));


    System.out.println("Unsorted");

    for (int i=0; i<ar.size(); i++)

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


    Collections.sort(ar, new Sortbyroll());


    System.out.println("\nSorted by rollno");

    for (int i=0; i<ar.size(); i++)

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

}

}
```

Output :

Unsorted

111 bbbb london

131 aaaa nyc

121 cccc jaipur

Sorted by rollno

111 bbbb london

121 cccc jaipur

131 aaaa nyc

[Arrays.sort\(\)](#) vs **Collections.sort()**

Arrays.sort works for arrays which can be of primitive data type

also. [Collections.sort\(\)](#) works for objects Collections like [ArrayList](#), [LinkedList](#), etc.

We can use Collections.sort() to sort an array after creating a ArrayList of given array items.

```
// Using Collections.sort() to sort an array

import java.util.*;

public class Collectionsort

{

    public static void main(String[] args)

    {

        // create an array of string objs

        String domains[] = {"Practice", "Geeks",

                            "Code", "Quiz"};

        // Here we are making a list named as Collist

        List collist =

            new ArrayList(Arrays.asList(domains));

        // Collection.sort() method is used here

        // to sort the list elements.
```

```
        Collections.sort(colList);

// Let us print the sorted list

System.out.println("List after the use of" +

                    " Collection.sort()  :\n" +

                    colList);

    }

}
```

Output:

```
List after the use of Collection.sort()  :
[Code, Geeks, Practice, Quiz]
```