Training Document

Comments: Comments can be used to explain code, and to make it more readable.

- And they can also be used to prevent execution when testing alternative code.

Single line comments- Single line comments starts with the two forward slashes(//).

- And the text between // and the end of the line is ignored by Java.

Multi-line comments- Multi-line comments start with /* and ends with */

- Any text between /* and */ will be ignored by Java.

Example:

```
public class CommentsExp

{

//Main method (Single line comment).

public static void main(String[] args)

{

/* The below statement is used to print the data at the
console screen by using print statement(Multi-line comment).
*/

System.out.println("Using Comments ");

}

}
```

Identifiers: Identifiers in Java are symbolic names used for identification.

- ❖ The identifiers are the class names, variable names, method names.
- ❖ The identifiers starts with the alphabet or Symbol but never start with the Number.

Example:

1. RUN—it is a valid Identifier
2. Run—it is a valid Identifier
3. Run7—it is valid identifier
4. 7Run—it is invalid Identifier
5. Run program—it is invalid identifer(Spaces are not allowed)

Keywords: Keywords are the special words with specific meaning.

- ❖ All the keywords must be used in smaller cases.

Some of the keywords are

- All the primitive datatypes are the keywords
- They are: Int, double, char, boolean, byte, short, long, float.
- And other keywords are: public, private, package, static, void, new, extends, instanceof, import, return, class....etc

Example:

```
import java.util.Scanner;
public class KeywordExp
{
public static void main(String[] args)
{
Scanner scan = new Scanner(System.in);
System.out.println("enter the value");
int a = scan.nextInt();
System.out.println(a);
}
}
```

In above example we have covered some keywords named as import, public, static, void, int.

Literals: A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

- ❖ Literals can be assigned to any primitive type variable.

Example:

int a = 200;

double b = 2.3;

In above examples int and double are the datatypes and the 200, 2.3 are literals.

For Integral data types (byte, short, int, long), we can specify literals in 4 ways:

Decimal literals (Base 10): In this form the allowed digits are 0-9.

int x = 101; **Octal literals (Base 8):** In this form the allowed digits are 0-7. The octal number should be prefix with 0.int x = 0146; **Hexa-decimal literals (Base 16):** In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive.

The hexa-decimal number should be prefix

with 0X or 0x.int x = 0X123Face;

Binary literals: From 1.7 onward we can specify literals value even in binary form also, allowed digits are 0 and 1.

Literals value should be prefixed with 0b or 0B.int x = 0b1111;

Example:

```
public class Test {

    public static void main(String[] args)

    {

        int a = 101; // decimal-form literal

        int b = 0100; // octal-form literal

        int c = 0xFace; // Hexa-decimal form literal

        int d = 0b1111; // Binary literal

        System.out.println(a);

        System.out.println(b);

        System.out.println(c);

        System.out.println(d);

    }

}
```

Separators: Separators help define the structure of a program. The separators used in HelloWorld are parentheses, ( ), braces, { }, the period, (.), and the semicolon, (;).

(.)-Period: It is used to separate the package name from sub-package name & class name. It is also used to separate variable or method from its object or instance.

(,)-Comma: It is used to separate the consecutive parameters in the method definition. It is also used to separate the consecutive variables of same type while declaration.

(;)-Semicolon: It is used to terminate the statement in Java.

()- Parenthesis: This holds the list of parameters in method definition. Also used in control statements & type casting.

{}-braces: This is used to define the block/scope of code, class, methods.

[ ]-Brackets: It is used in array declaration.

Example: In the above mentioned examples we have used all this types of literals.

Operators: Operators are used to perform operations on variables and values.

We have five types of operators:

1. Arithmetic operators
2. Assignment operators
3. Comparison operators
4. Logical operators
5. Bitwise operators

Arithmetic operators: Arithmetic operators are used to perform common mathematical operations.

Operator	**Name**	**operation**	**Example**
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y

/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment operators: Assignment operators are used to assign values to variables.

Operator	**Example**	**Example**
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators: Comparison operators are used to compare two values.

Operator	**Description**	**Example**
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators: Logical operators are used to determine the logic between variables or values.

Operator	**Name**	**Description**	**Example**

&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Bitwise Operators: Bitwise operators are used to performing manipulation of individual bits of a number. They can be used with any of the integral types (char, short, int, etc). They are used when performing update and query operations of Binary indexed tree.

Example:

```
public                  class                   Demo2                 {
public      static      void      main(String[]      args)      {
System.out.println(10>20                 &&                10<20);
System.out.println(!true                 ||                 true);
System.out.println(100.2>100              &&               false);
System.out.println((4+5>60-1)||(98*0<0));
System.out.println(50/2>30                &&              !false);
int                     a                   =                  40;
int                    b                   =                 a+10;
System.out.println(!(a<b)                 ||                b>30);
System.out.println(!(599<=       599.9        &&          a>b));
int                      c                   =                  a;
b                            +=                                 c;
a                            +=                                 c;
System.out.println(!(c>b));
System.out.println(a>c&                                      c<b);
System.out.println((00>94      &&      a<=b)||(a+30>b-30));
System.out.println((!true&&a<c)&&(!false&&b<c));

}
}
```

Variables: A variable is a container which holds the value while the program is executed. A variable is assigned with a data type.

- ❖ Variable is a name of memory location. There are three types of variables in java: local, instance and static.

- A variable declared inside the body of the method is called **local variable.** A local variable cannot be defined with "static" keyword.
- A variable declared inside the class but outside the body of the method, is called an **instance variable.** It is not declared as static.
- A variable that is declared as static is called a **static variable**. It cannot be local. You can create a single copy of the static variable and share it among all the instances of the class.

Example:

```
public class VariableExp
{
public String name; //instance variable
public VariableExp(String name)
{
this.name = name; //this keyword is used for instance
}
public static void test1()
{
int a = 10; //local variable
double b = 1.1; // local variable
System.out.println(a);
System.out.println(b);
}
public static void main(String[] args)
{
VaiableExp v1 = new VariableExp(“XYZ”);
System.out.println(v1.name);
test1();
}
}
```

Conversions: Converting one primitive datatype into another is known as type casting (type conversion). You can cast the primitive datatypes in two ways namely, Widening and Narrowing.

Widening − Converting a lower datatype to a higher datatype is known as widening. In this case the casting/conversion is done automatically therefore, it is known as implicit type casting. In this case both datatypes should be compatible with each others.

Narrowing − Converting a higher datatype to a lower datatype is known as narrowing. In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast operator "( )" explicitly. Therefore, it is known as explicit type casting. In this case both datatypes need not be compatible with each other.

Example:

```
public class Conversion {
public static void main(String[] args) {
int a = 45;
double b = 56;
char c = (char)b;
int ch = '8';
System.out.println(a);
System.out.println(b);
System.out.println(c);
System.out.println(ch);
int r = 98;
double y = 99;
int w = (int)y;
char l = 99;
char v = (char)w;
System.out.println(l);
System.out.println(v);
}
}
```

Packages: A package is a group of similar types of classes, interfaces and sub-packages.

- ❖ Package in java can be categorized in two form, built-in package and user-defined package.

❖ There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc

Example:

```
package demopack1;

public class A {
protected int i = 10;
public static void main(String[] args) {
A a1 = new A();
System.out.println(a1.i);
}
}


package demopack1;

public class B {
public static void main(String[] args) {
A a1 = new A();
System.out.println(a1.i);
}

}



package demopack2;

import demopack1.A;

public class C extends A {
public static void main(String[] args) {
C c1 = new C();
System.out.println(c1.i);
}

}
```

Arrays: Normally, an array is a collection of similar type of elements which has contiguous memory location.

Array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Example:

```
public class ArrayEmp{
public static void main(String args[]){
int a[]=new int[5];
a[0]=10;
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
for(int i=0;i<a.length;i++)
System.out.println(a[i]);
}
}
```

Classes: A class is a blueprint from which individual objects are created. A class can contain any of the local variables, instance variables, Class variables.

Interfaces: An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

- Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Example: For both Class and Interface

```
interface Animal //Interface
{
public void eat(); //Abstract method
}
```

```
class Tiger implements Animal//class implementing interface
{
public void eat() //inherited method implementation
{
System.out.println("Tiger eats");
}
}

public class MainClass {
public static void main(String args[]) {
Animal tiger = new Tiger();
tiger.eat();
Tiger tiger1 = new Tiger();
tiger1.eat();
}
}
```

Inner Classes: Inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Example:

```
class OuterClass {
   // Simple nested inner class
   class Inner {
      public void show() {
           System.out.println("In a nested class method");
      }
```

```
    }
}
class MainClass {
    public static void main(String[] args) {
        OuterClass.Inner i = new OuterClass().new Inner();
        i.show();
    }
}
```

Anonymous Inner Class:
It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

Packages: **Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, wavelabs.staff.development.Employee and wavelabs.staff.testing.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:
- Default – No keyword required
- Private
- Protected
- Public

→ Providing controlled access:

→ **protected** and **default** have package level access control. A **protected** member is accessible by classes in the same package and its subclasses,and it can accessible in different package with import statement.

→ A **default member** (without any access specifier) is accessible by classes in the same package only.

→ **public member** will accessible in the same class, within another class of same package, and within another class of different package.

→ **private member** will accessible only in the same class,and it is not accessible within another class of same package and different packages of the project.

- The difference between the members is provided in the below table.

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Access specifiers in java

```
package pack1;

public class A

{

public String name = "packages"; //public member

protected int i  = 10; //protected member

int j = 99; //default member

private double b = 2.2; //private member

{

Public static void main(String[] args)

{

A a1 = new A1();

System.out.println(a1.i);

System.out.println(a1.j);
System.out.println(a1.name);

System.out.println(a1.b);

}

}

Package pack1;

public class B

{

public static void main(Strin[] args)

{

A a1 = new A();

System.out.println(a1.i);

// Within same package the protected member is accessed.

System.out.println(a1.j);

System.out.println(a1.name);

System.out.println(a1.b);/* b is private variable it cannot
be accessed in another class so this statement shows an error
*/

}

}

package pack2;

Import pack1.A;

//if it was public class import is sufficient

Class C extends A // if it was protected extends was required

{

Public static void main(String[] args)

{

C c1 = new C1();

System.out.println(c1.i);
System.out.println(a1.j);/* it will shows an error because
the default member is not accessed in the another package we
can access it in the same package */

System.out.println(a1.name);

System.out.println(a1.b); /* The private member is accessed
in within the class */

}

}
```

In the above packages all the Access specifers are explained with their accessing.

Advantages of using packages in java:

1. **Maintenance:** Java packages are used for proper maintenance. If any developer newly joined a company, he can easily reach to files needed.
2. **Reusability:** We can place the common code in a common folder so that everybody can check that folder and use it whenever needed.
3. **Name conflict:** Packages help to resolve the naming conflict between the two classes with the same name. Assume that there are two classes with the same name Employee.java.
4. **Organized:** It also helps in organizing the files within our project.
5. **Access Protection:** A package provides access protection. It can be used to provide visibility control. The members of the class can be defined in such a manner that they will be visible only to elements of that package.

Types of Packages in Java:

1. **User-defined Package:** The package which is defined by the user is called a User-defined package. It contains user-defined classes and interfaces.

Creating package in Java: Java supports a keyword called “package” which is used to create user-defined packages in java programming. It has the following general form:

Package packageName;

Naming Convention to declare User-defined Package in Real-time Project

While developing your project, you must follow some naming conventions regarding packages declaration. Let’s take an example to understand the convention.

See below a complete package structure of the project.

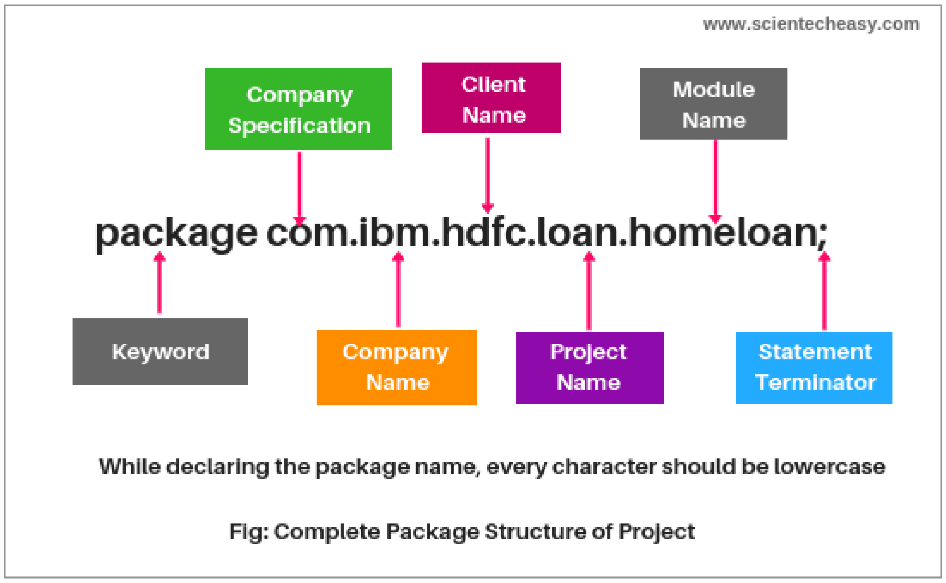


Fig: Complete Package Structure of Project

We are working in Wavelabs and the domain name of WAVELABS is www.wavelabs.com. You can declare the package by reversing the domain like this:

package com.wavelabs;

where,

- com → It is generally company specification name and the folder starts with com which is called root folder.
- wavelabs→ Company name where the product is developed. It is the subfolder.

2. Predefined Packages in Java (Built-in Packages):

Predefined packages in java are those which are developed by Sun Microsystem. They are also called built-in packages in java.

These packages consist of a large number of predefined classes, interfaces, and methods that are used by the programmer to perform any task in his programs.

Java APIs contains the following predefined packages, as shown in the below figure:

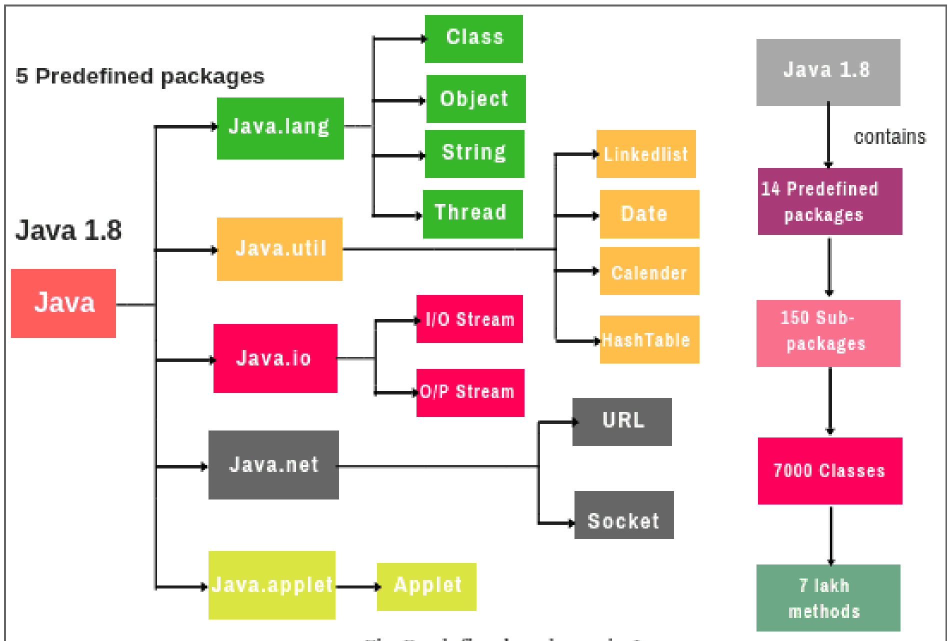


Fig: Predefined packages in Java

Core packages:

1. **Java.lang:** lang stands for language. The Java language package consists of java classes and interfaces that form the core of the Java language and the JVM. It is a fundamental package that is useful for writing and executing all Java programs.

Examples are classes, objects, String, Thread, predefined data types, etc. It is imported automatically into the Java programs.

```
import java.lang.Object;

/* We don’t need to write this statement it will automatically
imported for java.lang packages impicitly */

public class Flower {

public String flower;

public String color;

public Flower(String flower, String color){

this.flower = flower;

this.color = color;

}

public String toString(){

//toString() overriden from Object class
return flower+"s are in "+color+" color";
}
public boolean equals(Object obj){
// equals method overridden from Object class
if (obj instanceof Flower)
{
Flower flower = (Flower)obj;
return this.color == flower.color;
}
else{
return false;
}
}
public static void main(String[] args) {
Flower flower1 = new Flower("Rose", "Red");
Flower flower2 = new Flower("Tulip", "Red");
System.out.println(flower1);
System.out.println(flower2);
System.out.println(flower1.equals(flower2));
}
```

In the above example we are using Object class which is present in the java.lang package and we are overriding the Object class methods to the flower class.

2. **Java.io:** io stands for input and output. It provides a set of I/O streams that are used to read and write data to files. A stream represents a flow of data from one place to another place.

```
import java.io.File;
public class  Demo1{
public static void main(String[] args) {
File f1 = new File("D:/data/info.text");
f1.createNewFile():

System.out.println("File created successfully");

}

}
```

3. **Java util:** util stands for utility. It contains a collection of useful utility classes and related interfaces that implement data structures like Scanner, LinkedList, Dictionary, HashTable, stack, vector, Calender, data utility, etc.

```
import java.util.Scanner;

class Test

{

public static void main(String[] args)

{

Scanner scan = new Scanner(System.in);

/* when ever we want to give the literals at execution time
the we want to import the Scanner class */

System.out.println("Enter 1st Number..");

int a = scan.nextInt();

System.out.println("Enter 2nd Number");

int b = scan.nextInt();

System.out.println(a/b);

}

}
```

Classes:

Classes are a blueprint for creating individual objects that contain the general characteristics of a defined object type. A modifier may or may not be used to declare a class.

Class Declarations: A class declaration specifies a new named reference type. There are two kinds of class declarations:

- ❑ **normal class declarations**: A normal class declarations may include class modifiers.

→ Class modifiers may be one of the Annotation, private, public, protected, abstract, final, static and strictfp.

```
modifier class MyClass { //class header
//field,
//constructor.
 //method declarations
}
```

→ **Enum Declarations:** An enum is a special "class" that represents a group of constants (unchangeable variables, like final variables).

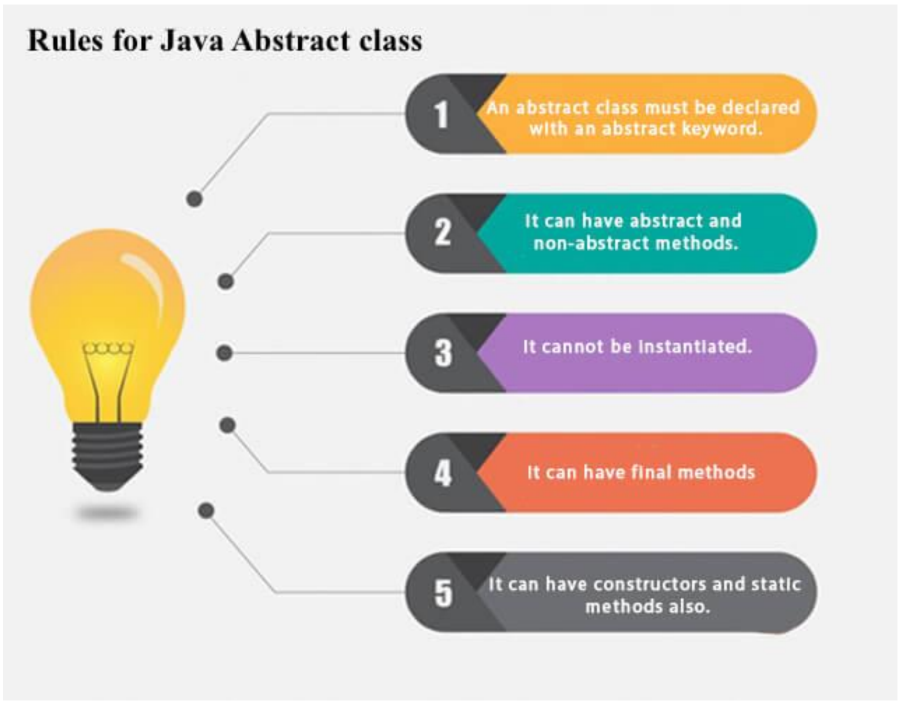
→ To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters.

```
enum Color{
/* A simple enum example where enum is declared outside any
class (Note enum keyword instead of class keyword) */
    RED, GREEN, BLUE;
}
public class Test{
    // Driver method
    public static void main(String[] args{
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

The body of a class declares members (fields and methods and nested classes and interfaces), instance and static initializers, and constructors.

```
public class Car {
    //fields:
    public int speed;
    public int gear;
    //Constructor:
    public Car(int speed, int gear) {
        this.gear = gear;
        this.speed = speed;
    }
    //Methods:
    public void setGear (int newValue) {
        this.gear = newValue;
    }
    public void applyBrake(int decrement) {
        this.speed -= decrement;
    }
    public void speedUp(int increment) {
        this.speed += increment;
    }
     // Main Method:
    public static void main(String[] args) {
     Car c = new Car(30, 3);
     c.setGear(4);
     c.applyBrake(10);
     c.speedUp(50);
     }
     }
```

Abstract classes: A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.


Rules for Java Abstract class
1 An abstract class must be declared with an abstract keyword.
2 It can have abstract and non-abstract methods.
3 It cannot be instantiated.
4 It can have final methods
5 It can have constructors and static methods also.

```
abstract class Vechile{  // abstract class
  abstract void run();  // abstract method
}
class Bike extends Vechile{  // implemented class
void run(){ //abstract method implementation
System.out.println("running safely");
}
public static void main(String args[]){
Vechile obj = new Bike();  // upcasting to super class.
obj.run();
}
}
```

In the above example the abstract method which is present in the Vechile class is implemented in its subclass Bike and the method is upcasted.

Final Classes: When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class.

→ One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer,Float etc. are final classes. We cannot extend them.

```
final class Animal {
     public void hunt() {
System.out.println("Animal hunts");
}
}
public class Lion extends Animal { /* Here we will get a compilation error about The type Lion cannot subclass the final class Animal */
public void hunt(){
System.out.println("Lion hunts other Animals");
}
}
```

→ The other use of final with classes is to create an immutable class like the predefined String class. You cannot make a class immutable without making it final.

```
public final class Student {
    /**
     * Integer and String classes are immutable
     */
    private final Integer studentId;
    private final String studentName;

    public Student(Integer i, String s) {
        this.studentId = i;
        this.studentName = s;
    }

    public String getStudentName() {
        return studentName;
    }
    public Integer getStudentId() {
        return studentId;
    }
    public String toString() {
        return "Student Name is "+ studentName + " and id is"
+studentId;
    }
    public static void main(String[] args) {
Student s1 = new Student(23, "Name");
System.out.println(s1);
}
}
```

strictfp class: If a class declared as strictfp all concrete methods in that class has to follow IEEE 754 standard, so that we will get platform independent results for floating point arithmetic.

→ Strictfp modifier was introduced in java 1.2 version strictfp means strict floating point applicable for classes and methods but not for variables.

```
strictfp class Demo {
public static void main(String[] args) {
System.out.println(10.0/3);
}
}
```

→ The result of floating-point arithmetic is varied from platform to platform

windows-- 3.3333333333333335

linux --- 3.3333333

MAC ---- 3.33333333333334I

If we want platform independent results for floating point arithmetic strictfp modifier is used.

Generic classes and type parameters: 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.

```
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("Name");

}

}
```

Example by using ArrayList

```
package generics;

import java.util.*;

class ArrayListExp {

public static void main(String args[]) {

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

list.add("FirstName");

list.add("SurName");
String s = list.get(1);
System.out.println("element is: " + s);

Iterator<String> itr = list.iterator();
while (itr.hasNext()) {
System.out.println(itr.next());
}
}
}
```

Inner classes: Inner class are defined inside the body of another class (known as outer class). These classes can have access modifier or even can be marked as abstract and final. Inner classes have special relationship with outer class instances. This relationship allows them to have access to outer class members including private members too.

Inner classes can be defined in four different ways:

→ Inner class
→ method – local inner class
→ Anonymous inner class
→ Static nested class

Inner class: An inner class is declared inside the curly braces of another enclosing class. Inner class is coded below

```
public class OuterClass {//Outer class
 private int myVar= 1;

 private class InnerClass {// Inner class
    public void test () {
    System.out.println("Value of myVar is :" + myVar);
    }
  }
```

```
     public void display(){
          InnerClass inner = new InnerClass();
           inner. test();
       }
       public static void main(String args[]){
          OuterClass obj = new OuterClass();
           obj.display();
       }
    }
```

Method-local inner class: A method local inner class is defined within a method of the enclosing class. If you want to use inner class, you must instantiate the inner class in the same method, but after the class definition code. Only two modifiers are allowed for method-local inner class which are abstract and final. The inner class can use the local variables of the method (in which it is present), only if they are marked final.

```
public class Outerclass2 {//Outer class
public void test() {
int num = 23;
class InnerClass2 {//Method-Local Inner class
public void display() {
System.out.println("Inner class Number is " + num);
}
}
InnerClass2 inner = new InnerClass2();
inner.display();
}
 public static void main(String args[]) {
Outerclass2 outer = new Outerclass2();
outer.test();
}
}
```

Anonymous inner class: It is a type of inner class which

1. has no name.
2. can be instantiated only once.
3. is usually declared inside a method or a code block, a curly brace ending with semicolon.
4. is accessible only at the point where it is defined.
5. does not have a constructor simply because it does not have a name
6. cannot be static.

```
class Food {//outer class
public void eat() {
System.out.println("eat pizza");
}
}
class Get {//Inner class
public static void main(String[] args) {
Food p = new Food() {// AnonymousInner class
public void eat() {
System.out.println("anonymous pizza");
}
};
p.eat();
}
}
```

Another Example by using Abstract class as follows.

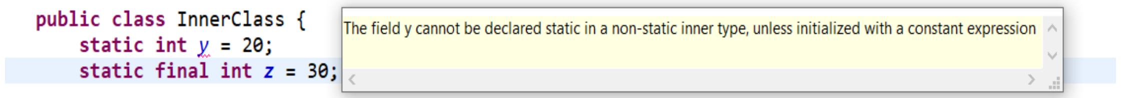
```
abstract class Person {//Outer class
  abstract void eat();
}
class Inner {//Inner class
 public static void main(String args[]){
  Person p=new Person(){ //Anonymous Inner class
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Static Nested Classes: A static nested classes are the inner classes marked with static modifier. Because this is static in nature so this type of inner class doesn't share any special kind of relationship with an instance of outer class. A static nested class cannot access non static members of outer class.

```
class Outer{// outer class
   static class Nested{}// static nested class
}


class Inner{//Inner class
   public static void main(String[] args){
      // use both class names
      Outer.Nested n= new Outer.Nested();
   }
}
```

1. **An inner class is a nested class that is not explicitly or implicitly declared static.**

```
public class OuterClass {
public static int x = 10;
class InnerClass {
static int y = 20; // static member is not allowed to
//declare in inner class
public void test() {
System.out.println("non-Static Method");
}
}
public static void main(String[] args) {
System.out.println(x);
System.out.println(InnerClass.y);
System.out.println(InnerClass.z);
OuterClass.InnerClass obj = new OuterClass().new InnerClass();
obj.test();
}
}
```



```
public class InnerClass {
    static int y = 20;
    static final int z = 30;
```

The field y cannot be declared static in a non-static inner type, unless initialized with a constant expression

→ In the above example an inner class is not declared as static implicitly or explicitly. If a nested class is marked static then it is called **static nested class** and if it not then it is called **non-static nested class** or **inner class**.

2. **An inner class may be a non-static member class, a local class, or an anonymous class.**

```
class Anonymous {
public void display() {
System.out.println("Anonymous class");
}
}
public class Outer1 {
class Inner1 {
private int x = 50;
public void display1() {
System.out.println("non-static method");
System.out.println(x);
}
public static void main(String[] args) {
Outer1.Inner1 obj = new Outer1().new Inner1();
obj.display1();
Anonymous obj1 = new Anonymous(){//AnonymousClass
public void display() {
System.out.println("Inner Anonymous");
}
};
obj1.display();
}
}
}
```

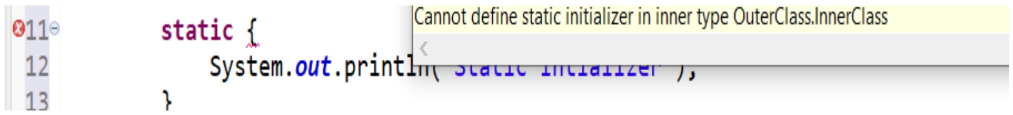
→ An inner class may be a non-static member class, a local class, or an anonymous class. A member class of an interface is implicitly static so is never considered to be an inner class.

3. A member class of an interface is implicitly static so is never considered to be an inner class.

```
interface Outer3 {
class Inner {
static int z = 30; // static member of interface
}
public static void main(String[] args) {
System.out.println(Inner.z);
}
}
```

4. **It is a compile-time error if an inner class declares a static initializer.**

```
public class OuterClass1 {
int z = 10;
class InnerClass1 {
public void display()
{
System.out.println("z");
}
static { // it gives compilation error for static block
System.out.println("Static Initializer");
}
}
public static void main(String[] args) {
OuterClass1.InnerClass1 obj = new OuterClass1().new
InnerClass1();
obj.display();
}
}
```

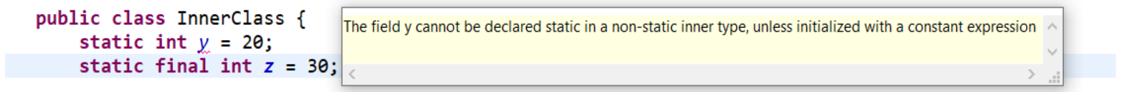


```
static {
    System.out.println("Static Initializer");
}
```

Cannot define static initializer in inner type OuterClass.InnerClass

5. It is a compile-time error if an inner class declares a member that is explicitly or implicitly static, unless the member is a constant variable.

```
class OuterClass {
 static int x = 10;
public class InnerClass {
static int y = 20; // static member is not allowed to
//declare in inner class
static  final  int  z  =  30;//  by  making  static  member  as
//constant it is allowed
public void test() {
System.out.println("non-Static Method");
}
}
}
public class MainClass1 {
public static void main(String[] args) {
System.out.println(OuterClass.x);
System.out.println(OuterClass.InnerClass.y);
System.out.println(OuterClass.InnerClass.z);
OuterClass.InnerClass obj = new OuterClass().new InnerClass();
obj.test();
}
}
```



```
public class InnerClass {
    static int y = 20;
    static final int z = 30;
```

The field y cannot be declared static in a non-static inner type, unless initialized with a constant expression

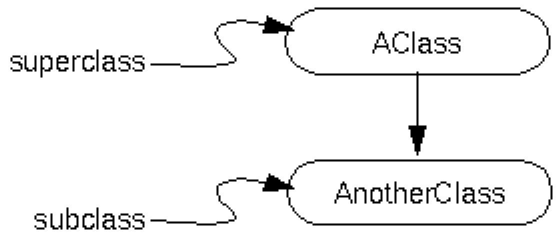
6. An inner class may inherit static members that are not constant variables even though it cannot declare them.

```
class Demo {
static int z = 10; //static member and not a constant
}
public class Outer4 {
public class Inner4 extends Demo {
int y = 15;
}
public static void main(String[] args) {
System.out.println(Inner4.z);
/* Z variable is inherited to innerclass so called through Inner class name */
Outer4.Inner4 obj = new Outer4().new Inner4();
System.out.println(obj.y);
}

}
```

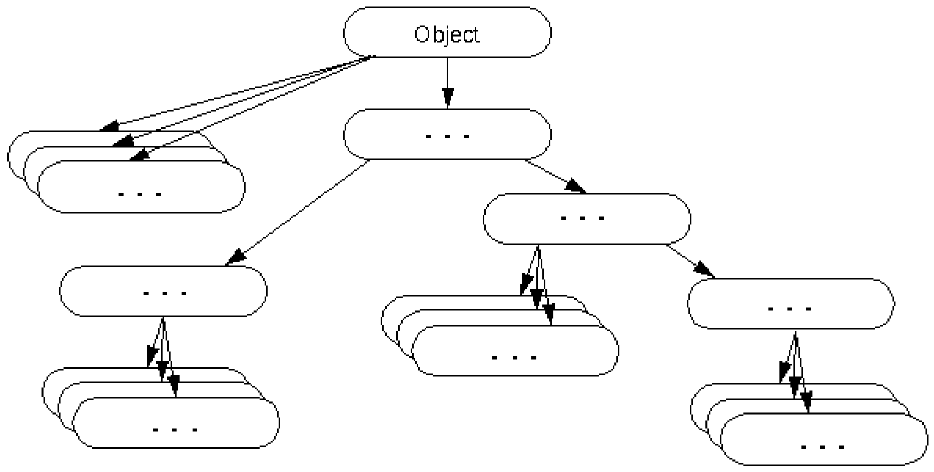
7. **A nested class that is not an inner class may declare static members freely, in accordance with the usual rules of the Java programming language.**

```
public class Outer5 {

static int z = 10;

public void display() {

System.out.println("Outerclass member "+z);

}

   static class Inner5 {

static int x = 15;

 /* Once the class is mentioned as static it don't shows
compilation error for static members */

public void display1() {

System.out.println("Innerclass member "+x);

}

}

public static void main(String[] args) {

Outer5 obj = new Outer5();

obj.display();

Outer5.Inner5 obj1 = new Outer5.Inner5();

obj1.display1();

}

}
```

Superclasses and subclasses: classes can be derived from other classes. The derived class (the class that is derived from another class) is called a **subclass**. The class from which its derived is called the **superclass**. And the extends keyword is used to derive a subclass from its superclass.



The top-most class, the class from which all other classes are derived, is the Object class defined in java.lang. Object is the root of a hierarchy of classes.



All objects in the Java environment inherit either directly or indirectly from the Object class.

```
public class Employee {

private String role = "Developer";

private String companyName = "Wavelabs";

public String getRole() {

return role;

}

protected void setRole(String role) {

this.role = role;

}

protected String getCompanyName() {

return companyName;
}
protected void setCompanyName(String companyName) {
this.companyName = companyName;
}
public void work() {
System.out.println("doing projects");
}
}
public class TeamLead extends Employee {
private String mainWork = "Coding";

public static void main(String args[]) {
TeamLead obj = new TeamLead();
/*
 * Note: we are not accessing the data members directly we
are using public
 * getter method to access the private members of super
class
 */
System.out.println(obj.getCompanyName());
System.out.println(obj.getRole());
System.out.println(obj.mainWork);
obj.work();
}
}
```

Superinterfaces: Superinterface is not a separate entity, it refers to the hierarchy. When an interface is extended or implemented it becomes a Superinterface of what extended/implemented it. If class C implements interface B, and interface B extends interface A, then A is a Superinterface to B and. Also, B is a Superinterface for C.

1.Inerface inherits interface.

```
interface A { // superinterface to B and C
public void develop();
}
interface B extends A { //superinterface to C
public void test();
}
public class C implements B {
public void develop() {
System.out.println("Devlop the code");
}
public void test() {
System.out.println("Test the code");
}
public static void main(String[] args) {
C obj = new C();
obj.develop();
obj.test();
}
}
```

2. Interface has multiple implemetation classes.

```
package interfaces;
interface Animal {
public void noise();
}
class Lion implements Animal {
public void noise() {
System.out.println("Lion roars..");
}
```

```
}
class Dog implements Animal {
public void noise() {
System.out.println("Dog barks");
}
}
class Cat implements Animal {
public void noise() {
System.out.println("Cat Meows..");
}
}
package interfaces;
import java.util.Scanner;
class MainClass1 {
public static void makeSound(Animal a) {
a.noise();
}
public static void main(String[] args) {
Scanner scan = new Scanner(System.in);
System.out.println("Welcome to circus");
System.out.println("Press 1 for dog Sound");
System.out.println("Press 2 for lion Sound");
System.out.println("Press 3 for cat Sound");
System.out.println("Enter your choise");
int choice = scan.nextInt();
if (choice == 1) {
makeSound(new Dog());
} else if (choice == 2) {
```

```
makeSound(new Lion());
} else if (choice == 3) {
makeSound(new Cat());
} else {
System.out.println("Invalid choice");
}
}
}
```

3. Multiple interface has single implementation class

```
package interfaces;
interface P {
public void move1();
}
interface Q {
public void move2();
}
interface R {
public void move3();
}
class Demo2 implements P, Q, R {
public void move1() {
System.out.println("Move car..");
}
public void move2() {
System.out.println("move cycle..");
}
public void move3() {
System.out.println("move bike..");
}
}
package interfaces;
class MainClass2 {
public static void main(String[] args) {
Demo2 d2 = new Demo2();
d2.move1();
d2.move2();
d2.move3();
}
}
```

4. Multiple interface inherits to single interface.

```
package interfaces;
interface M {
public void walk1();
}
interface N {
public void walk2();
}
interface O {
public void walk3();
}
interface T extends M,N,O {
public void walk4();
}
class Demo4 implements T {
public void walk1() {
System.out.println("Walk for 1 km");
}
public void walk2() {
System.out.println("Walk for 2 kms");
}
public void walk3() {
System.out.println("Walk for 3 kms");
}
public void walk4() {
System.out.println("Walk for 4 kms");
}
}
package interfaces;
class MainClass3 {
public static void main(String[] args) {
Demo4 d4 = new Demo4();
d4.walk1();
d4.walk2();
d4.walk3();
d4.walk4();
}
}
```

5. Single class inherits from superclass as well as implements interface.

```
package interfaces;
interface Sample1 {
public void work();
}
class Demo5 {
public void getUp() {
```

```
System.out.println("getup early");
}
}
class Run1 extends Demo5 implements Sample1 {
public void work() {
System.out.println("developing code");
}
}
package interfaces;
class MainClass4 {
public static void main(String[] args) {
Run1 r1 = new Run1();
r1.getUp();
r1.work();
}
}
```

Class Body and member declarations: A class body may contain declarations of members of the class, that is, fields, methods, classes, and interfaces.

→ A class body may also contain instance initializers, static initializers, and declarations of constructors for the class.
→ Member variables in a class—these are called fields.

```
class ClassDeclaration {
     //Fields(member declarations)
//constructor
//methods
//classes
//interfaces
//instance initializers
//static initializers
}
```

Class members: The members of a class type are all of the following:

- ❑ Members inherited from its direct superclass, except in class Object, which has no direct superclass.
- ❑ Members inherited from any direct superinterfaces.
- ❑ Members declared in the body of the class.

Members of a class that are declared private are not inherited by subclasses of that class. Only members of a class that are declared protected or public are inherited by subclasses declared in a package other than the one in which the class is declared.

Constructors, static initializers, and instance initializers are not members and therefore are not inherited.

Field Declarations: A Java field is a variable inside a class. For instance, in a class representing an employee, the Employee class might contain the following fields:

- → name
- → position
- → salary
- → HiredDate

```
public class Employee {

 String name;

 String position;

 int      salary;

 Date     hiredDate;

}
```

Field modifiers: The Java field access modifier determines whether the field can be accessed by classes other than the the class owning the field. There are some possible access modifiers for Java fields:

- → Annotation
- → public
- → private
- → protected

→ default
→ Static
→ final
→ Transient

```
public class Customer {
private   String email;
String position;
protected String name;
public String city;
}
```

We would probably not use all access modifiers in the same class. Most often you use private and protected. For simple, data carrying classes we may declare all fields public.

Static fields: A static field belongs to the class. Thus, no matter how many objects you create of that class, there will only exist one field located in the class, and the value of that field is the same, no matter from which object it is accessed. Static Java fields are located in the class, not in the instances of the class.

```
public class Customer {
    public static String staticField1 = "static value";
    public static void staticMethod() {
     System.out.println("Static method");
    }
    public static void main(String[] args) {
     System.out.println(staticField1);
     staticMethod();
     }
     }
```

Static fields are located in the class, so you don't need an instance of the class to access static fields. You just write the class name in front, to access them in another class.

Instance Fields: Non-static fields, on the other hand, are located in the instances of the class. Each instance of the class can have its own values for these fields. Non-static Java fields are located in the instances of the class.

```
public class Customer1 {
    public String instanceField1 =  "instance value";
    public void instanceMethod() {
     System.out.println("Non-static method");
    }
    public static void main(String[] args) {
     Customer1 obj = new Customer1();
System.out.println(obj.instanceField1);
obj.instanceMethod();
}
}
```

To access a non-static field you need an instance of the class (an object) on which you can access it.

Final Fields: A field can be declared final. A final field cannot have its value changed, once assigned. You declare a field to be final by adding the final keyword to the field declaration.

```
public class Customer2 {
    public final String finalField1 =  "Fixed value";
    public static final String finalField2 = "Fixed value";
    public final void finalMethod() {
     System.out.println("Fixed method");
    }
    public static void main(String[] args) {
     Customer2 obj = new Customer2();
     System.out.println(obj.finalField1);
     System.out.println(obj.finalField1);
```

```
          obj.finalMethod();
          }
          }
```

The value of the field1 field cannot be changed now. That means, that even if the field belongs to objects (class instances), you cannot vary the value of the field from object to object.

When you cannot change the value of a final field anyways, in many cases it makes sense to also declare it static. That way it only exists in the class, not in every object too.

Transient fields: Transient in Java is used to indicate that a field should not be part of the serialization process.

The modifier Transient can be applied to member variables of a class to turn off serialization on these member variables. Every field that is marked as transient will not be serialized.

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

public class Employee1 implements Serializable{
 private int id;
 private String name;
 transient int age;//Now it will not be serialized
 public Employee1(int id, String name, int age) {
  this.id = id;
  this.name = name;
  this.age=age;
 }
 public static void main(String args[])throws Exception{
 Employee1 s1 =new Employee1(211,"Name",22);
      //creating object
 FileOutputStream f=new FileOutputStream("f.txt");
 ObjectOutputStream out=new ObjectOutputStream(f);
 out.writeObject(s1);
 out.flush();
 out.close();
 f.close();
 System.out.println("success");
      }
}
```

Volatile fields: Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile keyword does not cache the value of the variable and always read the variable from the main memory. The volatile keyword cannot be used with classes or methods. However, it is used with variables. It also guarantees visibility and ordering. It prevents the compiler from the reordering of code.

```
package classes;
 public class VolatileExp {
  private static  volatile boolean flag = false;
  public static void main(String[] args) {
    // Thread-1
    new Thread(new Runnable(){
      @Override
      public void run() {
        for (int i = 1; i <= 20; i++){
          System.out.println("value - " + i);
        }
        // changing status flag
        flag = true;
   System.out.println("status flag changed " + flag );
      }
    }).start();
    // Thread-2
    new Thread(new Runnable(){
      @Override
      public void run() {
        int i = 1;
        while (!flag){
          i++;
        }
  System.out.println("Start other processing " + i);
      }
    }).start();
  }
}
```

Field initializations: class declares nine fields of types boolean, byte, char, double, float, int, long, short, and String. When SomeClass is loaded, each field's bits are set to zero, which you interpret as follows `false, 0, \u0000, 0.0, 0.0, 0, 0, 0, null.`

The previous class fields were implicitly initialized to zero. However, you can also explicitly initialize class fields by directly assigning values to them, as shown in below

```
public class Program {
// global variables
public static int x = 10; // static initialization
public double y;
public boolean z;
public Program(double y, boolean z) {
```

```
this.y = y;
this.z = z;
}
public void test() {
// local variables
int a = 15; // local intialization
double b;
b = 4.5;
System.out.println(a + b);
}
public static void main(String[] args) {
Program p = new Program(3.3, true);//initialization through
constructor
System.out.println(x);
System.out.println(p.y);
System.out.println(p.z);
p.test();

}
}
```

Forward References During Field Initialization: Use of class variables whose declarations appear textually after the use is sometimes restricted, even though these class variables are in scope. Specifically, it is a compile-time error if all of the following are true:

1. The declaration of a class variable in a class or interface C appears textually after a use of the class variable.

```
public class MyClass1 {
public static boolean staticVar1 = false;
static {
 if (!staticVar1) { //Here we are changing the
//variable data

     System.out.println("Set staticVar1 to true");

     staticVar1 = true;

    }

}

public static void main(String[] args) {

System.out.println(staticVar1);

}

}
```

→ The usage of **staticVar1** occurs after it is defined. The most basic condition for a compiler error is violated. As you would expect, the value of **staticVar1** after initialization is **true**.

2. The use is a simple name in either a class variable initializer of C or a static initializer of C.

```
public class MyClass2 {

static {

    if (!MyClass2.staticVar2) { //changing the
                              //variable  to true

        System.out.println("Set staticVar2 to true");

        MyClass2.staticVar2 = true;

    }

}

public static boolean staticVar2; // The default value
                                   //is false

public static void main(String[] args) {

System.out.println(staticVar2);

}

}
```

→ This code compiles perfectly fine and the value of **staticVar2** at the end of initialization will be **true**.

3. The use is not on the left-hand side of an assignment.

```
public class MyClass3 {
static {
    System.out.println("Set staticVar3 to true");
    staticVar3 = true;
}
public static boolean staticVar3;
public static void main(String[] args) {
System.out.println(staticVar3);
}
}
```

→ The same as in example 2 holds true. If there is no assignment on the last line of code (or the **staticVar3** = MyClass3. **staticVar3** no-op), **staticVar3** is initialized to true.

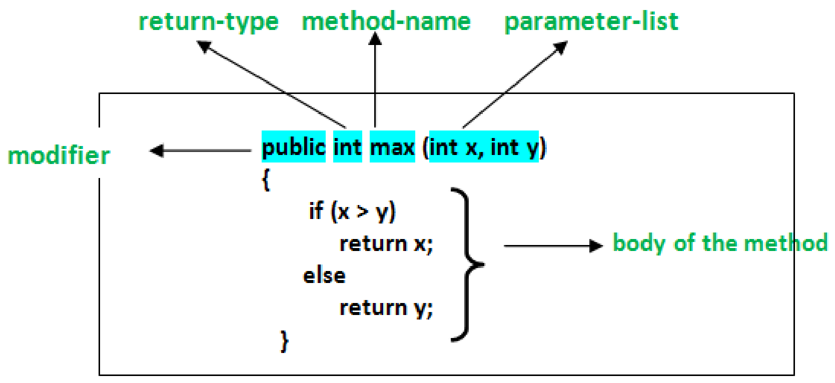
4. C is the innermost class or interface enclosing the use.

```
public class MyClass4 {
static class InnerClass {
    static {
        if (!staticVar4) {
         System.out.println("Set staticVar4 to true");
         staticVar4 = true;
        }
    }
}
public static boolean staticVar4 = false;
public static void main(String[] args) {
System.out.println(staticVar4);
}

}
```

→ since outer classes are always initialized before any nested class, the **staticVar4= false** assignment in **MyClass4** happens before the static initializer in InnerClass

Method declarations: In general, method declarations have six components:

- ❑ **Modifier-**: Defines **access type** of the method i.e., from where it can be accessed in your application. In Java, there 4 types of the access specifiers.
  - public: accessible in all class in your application.
  - protected: accessible within the class in which it is defined and, in its subclasses
  - private: accessible only within the class in which it is defined.
  - default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- ❑ **The return type:** The data type of the value returned by the method or void if does not return a value.
- ❑ **Method Name:** the rules for field names apply to method names as well, but the convention is a little different.
- ❑ **Parameter list:** Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- ❑ **Exception list:** The exceptions you expect by the method can throw; you can specify these exception(s).
- ❑ **Method body:** it is enclosed between braces. The code you need to be executed to perform your intended operations.

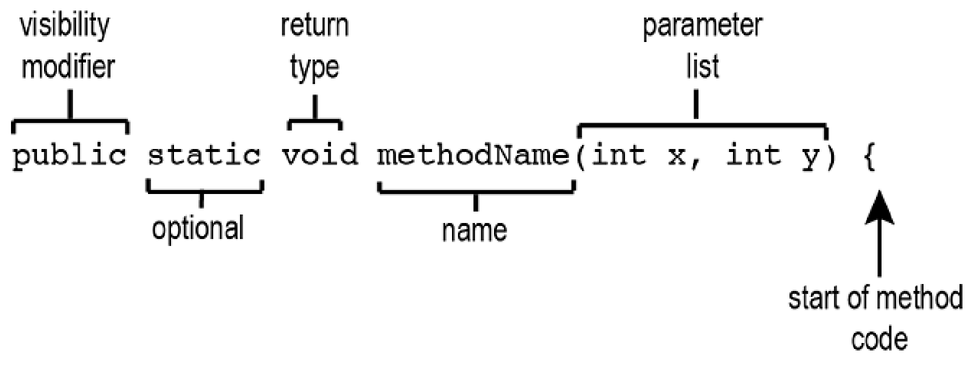


```
public int max (int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

Formal Parameters: The parameters that appear in the method definition are called formal parameters.
If a method or constructor has no formal parameters, only an empty pair of parentheses appears in the declaration of the method or constructor.

```
public class Parameters {

public void add(int a, int b) // a and b are formal
//parameters

{

int sum = a + b;

System.out.println("the sum is " + sum);

}

public static void main(String[] args) {

int num1 = 4;

int num2 = 5;

Parameters p = new Parameters();

p.add(num1, num2); // num1, num2 are actual parameters

}

}
```

Method signature: A method signature is part of the method declaration where a signature is a combination of the method name and parameter list. Method signature doesn't consider return type as part of the signature.



Method Modifiers: There a several modifiers that may be part of a method declaration:

- Access modifiers: public, protected, and private

- Modifier restricting to one instance: static
- Modifier prohibiting value modification: final
- Modifier requiring override: abstract
- Modifier preventing reentrancy: synchronized
- Modifier indicating implementation in another programming language: native
- Modifier forcing strict floating point behavior: strictfp
- Annotations.

It is a compile-time error if the same keyword appears more than once as a modifier for a method declaration.

It is a compile-time error if a method declaration that contains the keyword abstract also contains any one of the keywords private, static, final, native, strictfp, or synchronized.

It is a compile-time error if a method declaration that contains the keyword native also contains strictfp.

Abstract Methods: A method declared without a body (no implementation) within an abstract class is an **abstract method.** In other words, if you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, then you can declare the method in the parent class as an abstract.

This is how an abstract method looks in Java:

```
abstract public void test();
```

Listed below are key features of Abstract Method:

- Abstract methods don't have an implementation (body), they just have method signature as shown in the above example
- If a class has an abstract method it should be declared abstract, the vice versa is not true
- Instead of curly braces, an abstract method will have a semicolon (;) at the end
- If a regular class extends an abstract class, then the class must have to implement all the abstract methods of that class or it has to be declared abstract as well.

```
package classes;

//abstract class
abstract class Animal1 {
```

```
String Name;

Animal1(String name) {
this.Name = name;
}

// declare non-abstract methods
// it has default implementation
public void test1() {
System.out.println(this.Name);
}

// abstract methods which will be
// implemented by its subclass(es)
abstract public void test2();

abstract public void test3();
}

class Lion1 extends Animal1 {

// constructor
Lion1(String name) {
super(name);
}

@Override
public void test2() {
System.out.println("Lion hunts Animals");
}

@Override
public void test3() {
System.out.println("Lion lives in forests ");
}
}

class Tiger1 extends Animal1 {

// constructor
Tiger1(String name) {
super(name);
}
@Override
public void test2() {
System.out.println("Tiger hunts Animals");
}
```

```
@Override
public void test3() {
System.out.println("Tigers are in forests ");
}
}

class MainClass5 {
public static void main(String[] args) {

// creating the Object of subclass
// and using Animal class reference.
Animal1 object1 = new Lion1("Animal");
object1.test1();
object1.test2();
object1.test3();

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

// creating the Object of subclass
Animal1 object2 = new Tiger1("Animal");
object2.test1();
object2.test2();
object2.test3();

}
}
```

Static methods: The static keyword is used to create methods that will exist independently of any instances created for the class.
Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.
Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.
class StaticMethodExp {
int a; //initialized to zero
static int b; //initialized to zero only when class is loaded not for each object created.

```
public class StaticMethodExp {
int a; //initialized to zero
static int b;
//initialized to zero only when class is loaded not for
//each object created.
public StaticMethodExp(){
   //Constructor incrementing static variable b
   b++;
```

```
        }
         public void showData(){
            System.out.println("Value of a = "+a);
            System.out.println("Value of b = "+b);
         }
      //public static void increment(){
      //a++;
      // Cannot make a static reference to the non-
      //static field a
      //}
         public static void main(String args[]){
           StaticMethodExp s1 = new StaticMethodExp();
           s1.showData();
           StaticMethodExp s2 = new StaticMethodExp();
           s2.showData();

        }
      }
```

Final Methods: We can declare a method as final, once you declare a method final it cannot be overridden. So, you cannot modify a final method from a sub class.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

```
      package classes;

      public class FinalMethodExp {
       public final void display(){
            System.out.println("FInal method");
         }
      }
      class Sample extends FinalMethodExp{
            public void display(){
            // Cannot override the final method
             System.out.println("hi");
          }
          public static void main(String args[]){
              new FinalMethodExp().display();
           }
      }
```

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object.

Native methods: A native method in Java is used to merge the efficiency and functions of C and C++ in the Java program. When the Java compiler did not implement or support some function, then, in that case, to increase the performance of the Java application, the native methods are used.

The native method is mainly importing C code files into Java application.

Strictfp methods: strictfp is used to ensure that floating points operations give the same result on any platform. As floating points precision may vary from one platform to another. strictfp keyword ensures the consistency across the platforms.

strictfp can be applied to class, method or on interfaces but cannot be applied to abstract methods, variable or on constructors.

```
package classes;
public class Sample {
strictfp void display() {
System.out.println(10.0/3);
}
public static void main(String[] args) {
new Sample().display();
}
}
```

When we use strictfp, JVM performs floating-point computations using values that can be represented by a standard Java float or double, guaranteeing that the result of the computations will match exactly across all JVMs and platforms.

synchronized Methods: Synchronization in Java is an important concept since Java is a multi-threaded language where multiple threads run in parallel to complete program execution. In multi-threaded environment synchronization of Java object or synchronization of Java class becomes extremely important. Synchronization in Java is possible by using the Java keywords "synchronized" and "volatile".

Using synchronized keyword along with method is easy just apply synchronized keyword in front of the method. What we need to take care is that static synchronized method locked on class object lock and Non static synchronized method locks on current object (this). So it's possible that both static and nonstatic java synchronized method running in parallel.

```
package classes;
class ThreadExp extends Thread {
  public void printCount() {
try {
  for(int i = 3; i > 0; i--) {
    System.out.println("Counter --- " + i );
 }
} catch (Exception e) {
System.out.println("Thread"+
Thread.currentThread().getName()+" interrupted.");
  }
  }
```

```
     public synchronized void run() {
     printCount();
 System.out.println("Thread"+
Thread.currentThread().getName() + " exiting.");
     }
     }
     public class TestThread {
       public static void main(String args[]) {
       ThreadExp te = new ThreadExp();
       Thread t1 = new Thread(te );
       Thread t2 = new Thread(te );
       t1.start();
       t2.start();
        // wait for threads to end
       try {
       t1.join();
       t2.join();
       } catch ( Exception e) {
       System.out.println("Interrupted");
           }
        }
     }
```
Generic methods: A method is called a generic method if you can pass any type of parameter to it and it will function the same like it allows the same logic to be applied to different types. It's known as a type-safe method as well. You can declare Generic methods not only on Generic or parameterized classes but also on regular classes which not generic.

The type declaration like <T> should come after modifiers like public static but before return type like the void, int, or float.

```
     public static <T> void swap(T a, T b){
     // code
      }
```
Method results: A return statement causes the program control to transfer back to the caller of a method. Every method in Java is declared with a return type and it is mandatory for all java methods. A return type may be a primitive type like int, float, double, a reference type or void type(returns nothing).

```
package classes;

public class ReturnTest {
public int add() { // without arguments
int x = 30;
int y = 70;
int z = x + y;
return z;
}
public int sub(int x, int y) { // with arguments
int z = x - y;
return z;
}
public static void main(String[] args) {
System.out.println("The sum is " + new ReturnTest().add());
System.out.println("The sub is " + new ReturnTest().sub(30,
70));
}
}
```

Method Throws: The throws keyword is used to declare the list of exception that a method may throw during execution of program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.

```
package Exceptions;

public class Throws {
void show() throws Exception {
throw new InvalidException();
}

void show2() throws Exception {
show();
}

void show3() throws Exception {
show2();
}

public static void main(String s[]) throws Exception {
Throws o1 = new Throws();
o1.show3();
}
}
class InvalidException extends RuntimeException {
public String toString()
{
return "Dummy Exception";
}
}
```

Method Body: A method body is either a block of code that implements the method or simply a semicolon, indicating the lack of an implementation.

→ It is a compile-time error if a method declaration is either abstract or native and has a block for its body.

→ It is a compile-time error if a method declaration is neither abstract nor native and has a semicolon for its body.

```
public static void display() {
//method body
}
```

Inheritance, Overriding, and Hiding:

Inheritance: Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.

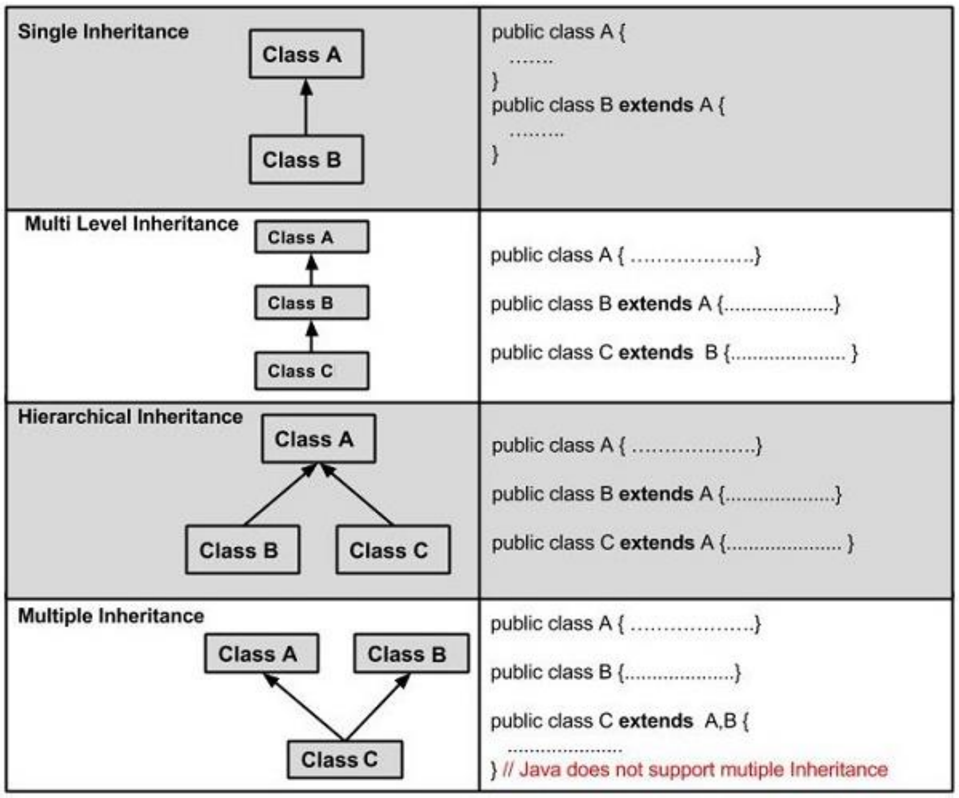
The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

```
class Super {
   .....   .....
}
class Sub extends Super {
  .....   .....
}
```

A class does not inherit static methods from its superinterfaces.

Types of Inheritance: There are various types of inheritance as demonstrated below.


Single Inheritance
Class A
Class B
public class A {
....
}
public class B extends A {
....
}
Multi Level Inheritance
Class A
Class B
Class C
public class A { ....}
public class B extends A {....................}
public class C extends B {..................... }
Hierarchical Inheritance
Class A
Class B
Class C
public class A { ....}
public class B extends A {....................}
public class C extends A {..................... }
Multiple Inheritance
Class A
Class B
Class C
public class A { ....}
public class B {....................}
public class C extends A,B {
....
} // Java does not support mutiple Inheritance

1. Single level Inheritance.

```
package inheritances;
class Shape {
public void display1() {
System.out.println("Inside display");
}
}

class Rectangle extends Shape {
public void display2() {
System.out.println("Inside area");
}
}

public class Tester {
public static void main(String[] arguments) {
Rectangle rect = new Rectangle();
rect.display1();
rect.display2();
}
}
```

2. Multi level inheritance.

```
package inheritance;
```

```
class Animal {
void eat() {
System.out.println("eating...");
}
}

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

class BabyDog extends Dog {
void weep() {
System.out.println("weeping...");
}
}

class Tester1 {
public static void main(String args[]) {
BabyDog d = new BabyDog();
d.weep();
d.bark();
d.eat();
}
}
```

3. Hierarchical inheritance:

```
package inheritance;

class Animal1 {
void eat() {
System.out.println("eating...");
}
}

class Dog1 extends Animal {
void bark() {
System.out.println("barking...");
}
}

class Cat extends Animal {
void meow() {
System.out.println("meowing...");
}
}
```

```
     class Tester3 {
     public static void main(String args[]) {
     Cat c = new Cat();
     c.meow();
     c.eat();
     //c.bark();//It shows the error
     }
     }
```
Overriding: Declaring a method in **subclass** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.
The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints the specified ringtones for each contact.
```
package overriding;

public class Contacts {
public void ringtone() {
System.out.println("Oneplus Ringtone");
}
}
public class DadContact extends Contacts {
public void ringtone() {//super class method is overrided
System.out.println("Dad ringtone");
}
}
public class MomContact extends Contacts {
public void ringtone() {//super class method is overrided
System.out.println("Mom ringtone");
}
}
public class SpecialPerson extends Contacts {
public void rintone() {// super class method is overrided
System.out.println("Special ringtone");
}
}
public class BestFriend extends Contacts {
public void ringtone() {
System.out.println("Best friend ringtone");
}
}
public class UnImpContact extends Contacts {
public void ringtone() {//super class method is overrided
   //there is no method body so superclass method body appears
}
}
public class MainClass2 {
public static void main(String[] args) {
DadContact dc = new DadContact();
MomContact mc = new MomContact();
BestFriend bf = new BestFriend();
SpecialPerson sp = new SpecialPerson();
UnImpContact uip = new UnImpContact();
dc.ringtone();
mc.ringtone();
bf.ringtone();
sp.ringtone();
uip.ringtone();
}
}
```

Hiding: Method hiding is functionally very similar to methods overriding. In overriding if you create a method in sub-class with the same type and signature in sub-class then it allows calling of methods based on the type of instance.
In the case of static methods with the same type and signature in superclass and sub-class, then the method in the subclass hides the method in the superclass.
It is a compile-time error if a static method hides an instance method.

```
package overriding;
class Parent {
public static void display1() {
System.out.println("Inside display1 method in parent class");
}
public void display2() {
System.out.println("Inside display2 method in parent class");
}
}
class Child extends Parent {
// Hiding
public static void display1() {
System.out.println("Inside display1 method in child class");
}
// Overriding
public void display2() {
System.out.println("Inside display2 method in child class");
}
}
public class MainClass1 {
public static void main(String[] args) {
Parent p = new Parent();
Parent c = new Child();
System.out.println("******Method Hiding********");
p.display1(); // This will call method in parent class
c.display1(); // This will call method in parent class
System.out.println("*****Method overriding******");
p.display2();// This will call method in parent class
c.display2(); // This will call method in child class
}
}
```

Overloading: If two or more methods may have the same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading.

```
package overloading;

class Addition {
public static void add(int i, int j) {
System.out.println(i + j);
}

public static void add(double i, double j) {
System.out.println(i + j);
}

public static void add(int i, double j) {
System.out.println(i + j);
}

public static void main(String[] args) {
add(10, 10);
add(2.2, 3.3);
add(5, 2.2);
}
}
```

If the programmer has created a method in Java to perform some action, but later as part of the requirement the programmer wants the same method to act on different parameters, then the coder can simply use the feature of **method overloading**. This will allow the programmer to create the methods of performing similar functions with the same name so that they do not have to remember the names later.

Member Type Declarations: A member class is a class whose declaration is directly enclosed in the body of another class or interface declaration.

A member interface is an interface whose declaration is directly enclosed in the body of another class or interface declaration.

It is a compile-time error if the same keyword appears more than once as a modifier for a member type declaration in a class.

A class may inherit two or more type declarations with the same name, either from two interfaces or from its superclass and an interface. It is a compile-time error to attempt to refer to any ambiguously inherited class or interface by its simple name.
Static Member Type Declarations: The static keyword may modify the declaration of a member type C within the body of a non-inner class or interface T. Its effect is to declare that C is not an inner class. Just as a static method of T has no current instance of T in its body, C also has no current instance of T, nor does it have any lexically enclosing instances.
Static is a keyword that is used for memory management mainly. Static means single copy storage for variables or methods. The members that are declared with the static keyword inside a class are called static members in java.
These members can be accessed even if no instance of the class exists because static members are not tied to a particular instance. They are shared across all instances of the class.
Instance Initializers: Instance initializer block works are used to initialize the properties of an object. It is invoked before the constructor is invoked. It is invoked every time an object is created.

```
package classes;

public class InstanceBlock {
{
System.out.println("Inside instance initializer block");
}

InstanceBlock() {
System.out.println("Inside constructor");
}

public static void main(String[] arguments) {
InstanceBlock test = new InstanceBlock();
InstanceBlock test1 = new InstanceBlock();
}
}
```

There are mainly three rules for the instance initializer block. They are as follows:
`

→ The instance initializer block is created when instance of the class is created.
→ The instance initializer block is invoked after the parent class constructor is invoked (i.e., after super() constructor call).
→ The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super ():

```
package classes;
```

```
public class Show{
Show(){
System.out.println("parent class constructor ");
}
}
public class Display extends Show{
Display(){
super();
System.out.println("child class constructor ");
}

{System.out.println("instance initializer ");}

public static void main(String args[]){
Display b=new Display();
}
}
```

Static Initializers: Instance variables are initialized using initialization blocks. However, the static initialization blocks can only initialize the static instance variables. These blocks are only executed once when the class is loaded. There can be multiple static initialization blocks in a class that is called in the order they appear in the program.

```
package classes;
public class StaticBlock {
static double percentage;
static int rank;
static {
percentage = 44.6;
rank = 12;
System.out.println("STATIC BLOCK");
}
public static void main(String args[]) {
StaticBlock st = new StaticBlock();
System.out.println("RANK: " + rank);
}
}
```

The following is an example of multiple static blocks:

```
package classes;

public class StaticBlock2 {
static {
System.out.println("FIRST STATIC BLOCK");
}
static {
System.out.println("SECOND STATIC BLOCK");
}
static {
System.out.println("THIRD STATIC BLOCK");
}

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

Constructor Declarations: The main purpose of constructors is to set the initial state of an object, when the object is created by using the new operator. Constructor declarations are very much like method declarations. However, the following restrictions on constructors should be noted:

→ Modifiers other than an accessibility modifier are not permitted in the constructor header. For accessibility modifiers for constructors.

→ Constructors cannot return a value and, therefore, do not specify a return type, not even void, in the constructor header. But their declaration can use the return statement that does not return a value in the constructor body.

→ The constructor name must be the same as the class name.

```
accessibility_modifier class_name (formal_parameter_list)
throws_clause  // Constructor header{
       // Constructor body
       //   local_variable_declarations
       //   Statements
}
```

For example, a constructor for the PairOfDice class could provide the values that are initially showing on the dice. Here is what the class would look like in that case

```
package classes;
public class PairOfDice {
public int die1;
public int die2;
public PairOfDice(int val1, int val2) {
// Constructor. Creates a pair of dice that
// are initially showing the values val1 and val2.
die1 = val1; // Assign specified values
die2 = val2; // to the instance variables.
}
public void roll() {
// Roll the dice by setting each of the dice to be
// a random number between 1 and 6.
die1 = (int) (Math.random() * 6) + 1;
die2 = (int) (Math.random() * 6) + 1;
System.out.println("First Die " + die1);
System.out.println("Second die " + die2);
}
public static void main(String[] args) {
PairOfDice pd = new PairOfDice(3, 5);
pd.roll();
}
}
```

Formal Parameters: The formal parameters in a constructor declaration specify a list of variables to which values are assigned when the constructor is called. Within the block that contains the implementation of the constructor, the constructor's formal parameters are treated as local variables; the name of each formal parameter is available as an identifier in the constructor's implementation. Formal parameters differ from local variables only in that their declaration and value come from outside the constructor's block.

The formal parameters of a constructor are identical in syntax and semantics to those of a method.

The constructor of a non-private inner member class implicitly declares, as the first formal parameter, a variable representing the immediately enclosing instance of the class.

```
accessibility_modifier class_name (formal_parameter_list)
throws_clause  // Constructor header{
       // Constructor body
       //   local_variable_declarations
       //   Statements}
```

Constructor Signature: Basically your constructor should have 3 parts.

- → Name of the constructor, which must be the same as the name of a class in which the constructor is located.
- → Access modifier-private, public or protected.
- → Parameters which can be optional.

It is a compile-time error to declare two constructors with override-equivalent signatures in a class.

It is a compile-time error to declare two constructors whose signatures have the same erasure in a class.

Constructor Modifier: ConstructorModifier Is one of Annotation public protected private.

It is a compile-time error if the same keyword appears more than once as a modifier in a constructor declaration.

A constructor cannot be abstract, static, final, native, strictfp, or synchronized:

- → A constructor is not inherited, so there is no need to declare it final.
- → An abstract constructor could never be implemented.
- → A constructor is always invoked with respect to an object, so it makes no sense for a constructor to be static.

→ There is no practical need for a constructor to be synchronized, because it would lock the object under construction, which is normally not made available to other threads until all constructors for the object have completed their work.

→ The lack of native constructors is an arbitrary language design choice that makes it easy for an implementation of the Java Virtual Machine to verify that superclass constructors are always properly invoked during object creation.

→ The inability to declare a constructor as strictfp (in contrast to a method)is an intentional language design choice; it effectively ensures that a constructor is FP-strict if and only if its class is FP-strict.

Generic Constructors: A constructor is *generic* if it declares one or more type variables.

These type variables are known as the type parameters of the constructor. The form of the type parameter section of a generic constructor is identical to the type parameter section of a generic class.

```
package classes;
public class Employee3 {
String data;
public <T> Employee3(T data) {
this.data = data.toString();
}
public void dsplay() {
System.out.println("value: " + this.data);
}
public static void main(String args[]) {
Employee3 emp1 = new Employee3("Name");
emp1.dsplay();
Employee3 emp2 = new Employee3(12548);
emp2.dsplay();

}
}
```

It is possible for a constructor to be generic independently of whether the class the constructor is declared in is itself generic.

Constructor Throws: Just like methods you can throw exceptions from constructors in. But, if you do so, you need to catch/throw (handle) the exception at the method where you invoke the constructor. If you don't a compile time error is generated.

```
package classes;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
public class Employee4 {
```

```
private String name;
private int age;
File empFile;
Employee4(String name, int age, String empFile) throws
IOException {
this.name = name;
this.age = age;
this.empFile = new File(empFile);
new FileWriter(empFile).write("Employee name is " +
name + "and age is " + age);
}
public void display() {
System.out.println("Name: " + name);
System.out.println("Age: " + age);
}
public static void main(String args[]) {
String filePath = "samplefile.txt";
Employee4 emp = null;
try {
emp = new Employee4("Krishna", 25, filePath);
} catch (IOException ex) {
System.out.println("Specified file not found");
}
emp.display();
}
}
```

The Type of a Constructor: The type of a constructor consists of its signature and the exception types given by its throws clause.

Constructor Body: The first statement of a constructor body may be an explicit invocation of another constructor of the same class or of the direct superclass.

It is a compile-time error for a constructor to directly or indirectly invoke itself through a series of one or more explicit constructor invocations involving this.

Explicit Constructor Invocations: Explicit constructor invocations appear as the first statement in the body of some other constructor. Typically they invoke a constructor of a superclass, although they may also invoke an alternative constructor of the current class.

Explicit constructor invocation statements are divided into two kinds:

→ Alternate constructor invocations begin with the keyword this (possibly prefaced with explicit type arguments). They are used to invoke an alternate constructor of the same class.

```
package classes;
class Test1 {
public Test1(int i) // 1st constructor
{
this(true); // calling 4th constructor
System.out.println(i);
}
public Test1(double j) // 2st constructor
{
this("String"); // calling 5th constructor
System.out.println(j);
}
public Test1(char a) // 3rd constructor
{
System.out.println(a);
}
public Test1(boolean b) // 4th constructor
{
this('@'); // calling 3rd constructor
System.out.println(b);
}
public Test1(String z) // 5th constructor
{
this(25); // calling 1st constructor
System.out.println(z);
}
public static void main(String[] args) {
Test1 t1 = new Test1(10.5);
           //For 2nd Constructor
}
}
```

→ Superclass constructor invocations begin with either the keyword super (possibly prefaced with explicit type arguments) or a Primary expression or an ExpressionName. They are used to invoke a constructor of the direct superclass.

```
package classes;
class Y {
public Y(int i) {
     System.out.println("i " + i);
}
}
class Z extends Y {
public Z(double j) {
super(34);
/* Will call super class constructor and pass integer 34
arg */
     System.out.println("j " + j);
}
}
class MainClass7 {
public static void main(String[] args) {
Z z1 = new Z(10.5);
}
}
```

Constructor Overloading: The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

```
package classes;
public class Student1 {
//instance variables of the class
int id;
String name;
Student1(){
System.out.println("this a default constructor");
}
Student1(int i, String n){
id = i;
name = n;
}
public static void main(String[] args) {
//object creation
Student1 s = new Student1();
System.out.println("Default Constructor");
System.out.println("Id : "+s.id + "Name : "+s.name);

System.out.println("Parameterized Constructor");
Student1 student = new Student1(10, "David");
System.out.println("Id : "+student.id + "Name :
"+student.name);
}
}
```

Default Constructor: If a class declaration does not contain any constructor declarations, Java supplies a default constructor for the class. The default constructor is public, it takes no arguments, and it simply calls the constructor of its class's superclass that takes no arguments. The default constructor is approximately equivalent to:

```
      public MyClass() {
      super();
      }
```

Because Java creates a default constructor only for a class that does not have any explicitly declared constructors, it is possible for the superclass of that class not to have a constructor that takes no arguments.

If a class declaration does not contain a constructor declaration and its immediate superclass does not have a constructor that takes no arguments, the compiler issues an error message because the default constructor references a non-existent constructor in the superclass. The default constructor for the class Object does not contain a call to another constructor because class Object has no superclass.

```
package classes;
public class Default {
int a;
String b;
public static void main(String[] args) {
// Default constructor
// is called to create a new object
Default t = new Default();
// print the default values
System.out.println(t.a);
System.out.println(t.b);
}
}
```

Preventing Instantiation of a Class: A class can be designed to prevent code outside the class declaration from creating instances of the class by declaring at least one constructor, to prevent the creation of a default constructor, and by declaring all constructors to be private.

A public class can likewise prevent the creation of instances outside its package by declaring at least one constructor, to prevent creation of a default constructor with public access, and by declaring no constructor that is public.

As the name implies, a class is said to be singleton if it limits the number of objects of that class to one.

We can’t have more than a single object for such classes.

```
package classes;
class MySingleton {
static MySingleton instance = null;
public int x = 10;
//private constructor can't be accessed outside the class
private MySingleton() {
}
// Factory method to provide the users with instances
static public MySingleton getInstance() {
if (instance == null)
instance = new MySingleton();
return instance;
}
}
class Main {
public static void main(String args[]) {
MySingleton a = MySingleton.getInstance();
MySingleton b = MySingleton.getInstance();
}
}
```

Enum Types: A Java Enum is a special Java type used to define collections of constants. More precisely, a Java enum type is a special kind of Java class. An enum can contain constants, methods etc. Java enums were added in Java 5.
simple Java enum example:

```
public enum Level {
   HIGH,
    MEDIUM,
    LOW
}
```

The enum constants are the short names of the Months like JAN, FEB, MAR and so on. After defining the enum, the values are displayed by using a for loop as shown below:

```
package enums;

public class EnumDemo {
    public enum Months {
        JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT,
NOV, DEC
    }

    public static void main(String[] args) {
        System.out.println("The enum values:");
        System.out.println("\n");
        for (Months month_name : Months.values())
            System.out.println(month_name);
    }
}
```

Enum Constants: The body of an enum declaration may contain *enum constants*. An enum constant defines an instance of the enum type.

```
public enum Season {
WINTER,
SPRING,
SUMMER, FALL
}
```

Enum Body Declarations: In addition to enum constants, the body of an enum declaration may contain constructor and member declarations as well as instance and static initializers.

Calculator Program by using enums.

```
package enums;

public class Calculator {

    enum Operation {
```

```
    PLUS("+") {
        double apply(double a, double b) {
            return a + b;
        }
    },
    MINUS("-") {
        double apply(double a, double b) {
            return a - b;
        }
    },
    TIMES("*") {
        double apply(double a, double b) {
            return a * b;
        }
    },
    DIVIDE("/") {
        double apply(double a, double b) {
            return a / b;
        }
    };

    private final String symbol;

    Operation(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String toString() {
        return symbol;
    }

    abstract double apply(double x, double y);
}

public static void main(String[] args) {

    double x = 5.0;
    double y = 3.0;

    for (Operation op : Operation.values()) {
        System.out.print(x + " ");
        System.out.print(op.toString() + " ");
        System.out.print(y + " = ");
        System.out.println(op.apply(x, y));
        }
    }
}
```

Enum nature:

1. **static final:**

→ Every enum constant is always implicitly public static final. Since it is static, we can access it by using enum Name. Since it is final, we can't create child enums.

→ We can declare main() method inside enum. Hence we can invoke enum directly from the Command Prompt.

```
package enums;

public class Enum1 {
    public enum Operation{
        ADD,// if we make it as static final we will get an
error
        SUB,
        MULTI;

        public static void main(String[] args){
           System.out.println("Addition "
+Operation.ADD(10,5));
            System.out.println("Substraction
"+Operation.SUB(10,5));
            System.out.println("Multiplication
"+Operation.MULTI(5,4));

        }
static final int ADD(int a,int b ){
            return a+b;
        }
static final int SUB(int x, int y){
            return x-y;
        }
static final int MULTI(int a , int b){
            return a*b;

        }
       public static void main(String[] args){
           System.out.println("Addition "
+Operation.ADD(10,5));
            System.out.println("Substraction
"+Operation.SUB(10,5));
            System.out.println("Multi
"+Operation.MULTI(5,4));
}
```

```
        }
}
```

→ If we provide any modifiers for enum constants it will give compilation error.

2. data type extends java.lang.Enum:

→ All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java, so an enum cannot extend anything else.

→ toString() method is overridden in java.lang.Enum class,which returns enum constant name.

→ enum can implement many interfaces.

values () ordinal() and valueOf() methods :

→ These methods are present inside java.lang.Enum.

→ values() method can be used to return all values present inside enum.

→ Order is important in enums.By using ordinal() method, each enum constant index can be found, just like array index.

→ valueOf() method returns the enum constant of the specified string value, if exists.

```
package enums;

 enum Planet {// implicitly extends java.lang.enum
    MERCURY,
    VENUS,
    EARTH,
    MARS,
    JUPITER,
    SATURN,
    URANUS,
    NEPTUNE,
}
public class MainClass {
    public static void main(String[] args) {
        Planet arry[] = Planet.values();
        for(Planet planet : arry) {
            System.out.println(planet+ " at 
index 
"+planet.ordinal());
        }
        System.out.println(Planet.valueOf("EARTH"));
    }
}
```

3.ordinals: The java.lang.Enum.ordinal() tells about the ordinal number (it is the position in its enum declaration, where the initial constant is assigned an ordinal of zero) for the particular enum.
ordinal() method is a non-static method, it means that it is accessible with the class object only and if we try to access the object of another class it will give the error. It is a final method, cannot be overridden.

```
package                                                    enums;
enum   Planet1   {//   implicitly   extends   java.lang.enum
    MERCURY,
    VENUS,
    EARTH,
    MARS,
    JUPITER,
    SATURN,
    URANUS,
    NEPTUNE,
}

public            class            MainClass1            {
    public    static    void    main(String[]    args)    {
        System.out.println("Planet    name    :    index");
        for(Planet1     planet     :     Planet1.values()){
       System.out.println(planet+  ":  "+planet.ordinal()+"
");
        }
    }
}
```

4. problem using ordinals in comparisons

```
package                                                    enums;

enum                       Fruits                          {
    APPLE,
    MANGO,
    ORANGE,
    PINEAPPLE
}

public            class            MainClass4            {
    public     static     void     main(String     args[])
    {
        Fruits           f1,           f2,           f3;

        //  Obtain  all  ordinal  values  using  ordinal().
        System.out.println("Fruits  "+"  and  their  ordinal
values:                                                  ");
        for(Fruits       a       :       Fruits.values())
```

```
            System.out.println(a + " " + a.ordinal());

        f1 = Fruits.APPLE;
        f2 = Fruits.MANGO;
        f3 = Fruits.PINEAPPLE;

        System.out.println();

        // Demonstrate compareTo() and equals()
        if(f1.compareTo(f2) < 0)
          System.out.println(f1 + " comes before " + f2);

        if(f1.compareTo(f2) > 0)
          System.out.println(f2 + " comes before " + f1);

        if(f1.compareTo(f3) == 0)
            System.out.println(f1 + " equals " + f3);

    if(f1.compareTo(f3) == 0)
        System.out.println(f1 + " equals " + f3);

    if(f2.compareTo(f3)<0)
        System.out.println(f2+" comes before "+f3);

    if(f1.compareTo(f1) == 0)
        System.out.println(f1 + " equals " + f1)
    ;
    if(f1.compareTo(f3) < 0)
        System.out.println(f1 + " comes before " + f3);
        }
    }
```

5. how do we compare two enums values (conditional checks):

There are two ways for making comparison of enum members :

→ By using == operator

→ By using equals() method

```
    package enums;

    public class Enum3 {
        public enum Day {
            MON,
            TUE,
            WED,
            THU,
            FRI,
            SAT,
            SUN;
        }

        public    static    void    main(String[]    args)    {
            Day                d                =                null;

            //                    statement                        5.1
            //System.out.println(d.name().equals(Day.MON));
            // it throws an NUllPointerException because "d" is
    null
            //                     at                          enums
            //                    statement                        5.2
            System.out.println(d    ==    Day.valueOf("TUE"));
            //      compiles     and     it     gives     false
            //statement                                            5.3
            System.out.println(d         ==          Day.WED);
            //      compiles     and     it     gives     false
            //statement                                            5.4
            switch                     (d)                         {
            }
            // it throws an NUllPointerException because "d" is
    null
            //                     at                          enums
        }
    }
```

6. enums has default methods:

An enum contains some default methods are

- → **values:** we know that the Java Compiler automatically generates a static method for each enum, called values
- → **valuesOf:** Returns Enum Constant by passing String specified enum type with the specified name.
- → **name:** name of enum constant as declared in its enum declaration. You may use *toString()* method in preference to this method.
- → **ordinal**: This method is used to obtain an enumeration constant's position in the list of constants. This is called the ordinal value.
- → **compareTo**: This method is used to compare the ordinal value of two constants of the same enumeration. This method returns a negative integer, zero, or a positive integer based on the ordinal positions of the two instances that are being compared.

```
package enums;

enum Planet2 {// implicitly extends java.lang.enum
    MERCURY,
    VENUS,
    EARTH,
    MARS,
    JUPITER,
    SATURN,
    URANUS,
    NEPTUNE,
}

public class MainClass2 {
    public static void main(String[] args) {
        Planet2 arry[] = Planet2.values();// values()
method
        for (Planet2 planet2 : arry) {
            System.out.println(planet2 + " at index " +
planet2.ordinal());
        }

System.out.println(Planet2.valueOf("EARTH"));//values of
//method
        Planet2 planet2 = Planet2.EARTH;
        System.out.println(planet2.name());// name()
method
        System.out.println(planet2.ordinal());// ordinal()
method
        planet2.compareTo(Planet2.EARTH);//compareTo()
method
    }
}
```

7. listing or looping possible values: The static values() method of java.lang. Enum class gives an array of enum values. After getting an array of enum values iteration over the array can be done using for loop.

Below program illustrate the iteration over enum using for loop:

```
package enums;

enum Planet3 {// implicitly extends java.lang.enum
    MERCURY,
    VENUS,
    EARTH,
    MARS,
    JUPITER,
    SATURN,
    URANUS,
    NEPTUNE,
}

public class MainClass3 {
    public static void main(String[] args) {
        for (Planet3 planet3 : Planet3.values()){
            System.out.println(planet3);

        }
    }
}
```

8. enums with values. Enum with constructor parameters:

```
    package enums;

    enum TrafficSignal {
        //this will call enum constructor with one String
    //argument
        RED("wait"),
        GREEN("go"),
        ORANGE("slow down");

        private String action;

        public String getAction() {
            return this.action;
        }

        // enum constructor - can not be public or protected
        TrafficSignal(String action) {
            this.action = action;
        }
    }

    public class Enum5 {

        public static void main(String args[]) {

            TrafficSignal[] signals = TrafficSignal.values();
            for (TrafficSignal signal : signals) {
                System.out.println("name : " + signal.name() +
    " action: " + signal.getAction());
            }
        }
    }
```

9. enums can have static and instance methods.

```
    package enums;
    enum Bikes {
        //Declaring the constants of the enum
        ACTIVA, PULSER, APACHE, VESPA, DUKE;
        //Instance variable of the enum
        int i;
        //Constructor of the enum
        Bikes() {}
        //methods of the enum
        public void instanceMethod() {
            System.out.println("This is instance method");
        }
        public static void staticMethod() {
            System.out.println("This is static method");
        }
    }
    public class MainClass5{
        public static void main(String args[]) {
            Bikes bikes[] = Bikes.values();
            for(Bikes bikes1: bikes) {
                System.out.println(bikes1);
            }
            System.out.println("Value of the variable:
    "+bikes[0].i);
            bikes[0].instanceMethod();
            Bikes.staticMethod();
        }
    }
```

10. enums are final, we cannot extend enum.

```
    package enums;

    enum Vechiles {
        BIKES,
        CARS,
        BUSES,
    }
    enum Cars extends Vechiles{// compiler error:
        HONDA,
        HUNDAI,
        SUZUKI,
        KIA,
        TATA;
    }

    public class MainClass6 {
    }
```

Now, let's find out why we got our compiler error.
When we compile an enum

→ It turns the enum into a subclass of the abstract class *java.lang.Enum.*

→ It compiles the enum as a *final* class.

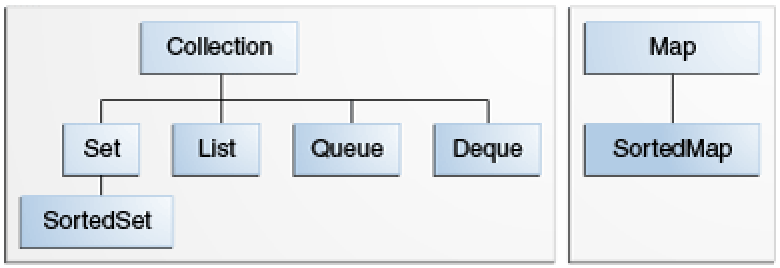
11. enums can have static and instance initializers.

```
package enums;

enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST;

    static {
        System.out.println("static Block");
    }

    {
        System.out.println("instance block");
    }
}

public class Enum6 {
    public static void main(String[] args) {
        Direction direction[] = Direction.values();

    }
}
```

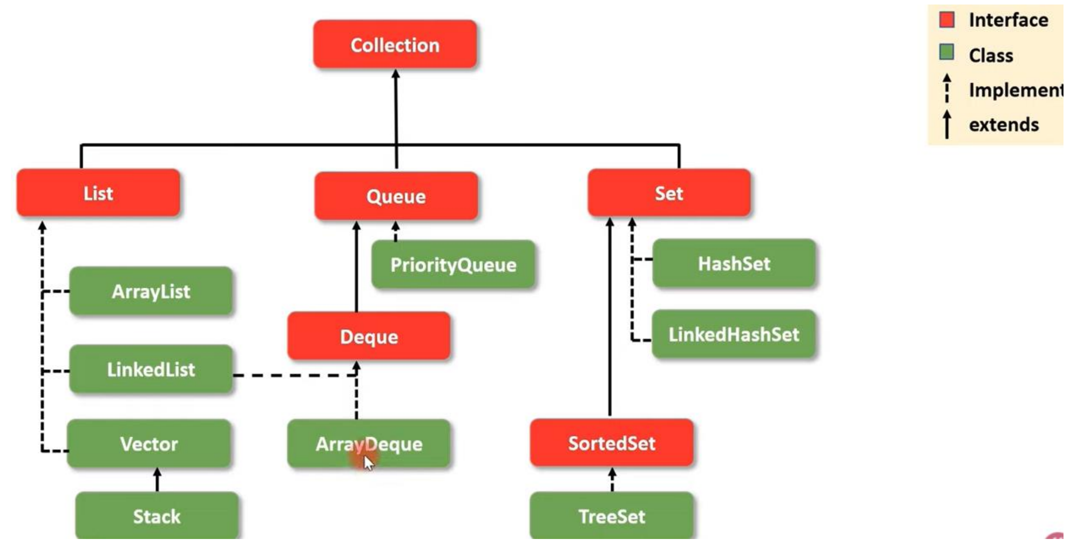
12. enum Constants with Class Bodies (enum values can have methods or implements interface).

```
package enums;

interface Year {
    public int month();
}
enum Month implements Year {
    JAN,
    FEB,
    MAR,
    APR,
    MAY; //....
    public int month() {
        return ordinal()+1;
    }

}

public class MainClass7 {
    public static void main(String[] args) {
        System.out.println("It is a month Number
"+Month.APR.month());
```

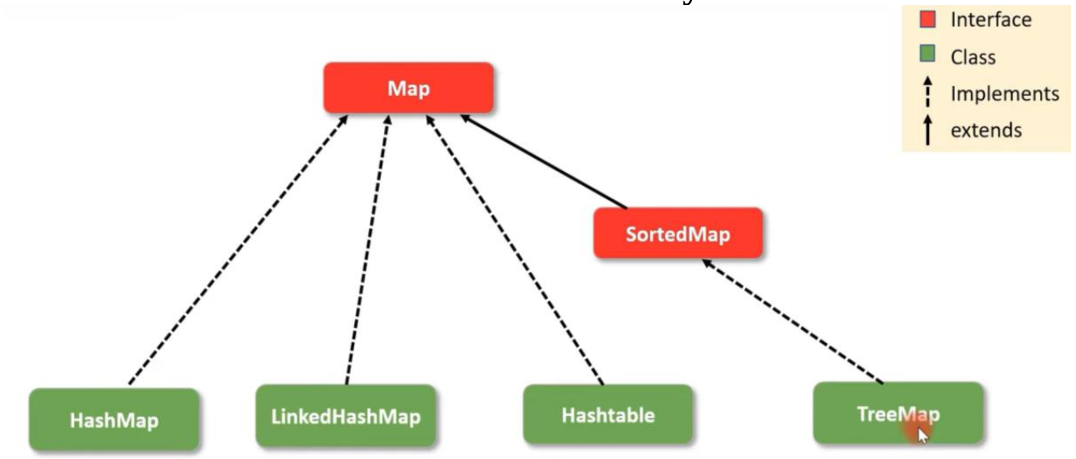
```
            }
        }
```
Collections: A collection, as name implies, is group of objects. Java Collections framework is consisting of the interfaces and classes which helps in working with different types of collections such as lists, sets, maps, stacks and queues etc.
These ready-to-use collection classes solve lots of very common problems where we need to deal with group of homogeneous as well as heterogeneous objects. The common operations in involve add, remove, update, sort, search and more complex algorithms. These collection classes provide very transparent support for all such operations using Collections APIs.
Java Collections Hierarchy: The Collections framework is better understood with the help of core interfaces. The collections classes implement these interfaces and provide concrete functionalities.



Collection interface is at the root of the hierarchy. Collection interface provides all general-purpose methods which all collections classes must support (or throw UnsupportedOperationException). It extends Iterable interface which adds support for iterating over collection elements using the "for-each loop" statement.
All other collection interfaces and classes (except Map) either extend or implement this interface. For example, List (indexed, ordered) and Set (sorted) interfaces implement this collection.
The following figures illustrates the in-detail hierarchy of the collection framework.



Collection hierarchy



Map hierarchy

Iterable interface: Iterable interface is the root interface for the entire collection framework. The collection interface extends the iterable interface. Therefore, inherently, all the interfaces and classes implement this interface. The main functionality of this interface is to provide an iterator for the collections.

Collection interface: The Collection interface is a member of the Java Collections Framework. It is a part of **java.util package.** It is one of the root interfaces of the Collection Hierarchy. The Collection interface is not directly implemented by any class. However, it is implemented indirectly via its subtypes or subinterfaces like List, Queue, and Set.

```
package                                 Collections.interfaces;

import                                      java.util.ArrayList;
import                                     java.util.Collection;

public class CollectionDemo {
    public static void main(String[] args) {
        Collection<String> bikesCollection = new
ArrayList<>();
        bikesCollection.add("Pulser");
        bikesCollection.add("Apache");
        bikesCollection.add("Activa");
        bikesCollection.add("Vespa");
     System.out.println(bikesCollection);

//                      removing                         element
        bikesCollection.remove("Activa");
        System.out.println(bikesCollection);

// checking element present or not

System.out.println(bikesCollection.contains("Apache"));

// clearing collection
        bikesCollection.clear();
        System.out.println("The collection is empty " +
bikesCollection);
    }
}
```

Output: [Pulser, Apache, Activa, Vespa]
[Pulser, Apache, Vespa]
true
The collection is empty []

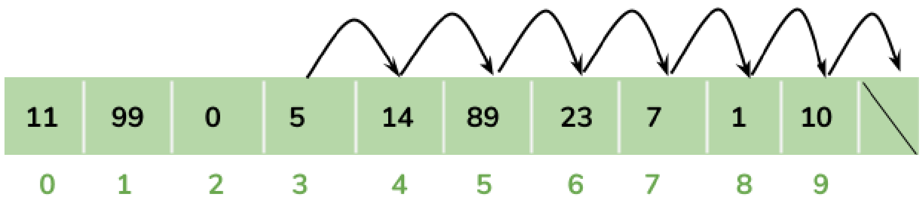
List interface: Lists represents an ordered collection of elements. Using lists, we can access elements by their integer index (position in the list), and search for elements in the list. index start with 0, just like an array. and lists types of operations like ArrayList, LinkedList, Vector, and Stack.

```
package Collections.interfaces;

import java.util.ArrayList;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        //List allows us to add duplicate elements
        list.add("element1");
        list.add("element1");
        list.add("element2");
```

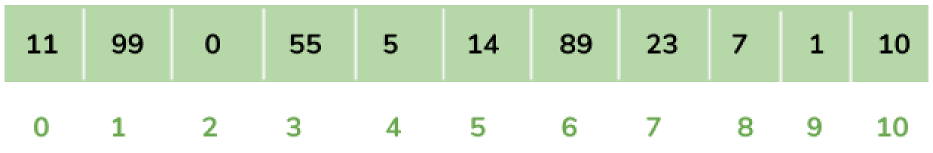
```
            list.add("element2");
            System.out.println(list);

            // list allows us to add null elements
            list.add(null);
            list.add(null);
            System.out.println(list);

            //insertion order
            list.add("ele1");
            list.add("ele2");
            list.add("ele3");
            list.add("ele5");
            list.add("ele4");
            System.out.println(list);

            //access elements from the list
            System.out.println(list.get(0));
            System.out.println(list.get(2));


        }
    }
```

Output:
```
[element1, element1, element2, element2]
[element1, element1, element2, element2, null, null]
[element1, element1, element2, element2, null, null, ele1, ele2, ele3, ele5, ele4]
element1
element2
```

ArrayList: An ArrayList in Java represents a resizable list of objects. We can add, remove, find, sort and replace elements in this list.
ArrayList is the part of the collections framework. It extends **AbstractList** which implements **List interface.** The List extends Collection and Iterable interfaces in hierarchical order.



Adding element 55 at fourth position(index 3)



ArrayList has the following features –

- → **Ordered** – Elements in arraylist preserve their ordering which is by default the order in which they were added to the list.
- → **Index based** – Elements can be randomly accessed using index positions. Index start with '0'.
- → **Dynamic resizing** – ArrayList grows dynamically when more elements need to be added than its current size.
- → **Non synchronized** – ArrayList is not synchronized, by default. Programmer needs to use synchronized keyword appropriately or simply use **Vector** class.
- → **Duplicates allowed** – We can add duplicate elements in arraylist. It is not possible in sets.

Methods of ArrayList:

1. **add(Object o)**: This method adds an object o to the arraylist.
   ```
   obj.add("hello");
   ```
2. **add(int index, Object o)**: It adds the object o to the array list at the given index.
   ```
   obj.add(2, "bye");
   ```
3. **remove(Object o)**: Removes the object o from the ArrayList.
   ```
   obj.remove("Name");
   ```
4. **remove(int index)**: Removes element from a given index.
   ```
   obj.remove(3);
   ```
5. **set(int index, Object o)**: Used for updating an element. It replaces the element present at the specified index with the object o.
   ```
   obj.set(2, "Name");
   ```
6. **int indexOf(Object o)**: Gives the index of the object o. If the element is not found in the list then this method returns the value -1.
   ```
   int pos = obj.indexOf("Name");
   ```
7. **Object get(int index)**: It returns the object of list which is present at the specified index.
   ```
   String str= obj.get(2);
   ```
8. **int size():** It gives the size of the ArrayList – Number of elements of the list.
   ```
   int numberofitems = obj.size();
   ```
9. **boolean contains(Object o)**: It checks whether the given object o is present in the array list if its there then it returns true else it returns false.
   ```
   obj.contains("Name");
   ```

10. clear(): It is used for removing all the elements of the array list in one go. The below code will remove all the elements of ArrayList whose object is obj.

```
obj.clear();
```

1. **Creating ArrayList and adding new elements to it.**

```
package Collections.arraylist;

import java.util.ArrayList;

public class ArrayListExp {
    public static void main(String[] args) {
        / /creating arrayList object
ArrayList<String> cars = new ArrayList<>();
        cars.add("Baleno");
        cars.add("Kwid");
        cars.add("Fortuner");
        cars.add("Innova");
        cars.add("Ertiga");
        System.out.println("List of cars : " + cars);

    }
}
```

Output: List of cars : [Baleno, Kwid, Fortuner, Innova, Ertiga]

2.Creating ArrayList from another collection.

```
package Collections.arraylist;

import java.util.ArrayList;
import java.util.List;

public class ArrayListExp1 {
    public static void main(String[] args) {
        //created arraylist object
        List<Integer> num = new ArrayList<>();
        num.add(2);
        num.add(3);
        num.add(5);

        List<Integer> num1 = new ArrayList<>(num);

        List<Integer> num2 = new ArrayList<>(num1);
        num2.add(7);
        num2.add(11);
        num2.add(13);
        num2.add(17);
        num2.add(19);

        num1.addAll(num2);
        System.out.println("Prime numbers are "+num2);
    }
}
```

Output: Prime numbers are [2, 3, 5, 7, 11, 13, 17, 19]

3. Accessing elements from an arrayList.

```
package Collections.arraylist;

import java.util.ArrayList;
import java.util.List;

public class ArrayListExp3 {
    public static void main(String[] args) {
        List<String> strings = new ArrayList<>();

        //check if an arrayList is empty
        System.out.println("Is empty " +
strings.isEmpty());
        strings.add("String1");
        strings.add("String2");
        strings.add("String3");
        strings.add("String4");
        strings.add("String5");

        //size of an ArrayList
    System.out.println("Strings are " + strings.size());

        //Retrieve the elements at a given index
        String string1 = strings.get(1);
   System.out.println("String at index 1 is " + string1);
  System.out.println("String at index 2 is " +
strings.get(2));

        //modify an element at given index
        strings.set(2, "Third String");
        strings.set(3, "Fourth String");
        System.out.println(strings);


    }
}
```

Output:
```
Is empty true
Strings are 5
String at index 1 is String2
String at index 2 is String3
[String1, String2, Third String, Fourth String, String5]
```

4. Removing elements from an ArrayList

```
package Collections.arraylist;

import java.util.ArrayList;

public class ArrayListExp2 {
    public    static    void    main(String[]    args)    {
ArrayList<String>    fruits    =    new    ArrayList<>();
        fruits.add("Apple");                              //0
        fruits.add("Mango");                              //1
        fruits.add("Orange");                             //2
        fruits.add("Pineapple");                          //3
        fruits.add("Grapes");                             //4
        System.out.println("All   fruits   "   +   fruits);

        //removing                                   element
        fruits.remove(2);
        fruits.remove("Mango");
        System.out.println("Fruits  after  removing  "  +
fruits);

        //removing            Multiple            elements
Arrayist<String>    fruits1    =    new    ArrayList<>();
        fruits1.add("banana");
        fruits1.add("Mango");
        fruits.removeAll(fruits1);
        System.out.println(fruits);

       //                   clearing                   all
        fruits.clear();
        System.out.println("After clearing all fruits " +
fruits);


    }
}
```

Output: All fruits [Apple, Mango, Orange, Pineapple, Grapes]
Fruits after removing [Apple, Pineapple, Grapes]
[Apple, Pineapple, Grapes]
After clearing all fruits []

5.Iterating over an arrayList

```
package                          Collections.arraylist;

import                                 java.util.Arrays;
import                               java.util.Iterator;
import                                   java.util.List;

public           class           ArrayListExp4         {
    public static void main(String[] args) {
List<String> cities = Arrays.asList("Hyderabad",
"Bangalore", "Vizag", "Mumbai", "Pune");

        //basic for loop
        System.out.println("Using for loop");
        for (int i = 0; i < cities.size(); i++) {
            System.out.println(cities.get(i));
        }

        //Enhanced          for          each          loop
        System.out.println("using    for    each    loop");
        for      (String      City      :      cities)      {
            System.out.println(City);
        }

        //basic loop with iterator
        System.out.println("using loop with iterator");
        for (Iterator iterator = cities.iterator();
iterator.hasNext(); ) {
            String city = (String) iterator.next();
            System.out.println(city);
        }

        //       iterator       with       while       loop
        System.out.println("using   iterator   with   while
loop");
        Iterator<String>   iterator   =   cities.iterator();
        while            (iterator.hasNext())            {
            String   city   =   (String)   iterator.next();
            System.out.println(city);
        }

    }
}
```
Output: Using for loop
```
Hyderabad
Bangalore
Vizag
Mumbai
Pune

using for each loop
Hyderabad
Bangalore
Vizag
Mumbai
Pune

using loop with iterator
```

```
Hyderabad
Bangalore
Vizag
Mumbai
Pune

using iterator with while loop
Hyderabad
Bangalore
Vizag
Mumbai
Pune
```

6. Reverse an ArrayList

```
package Collections.arraylist;

import java.util.ArrayList;

public class ArrayListExp6 {
    public static void main(String[] args) {
       ArrayList<String> names = new ArrayList<String>();
        names.add("Name1");
        names.add("Name2");
        names.add("Name3");
        names.add("NAme4");
        System.out.println("Before reversing " + names);

        ArrayList<String> newNames = reverseArray(names);
        System.out.println("After reversing " + newNames);


    }

    public static ArrayList<String>
reverseArray(ArrayList<String> names) {
        ArrayList<String> res = new ArrayList<String>();
        for (int i = names.size() - 1; i >= 0; i--) {
            res.add(names.get(i));
        }
        return res;
    }
}
```

Output:
```
Before reversing [Name1, Name2, Name3, NAme4]
After reversing [NAme4, Name3, Name2, Name1]
```

7. Sorting ArrayList in Asc and desc

```
package Collections.arraylist;

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

public class ArrayListExp5 {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(10);
        list.add(20);
        list.add(50);
        list.add(40);
        list.add(30);
        Collections.sort(list); //Asc order
        System.out.println("Ascending order " + list);
        Collections.reverse(list); //Desc order
        System.out.println("Descending order " + list);


    }
}
```

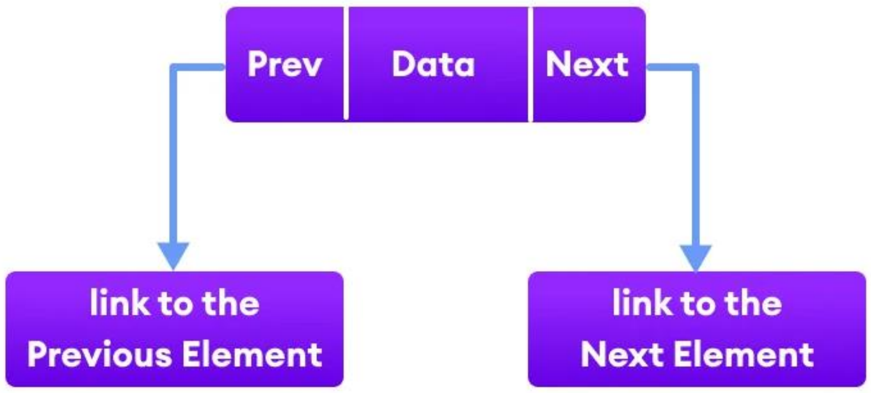
Output: Ascending order [10, 20, 30, 40, 50]
Descending order [50, 40, 30, 20, 10]

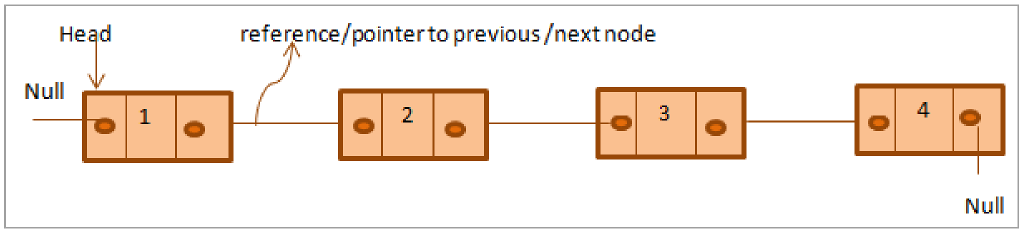
8. Sorting Employees by using ArrayList in Ascending order by using ArrayList

```
package Collections.arraylist;

public class Employee {
    private int id;
    private String name;
    private int age;
    private int salary;

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public int getSalary() {
        return                                    salary;
    }

    public  Employee(int  id,  String  name,  int  age,  int
salary)                                                     {
        this.id                   =                       id;
        this.name                 =                     name;
        this.age                  =                      age;
        this.salary               =                   salary;
    }

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

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

    public       void       setAge(int       age)         {
        this.age                  =                      age;
    }

    public      void      setSalary(int      salary)      {
        this.salary               =                   salary;
    }

    @Override
    public String toString() {

             return             "Employee{"               +
                     "id="         +         id           +
                     ",   name='"   +   name   +   '\''   +
                     ",      age="      +      age        +
                     ",     salary="     +     salary     +
                     '}';
         }
     }

package Collections.arraylist;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class SortList {
    public static void main(String[] args) {
        List<Employee> employeeList = new
ArrayList<Employee>();
        employeeList.add(new Employee(001, "Name1", 30,
600000));
        employeeList.add(new Employee(002, "Name2", 27,
400000));
        employeeList.add(new Employee(003, "Name3", 25,
350000));
        employeeList.add(new Employee(004, "Name4", 23,
300000));

        Collections.sort(employeeList, new Mysort());
//Ascending order
        System.out.println(employeeList);

    }
}

class Mysort implements Comparator<Employee> {
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.getSalary() - o2.getSalary();
    }
}
```

Output:
```
[Employee{id=4, name='Name4', age=23, salary=300000},
Employee{id=3, name='Name3', age=25, salary=350000},
Employee{id=2, name='Name2', age=27, salary=400000},
Employee{id=1, name='Name1', age=30, salary=600000}]
```

LinkedList: Similar to arrays in Java, LinkedList is a linear data structure. However LinkedList elements are not stored in contiguous locations like arrays, they are linked with each other using nodes. Each element of the LinkedList has the reference(address/node) to the next element of the LinkedList.





Each element in a linked list is known as a **node**. It consists of 3 fields:

→ **Prev** - stores an address of the previous element in the list. It is null for the first element

→ **Next** - stores an address of the next element in the list. It is null for the last element

→ **Data** - stores the actual data

Methods of LinkedList:

1. **boolean add(Object item)**: It adds the item at the end of the list.
2. **void add(int index, Object item)**: It adds an item at the given index of the the list.
3. **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null.
4. **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.
5. **void addFirst(Object item)**: It adds the item (or element) at the first position in the list.
6. **void addLast(Object item)**: It inserts the specified item at the end of the list.
7. **void clear()**: It removes all the elements of a list.
8. **Object clone()**: It returns the copy of the list.

9. **boolean contains(Object item)**: It checks whether the given item is present in the list or not. If the item is present then it returns true else false.
10. **Object get(int index)**: It returns the item of the specified index from the list.
11. **Object getFirst()**: It fetches the first item from the list.
12. **int indexOf(Object item)**: It returns the index of the specified item.
13. **int lastIndexOf(Object item)**: It returns the index of last occurrence of the specified element.
14. **Object poll()**: It returns and removes the first item of the list.
15. **Object pollFirst()**: same as poll() method. Removes the first item of the list.
16. **Object pollLast()**: It returns and removes the last element of the list.
17. **offer(E e):** This method Adds the specified element as the tail (last element) of this list.
18. **offerFirst(E e):** This method Adds the specified element as the tail (last element) of this list.
19. **offerLast(E e):** This method Inserts the specified element at the end of this list
20. **peek():** This method retrieves, but does not remove, the head (first element) of this list.
21. **peekFirst():** This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
22. **peekLast():** This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
23. **Object remove()**: It removes the first element of the list.

24. **Object remove(int index)**: It removes the item from the list which is present at the specified index.
25. **Object remove(Object obj)**: It removes the specified object from the list.
26. **Object removeFirst()**: It removes the first item from the list.
27. **Object removeLast()**: It removes the last item of the list.
28. **Object removeFirstOccurrence(Object item)**: It removes the first occurrence of the specified item.
29. **Object removeLastOccurrence(Object item)**: It removes the last occurrence of the given element.
30. **Object set(int index, Object item)**: It updates the item of specified index with the give value.
31. **int size()**: It returns the number of elements of the list.

1. **Creating LinkedList and adding elements to it.**

```
package Collections.linkedlist;
import java.util.LinkedList;
public class LinkedList1 {
    public static void main(String[] args) {
        LinkedList<String> cities = new LinkedList<>();
        //by using add methods
        cities.add("Hyderabad");
        cities.add("Bangalore");

        //by using add(position, element) method
        cities.add(2, "Mumbai");
        cities.add(3, "Pune");
        //by          using          addFirst          method
        cities.addFirst("Delhi");

        //by          using          addLast          method
        cities.addLast("Chennai");

        System.out.println("Cities " + cities);
//clone                                                   List
LinkedList<String>   cities1   =   new   LinkedList<>();
cities1       =       (LinkedList)       cities.clone();
System.out.println("Cities1                    "+cities1);

    }
}
```

Output: Cities [Delhi, Hyderabad, Bangalore, Mumbai, Pune, Chennai]

2. **Retrieving elements from a linkedList.**

```
package Collections.linkedlist;
import java.util.LinkedList;
public class LinkedList2 {
    public static void main(String[] args) {
        LinkedList<String> names = new LinkedList<>();
        names.add("Name1");
        names.add("Name2");
        names.add("Name3");
        names.add("Name4");
        names.add("Name5");

        //getting first element by using getFirst()
        String firstElement = names.getFirst();
        System.out.println("First element is " +
firstElement);

        //getting last element by using getLast()
        String lastElement = names.getLast();
        System.out.println("Last element is " +
lastElement);

        //getting the element by using the position
        String element = names.get(3);
        System.out.println("The element at index 3 is " +
element);

        //getting all the elements
        for (String elements : names) {
            System.out.println(elements);
        }
    }
}
```

Output:
```
First element is Name1
Last element is Name5
The element at index 3 is Name4
Name1
Name2
Name3
Name4
Name5
```

3. **Removing the elements from a linkedList.**

```
package Collections.linkedlist;
import java.util.LinkedList;
public class LinkedList3 {
    public static void main(String[] args) {
        LinkedList<String> strings = new LinkedList<>();
        strings.add("String1");
        strings.add("String2");
        strings.add("String3");
        strings.add("String4");
        strings.add("String5");
        System.out.println("Initial linkedList " +
strings);

        //removing 1st element
        strings.removeFirst();
        System.out.println("After removing 1st String " +
strings);
        //removing last element
        strings.removeLast();
        System.out.println("After removing last string " +
strings);
        //removing specified element
        strings.remove("String3");
        System.out.println("After removing String3 " +
strings);

        //clearing all elements
        strings.clear();
        System.out.println("Empty LinkedList " + strings);
    }
}
```

Output: Initial linkedList [String1, String2, String3, String4, String5]

After removing 1st String [String2, String3, String4, String5]

After removing last string [String2, String3, String4]

After removing String3 [String2, String4]

Empty LinkedList []

4. **Iterating and searching for element in LinkedList.**

```
package                                    Collections.linkedlist;
import                                         java.util.Iterator;
import                                       java.util.LinkedList;
public            class            LinkedList4                   {
    public    static    void    main(String[]    args)    {
        LinkedList<String> vechiles = new LinkedList<>();
        vechiles.add("Bikes");
        vechiles.add("Cars");
        vechiles.add("Buses");
        vechiles.add("Trains");
        vechiles.add("Aeroplanes");
      System.out.println("List of vechiles " + vechiles);

        //using              contains              method
        boolean    res    =    vechiles.contains("Cars");
        System.out.println("Car      "      +      res);

        //finding                                   index
        int     index     =     vechiles.indexOf("Trains");
        System.out.println("Trains  index  is  "  +  index);

        //finding                   last                  Index
     int  lastIndex  =  vechiles.lastIndexOf("Aeroplanes");
System.out.println("Aeroplanes  last  index  "  +  lastIndex);

        //using                 iterator                  method
        Iterator<String>  iterator  =  vechiles.iterator();
        while              (iterator.hasNext())              {
            String   string   =   (String)   iterator.next();
            System.out.println(string);
        }
    }
}
```

Output:
```
List of vechiles [Bikes, Cars, Buses, Trains, Aeroplanes]
Car true
Trains index is 3
Aeroplanes last index 4
Bikes
Cars
Buses
Trains
Aeroplanes
```

5. **reverses the linked list using the descendingIterator () method.**

```
package                            Collections.linkedlist;

import                                        java.util.*;
```

```
public class LinkedList5 {
    public static void main(String args[]) {
        //create a LinkedList object
        LinkedList<String> list = new LinkedList<>();
        list.add("Pune");
        list.add("Mumbai");
        list.add("Nagpur");
        System.out.println("Linked List : " + list);
     System.out.println("Linked List in reverse order:");
    //descendingIterator method to get a reverse iterator
        Iterator iter = list.descendingIterator();
        //traverse the list using iterator
        while (iter.hasNext()) {
            System.out.print(iter.next() + " ");
        }
    }
}
```

Output: Linked List : [Pune, Mumbai, Nagpur]

Linked List in reverse order:

Nagpur Mumbai Pune

6. **Sorting a LinkedList using Collections.sort () and comparator.**

```
package Collections.linkedlist;
import java.util.*;
public class LinkedList6 {
    public static void main(String args[]) {
        // create and initialize the LinkedList object
        LinkedList<String> months = new LinkedList<>();
        months.add("Jan");
        months.add("Feb");
        months.add("Mar");
        months.add("Apr");
        months.add("May");
        months.add("Jun");
        //print original unsorted linkedlist
System.out.println("Original  LinkedList(unsorted):  "  +
months);
        //   sort   LinkedList   with   Collecitons.sort()
        Collections.sort(months);
        System.out.println("LinkedList   (sorted):   "   +
months);
        //   sort   LinkedList   using   Collection.sort()
        Collections.sort(months, new Comparator<String>() {
            @Override
            public  int  compare(String  s1,  String  s2)  {
                return    s1.length()    -    s2.length();
            }
}                                                          );
    System.out.println("LinkedList  (sorted)  "  +  months);
    }
}
```

Output: Original LinkedList (unsorted): [Jan, Feb, Mar, Apr, May, Jun]

LinkedList (sorted): [Apr, Feb, Jan, Jun, Mar, May]

LinkedList (sorted): [Apr, Feb, Jan, Jun, Mar, May]

7. **Using peak and poll methods in LinkedList.**

```
package Collections.linkedlist;

import java.util.LinkedList;

public class LinkedList7 {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Kohli");
        list.add("Rahul");
        list.add("Rohit");
        list.add("Hardik");
        list.add("Bumrah");
        System.out.println("Initial List : " + list);

        //peak method
        System.out.println("top of the List : " +
list.peek());
        //peekFirst method
        System.out.println("First element of the list : "
+
list.peekFirst());
        //peekLast method
        System.out.println("Last element of the list : " +
list.peekLast());

        //poll method
        System.out.println("Removed head element : " +
list.poll());
```

```
        //pollFirst method
        System.out.println("Removed First element : " +
list.pollFirst());
        //pollLast method
        System.out.println("Removed last element : " +
list.pollLast());
        System.out.println("Final list : " + list);
    }
}
```

Output: Initial List : [Kohli, Rahul, Rohit, Hardik, Bumrah]

```
top of the List : Kohli
First element of the list : Kohli
Last element of the list : Bumrah
Removed head element : Kohli
Removed First element : Rahul
Removed last element : Bumrah
Final list : [Rohit, Hardik]
```

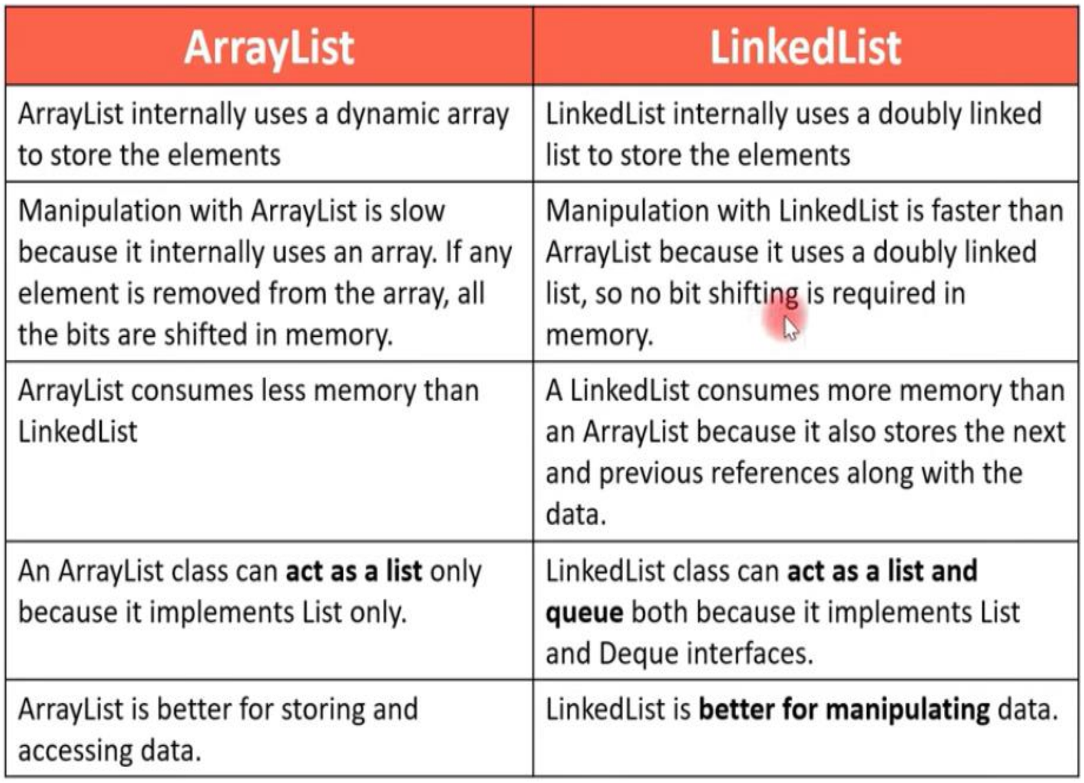
8. **Using offer methods in LinkedList.**

```
package Collections.linkedlist;

import java.util.LinkedList;

public class LinkedList8 {
    public static void main(String[] args) {
        LinkedList<String> fruits = new LinkedList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Grapes");
        fruits.add("Mango");
        System.out.println("Initial list : " + fruits);

        //offer method
        fruits.offer("Pineapple");
        //offerFirst method
        fruits.offerFirst("Pomegranate");
        //offerLast method
        fruits.offerLast("Beetroot");
        System.out.println("After using offer methods : "
+
fruits);
    }
}
```

Output:
```
Initial list : [Apple, Banana, Grapes, Mango]
After using offer methods : [Pomegranate, Apple, Banana, Grapes, Mango,
Pineapple, Beetroot]
```

Difference between ArrayList and LinkedList:

ArrayList and LinkedList both implements List interface and their methods and results are almost identical. However there are few differences between them which make one better over another depending on the requirement.



ArrayList	LinkedList
ArrayList internally uses a dynamic array to store the elements	LinkedList internally uses a doubly linked list to store the elements
Manipulation with ArrayList is slow because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
ArrayList consumes less memory than LinkedList	A LinkedList consumes more memory than an ArrayList because it also stores the next and previous references along with the data.
An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
ArrayList is better for storing and accessing data.	LinkedList is **better for manipulating** data.

When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance in LinkedList compared to ArrayList. Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist but not in LinkedList so If there are less add and remove operations and more search operations requirement, ArrayList would be your best.

Vector: Vector implements List Interface. Like ArrayList it also maintains insertion order but it is rarely used in non-thread environment as it is synchronized and due to which it gives poor performance in searching, adding, delete and update of its elements.

List of constructors provided by the vector class:

Vector( ): This constructor creates a default vector, which has an initial size of 10.

```
Vector<String> vec = new Vector<>();
```

Vector(int size): This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.

```
Vector <String> vec = new Vector<>(3);
```

Vector(int size, int incr): This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.

```
Vector <String> vec= new Vector<>(4, 6);
```

Vector(Collection c): This constructor creates a vector that contains the elements of collection c.

Methods of Vector:

1. **void addElement(Object element):** It inserts the element at the end of the Vector.
2. **int capacity():** This method returns the current capacity of the vector.
3. **int size():** It returns the current size of the vector.
4. **void setSize(int size):** It changes the existing size with the specified size.
5. **boolean contains(Object element):** This method checks whether the specified element is present in the Vector. If the element is been found it returns true else false.
6. **boolean containsAll(Collection c):** It returns true if all the elements of collection c are present in the Vector.
7. **Object elementAt(int index):** It returns the element present at the specified location in Vector.
8. **Object firstElement():** It is used for getting the first element of the vector.
9. **Object lastElement():** Returns the last element of the array.
10. **Object get(int index):** Returns the element at the specified index.
11. **boolean isEmpty():** This method returns true if Vector doesn't have any element.
12. **boolean removeElement(Object element):** Removes the specifed element from vector.
13. **boolean removeAll(Collection c):** It Removes all those elements from vector which are present in the Collection c.
14. **void setElementAt(Object element, int index):** It updates the element of specifed index with the given element.
15. **clone():** It just creates a copy of the vector.
16. **copyInto(Object[] anArray) :** it Copies the components of this vector into the specified array.

1. **Adding elements to the vector and accessing them.**

```
package                                   Collections.vector;

import                                    java.util.Iterator;
import                                      java.util.Vector;
```

```
class Vector1 {
    public static void main(String[] args) {
        Vector<String> animals = new Vector<>();

        // Using the add() method
        animals.addElement("Dog");
        animals.addElement("Horse");

        // Using index number
        animals.add(2, "Cat");
        System.out.println("List of Animals : " + animals);
        // Using addAll()
        Vector<String> animals1 = new Vector<>();
        animals1.add("Crocodile");

        animals1.addAll(animals);
        System.out.println("New list of Animals : " +
animals1);
        // Using get()
        String Animal = animals1.get(2);
        System.out.println("Animal at index 2: " + Animal);

        // Using iterator()
        Iterator<String> iterate = animals1.iterator();
        System.out.print("Animals: ");
        while (iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
    }
}
```

Output: List of Animals : [Dog, Horse, Cat]
New list of Animals : [Crocodile, Dog, Horse, Cat]
Animal at index 2: Horse
Animals: Crocodile, Dog, Horse, Cat,

2. **Obtaining sub list, removing element and clearing list by using vector.**

```
package Collections.vector;

import java.util.Vector;
import java.util.List;

public class Vector2 {

    public static void main(String[] args) {

        //Create a Vector
        Vector<String> mainList = new Vector<String>();
        //Add                                              elements
        mainList.add("Item1");
        mainList.add("Item2");
        mainList.add("Item3");
        mainList.add("Item4");
        mainList.add("Item5");
        mainList.add("Item6");
        System.out.println("Main  List  elements  :  "  +
mainList);

        // The method subList(int fromIndex, int toIndex)
        List    subList    =    mainList.subList(2,    5);

        System.out.println("Sub    list    elements    :");
        for  (int  i  =  0;  i  <  subList.size();  i++)  {
            System.out.println(subList.get(i));
        }
        //                  Using                  remove()
        String      element      =      mainList.remove(1);
        System.out.println("Removed Element: " + element);
        System.out.println("New main List: " + mainList);

        //                  Using                   clear()
        mainList.clear();
        System.out.println("main  list  after  clear:  "  +
mainList);
    }
}
```

Output: Main List elements : [Item1, Item2, Item3, Item4, Item5, Item6]
Sub list elements :
Item3
Item4
Item5
Removed Element: Item2
New main List: [Item1, Item3, Item4, Item5, Item6]
main list after clear: []

3. **max and min, capacity, hashcode of vector and insertAt methods.**

```
package Collections.vector;

import java.util.Collections;
import java.util.Vector;

public class Vector3 {
    public static void main(String[] args) {
        Vector<Integer> nums = new Vector<>();
        nums.add(3);
        nums.add(6);
        nums.add(5);
        nums.add(2);
        nums.add(9);
        nums.add(10);

        // finding min and max values
        System.out.println("Min value
:"+Collections.min(nums));
        System.out.println("Max value :"+
Collections.max(nums));

        //finding capacity of vector
        System.out.println("Capacity of vec :"+
nums.capacity());

        //finding hashcode
        System.out.println("Hash code is " +
nums.hashCode());

        // inserting 20 at the index 5
        nums.insertElementAt(20, 4);
        System.out.println("After insertion :" + nums);
    }
}
```

Output:
```
Min value : 2
Max value : 10
Capacity of vector is :10
Hash code is 979083426
After insertion :[3, 6, 5, 2, 20, 9, 10]
```

4. **Cloning and coping vector to Array.**

```
package Collections.vector;

import java.util.Vector;

public class Vector5 {
    public static void main(String[] args) {
        Vector<Integer> nums = new Vector<>();
        nums.add(10);
        nums.add(5);
        nums.add(20);
        nums.add(78);
        nums.add(07);
        nums.add(18);
        System.out.println("Original list : " + nums);
        // clone method
        Object copiedNums = (Vector) nums.clone();
        System.out.println("Copied clone : " +
copiedNums);

        Integer[] arry = new Integer[8];
        arry[0] = 5;
        arry[1] = 10;
        arry[2] = 15;
        System.out.println("Array numbers : " + arry);
        for (Integer nums1 : arry)
            System.out.println(nums1);
        //coping vector to array
        nums.copyInto(arry);
        System.out.println("final array : ");
        for (Integer nums2 : arry)
            System.out.println(nums2);
    }
}
```

```
Output: Original list : [10, 5, 20, 78, 7, 18]
Copied clone : [10, 5, 20, 78, 7, 18]
Array numbers :
5
10
15
null
null
null
null
null
final array :
10
5
20
78
7
18
null
null
```

5. **Storing employee details by using vector.**

```
package Collections.vector;
```

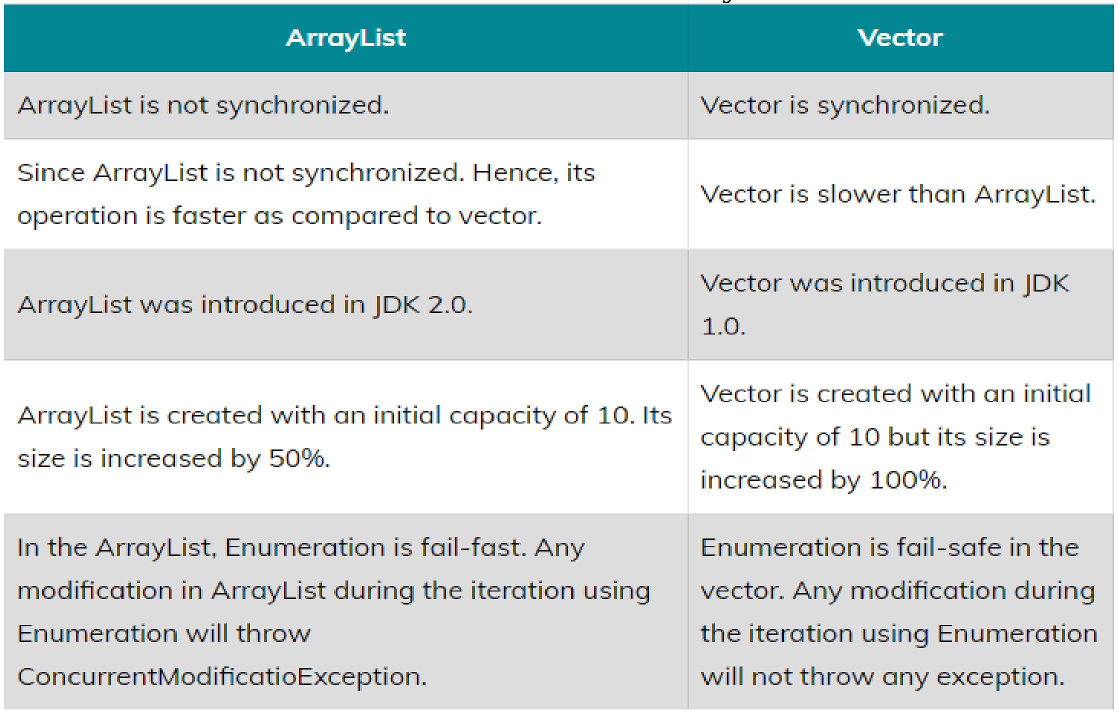
```
public class Employee {
    private int empId;
    private String empName;

    public Employee(int empId, String empName) {
        this.empId = empId;
        this.empName = empName;
    }

    public int getEmpId() {
        return empId;
    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    @Override
    public String toString() {
        return "[" +
                "empId=" + empId +
                ", empName='" + empName + '\'' +
                ']';
    }
}
package Collections.vector;

import java.util.Enumeration;
import java.util.Vector;

public class Vector4 {
    public static void main(String[] args) {
        Vector<Employee> employees = new Vector<>();
        Employee emp1 = new Employee(101, "Employee1");
        Employee emp2 = new Employee(102, "Employee2");
        Employee emp3 = new Employee(103, "Employee3");
        Employee emp4 = new Employee(104, "Employee4");
        employees.add(emp1);
        employees.add(emp2);
```

```
            employees.add(emp3);
            employees.add(emp4);

            Enumeration enumeration = employees.elements();
            System.out.println("Employee details are ");
            while (enumeration.hasMoreElements()) {
                Employee emp = (Employee)
enumeration.nextElement();
                System.out.println(emp.toString());
            }

        }
}
```

Output: Employee details are
[empId=101, empName='Employee1']
[empId=102, empName='Employee2']
[empId=103, empName='Employee3']
[empId=104, empName='Employee4']

Difference between Vector and ArrayList:

ArrayList and Vector both use Array as a data structure internally. However there are few differences in the way they store and process the data. In this post we will discuss the difference and similarities between ArrayList and Vector.



ArrayList	Vector
ArrayList is not synchronized.	Vector is synchronized.
Since ArrayList is not synchronized. Hence, its operation is faster as compared to vector.	Vector is slower than ArrayList.
ArrayList was introduced in JDK 2.0.	Vector was introduced in JDK 1.0.
ArrayList is created with an initial capacity of 10. Its size is increased by 50%.	Vector is created with an initial capacity of 10 but its size is increased by 100%.
In the ArrayList, Enumeration is fail-fast. Any modification in ArrayList during the iteration using Enumeration will throw ConcurrentModificatioException.	Enumeration is fail-safe in the vector. Any modification during the iteration using Enumeration will not throw any exception.

When to use ArrayList and when to use vector?

It totally depends on the requirement. If there is a need to perform "thread-safe" operation the vector is your best as it ensures that only one thread access the collection at a time.

Performance: Synchronized operations consumes more time compared to non-synchronized ones so if there is no need for thread safe operation, ArrayList is a better choice as performance will be improved because of the concurrent processes.

Conversions:

1. **Converting LinkedList to ArrayList.**

```
package Collections.conversions;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class LListToAList {
    public static void main(String[] args) {
     LinkedList<String> names = new LinkedList<String>();
        names.add("Harry");
        names.add("Jack");
        names.add("Tim");
        names.add("Rick");
        names.add("Rock");
     System.out.println("LinkedList elements :" + names);

        //conversion
      List<String> names1 = new ArrayList<String>(names);
        System.out.println("ArrayList elements :");
        for (String str : names1) {
            System.out.println(str);
        }
    }
}
```

Output: LinkedList elements :[Harry, Jack, Tim, Rick, Rock]
ArrayList elements :
Harry
Jack
Tim
Rick
Rock

2. **Converting Vector to ArrayList.**

```
package Collections.conversions;
```

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

public           class           VecToArryList             {

    public    static    void    main(String[]    args)    {

        //   Creating   a   Vector   of   String   elements
        Vector<String>   names   =   new   Vector<String>();
        names.add("Kohli");
        names.add("Rahul");
        names.add("Rohit");
        names.add("Jaddu");
        names.add("Siraj");
        names.add("Bumrah");

        //Displaying            Vector              elements
        System.out.println("Vector      Elements      :");
        for        (String        str        :        names){
            System.out.println(str);
        }

        //Converting       Vector       to       ArrayList
     ArrayList<String>        arraylist        =        new
ArrayList<String>(names);

        //Displaying          ArrayList           Elements
        System.out.println("ArrayList     Elements     :");
        for       (String       s       :       arraylist){
            System.out.println(s);
        }
    }
}
```

Output:
```
Vector Elements :
Kohli
Rahul
Rohit
Jaddu
Siraj
Bumrah
ArrayList Elements :
Kohli
Rahul
Rohit
Jaddu
```

Siraj
Bumrah

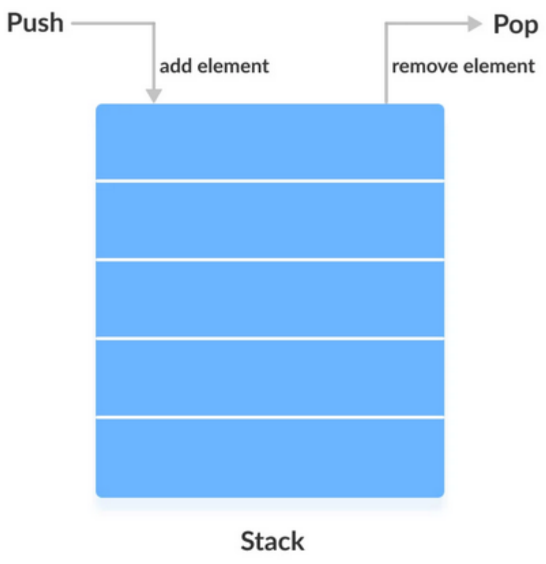
3. Converting Vector to LinkedList.

```
package Collections.conversions;

import java.util.Vector;
import java.util.List;
import java.util.Collections;
public class VecToList {

    public static void main(String[] args) {
        Vector<String> names = new Vector<String>();
        names.add("Joe Root");
        names.add("Jos Buttler");
        names.add("Sam curran");
        names.add("Ben stokes");

        // Displaying Vector elements
        System.out.println("Vector Elements :");
        for (String str : names){
            System.out.println(str);
        }

        //Converting Vector to List
List<String> names1 = Collections.list(names.elements());

        //Displaying List Elements
        System.out.println("List Elements :");
        for (String str2 : names1){
            System.out.println(str2);
        }
    }
}
```

Output:
```
Vector Elements :
Joe Root
Jos Buttler
Sam curran
Ben stokes
List Elements :
Joe Root
Jos Buttler
Sam curran
Ben stokes
```

Stack: Stack is one of the sub-class of Vector class so that all the methods of Vector are inherited into Stack. The concept of Stack of Data Structure is implemented in java and developed a pre-defined class called Stack.

Stack Implementation: In stack, elements are stored and accessed in **Last In First Out** manner. That is, elements are added to the top of the stack and removed from the top of the stack.



Stack Important Points:

→ Stack class allow to store Heterogeneous elements.
→ Stack work on Last in First out (LIFO) manner.
→ Stack allow to store duplicate values.
→ Stack class is Synchronized.
→ Initial 10 memory location is create whenever object of stack is created and it is re-sizable.
→ Stack also organizes the data in the form of cells like Vector.

Methods of Stack:

1. **public boolean empty():** is used for returns true provided Stack is empty. It returns false in case of Stack is non-empty.
2. **public void push (Object):** is used for inserting the elements into the Stack.
3. **public Object pop():** is used for removing Top Most elements from the Stack.
4. **public Object peek():** is used for retrieving Top Most element from the Stack.
5. **public int search(Object):** is used for searching an element in the Stack. If the element is found then it returns Stack relative position of that element

otherwise it returns -1, -1 indicates search is unsuccessful and element is not found.

We can use all the methods inherited from the vector class which are mentioned above in the vector class.

1. **Adding, removing and accessing elements from stack.**

```
package Collections.stack;

import java.util.Stack;

public class stack1 {
    public static void main(String[] args) {
        Stack<String> animals = new Stack<>();

        // Add elements to Stack
        animals.push("Dog");
        animals.push("Horse");
        animals.push("Cat");
        System.out.println("Stack elements: " + animals);

        // Remove element stacks
        String element = animals.pop();
        System.out.println("Removed Element: " + element);

        // Access element from the top
        String element1 = animals.peek();
        System.out.println("Element at top: " + element1);
    }
}
```

Output:
```
Stack elements: [Dog, Horse, Cat]
Removed Element: Cat
Element at top: Horse
```

2. **Searching and checking empty or not of a stack.**

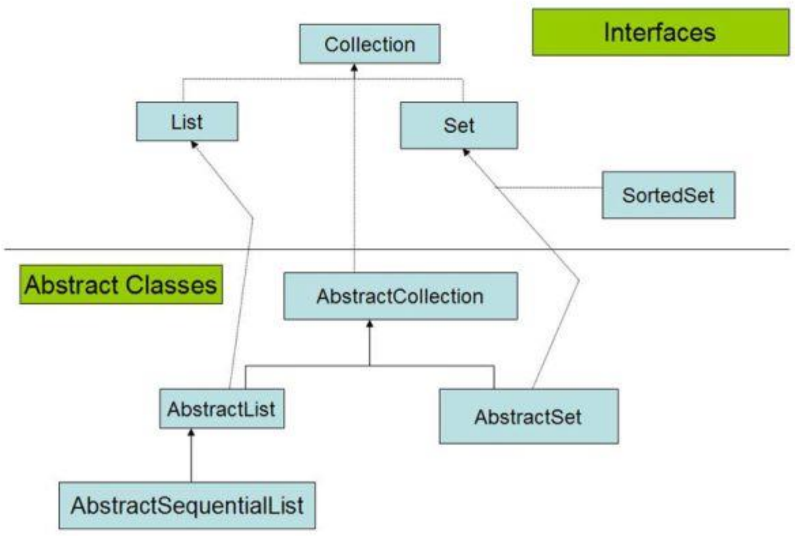
```
package Collections.stack;

import java.util.Stack;

public class Stack2 {
    public static void main(String[] args) {
        Stack<String> names = new Stack<>();

        // Add elements to Stack
        names.push("Virat");
        names.push("Rohit");
        names.push("Rahul");
        System.out.println("Stack: " + names);
        // Search an element
        int position = names.search("Rohit");
        System.out.println("Position of Rohit: " +
position);

        // Check if stack is empty
        boolean result = names.empty();
        System.out.println("Is the stack empty? " +
result);
    }
}
```

Output: Stack: [Virat, Rohit, Rahul]

Position of Rohit: 2

Is the stack empty? False

AbstractList: Abstract List is the part of the Java Collection Framework. The Abstract list is implemented by the collection interface and the Abstract Collection class. This is used when the list can not be modified. To implement this AbstractList class is used with get() and size() methods.



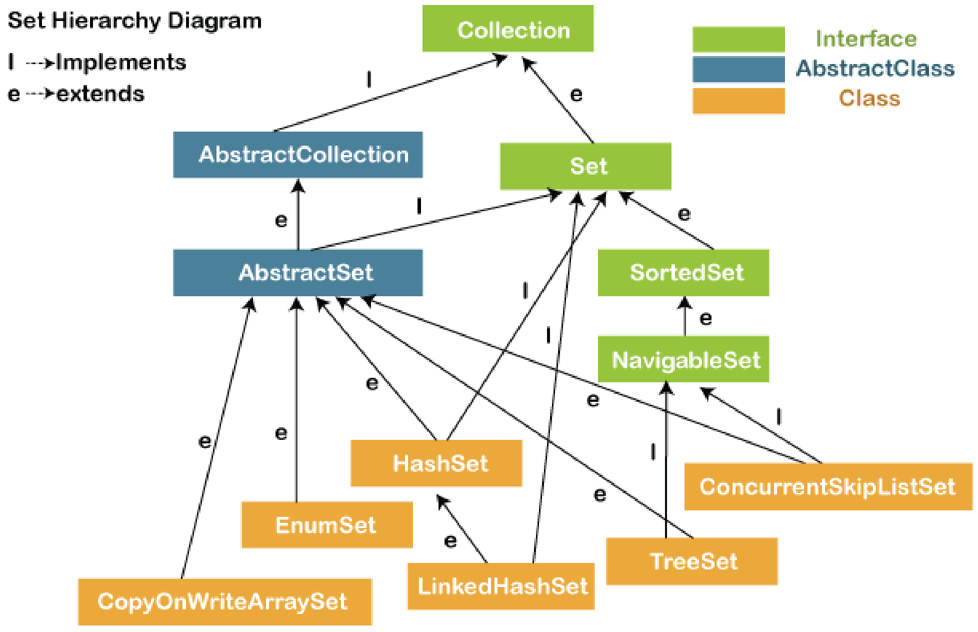
```
package                          Collections.abstractlist;

import                            java.util.AbstractList;
import                               java.util.ArrayList;

public             class             AbList1             {
    public    static    void    main(String[]    args)    {
        AbstractList<String> places = new ArrayList<>();
        places.add("Hi-Tech                        city");
        places.add("Kukatpally");
        places.add("Jubliee                       Hills");
        places.add("Banjara                       Hills");
        System.out.println("First  list  :  "  +  places);
        AbstractList<String> places1 = new ArrayList<>();
        places1.add("Hi-Tech city");
        places1.add("Kukatpally");
        places1.add("Jubliee Hills");
        places1.add("Banjara Hills");
        System.out.println("Second list : " + places);

        //comparing two lists
        boolean ab = places.equals(places1);
       System.out.println("Two lists are equal : " + ab);

        //getting the last index
        int lastindex = places.lastIndexOf("A");
        System.out.println("Last index " + lastindex);
    }
}
```

Output: First list : [Hi-Tech city, Kukatpally, Jubliee Hills, Banjara Hills]
Second list : [Hi-Tech city, Kukatpally, Jubliee Hills, Banjara Hills]
Two lists are equal : true
Last index –1

AbstractSequntialList: AbstractSequentialList class is the part of the Java Collection Framework. The Abstract Sequential list is implemented by the collection interface and the Abstract Collection class. This is used when the list can not be modified. To implement this AbstractList class is used with get() and size() methods.

```
package Collections.abstractsequentiallist;

import java.util.AbstractSequentialList;
import java.util.LinkedList;

public class AbSeqList {
    public static void main(String[] args) {
AbstractSequentialList<String> ides = new LinkedList<>();
        //adding elements
        ides.add("Ecllipse");
        ides.add("Intellij");
        ides.add("VS code");
        System.out.println("Initial list :"+ides);

        //removing element
        ides.remove(1);

        //obtaining the specified element
        System.out.println("element is "+ides.get(1));
        //isEmpty                                              method
    System.out.println("List is empty : "+ides.isEmpty());
     }
}
```
Output: Initial list :[Ecllipse, Intellij, VS code]
element is VS code
List is empty : false

Set Interface: Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.



In the above diagram, the NavigableSet and SortedSet are both the interfaces. The NavigableSet extends the SortedSet, so it will not retain the insertion order and store the data in a sorted way.

```
package                                    Collections.interfaces;

import                                        java.util.ArrayList;
import                                    java.util.LinkedHashSet;
import                                              java.util.Set;

public              class             SetDemo                   {
    public    static    void    main(String[]    args)    {
        Set<Integer>    nums    =    new    LinkedHashSet<>();
        nums.add(3);
        nums.add(10);
        nums.add(12);
        nums.add(23);
        nums.add(100);
        System.out.println("Set List : " + nums);

        ArrayList<Integer> nums1 = new ArrayList<>();
        nums1.add(11);
        nums1.add(18);
        System.out.println("Array List : " + nums1);

        //using addAll method
        nums.addAll(nums1);
        System.out.println("New List : " + nums);

        //using contains method
        System.out.println("set contains 10 : " +
nums.contains(10));

        //using containsAll method
        System.out.println("Nums contains Nums1 : " +
nums.containsAll(nums1));

        //hashcode
        System.out.println("Hashcode : " +
nums.hashCode());

        //clear method
        nums.clear();
        System.out.println("Set is clear : " + nums);

    }
}
```

Output:
```
Set List : [3, 10, 12, 23, 100]
Array List : [11, 18]
New List : [3, 10, 12, 23, 100, 11, 18]
set contains 10 : true
Nums contains Nums1 : true
Hashcode : 177
Set is clear : []
```

HashSet: This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element. This class is not synchronized.
It can be synchronized explicitly like this:

```
Set s = Collections.synchronizedSet(new HashSet(...));
```

Notice, the part new HashSet<>(8, 0.75). Here, the first parameter is **capacity**, and the second parameter is **loadFactor**.

- → **capacity** - The capacity of this hash set is 8. Meaning, it can store 8 elements.
- → **loadFactor** - The load factor of this hash set is 0.6. This means, whenever our hash set is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

By default,

- → the capacity of the hash set will be 16
- → the load factor will be 0.75

Points about HashSet:

- HashSet doesn't maintain any order, the elements would be returned in any random order.
- HashSet doesn't allow duplicates. If you try to add a duplicate element in HashSet, the old value would be overwritten.
- HashSet allows null values however if you insert more than one nulls it would still return only one null value.
- HashSet is non-synchronized.
- The iterator returned by this class is fail-fast which means iterator would throw ConcurrentModificationException if HashSet has been modified after creation of iterator, by any means except iterator's own remove method.

HashSet Methods:

1. **boolean add(Element e)**: It adds the element e to the list.
2. **void clear()**: It removes all the elements from the list.
3. **Object clone()**: This method returns a shallow copy of the HashSet.
4. **boolean contains(Object o)**: It checks whether the specified Object o is present in the list or not. If the object has been found it returns true else false.
5. **boolean isEmpty()**: Returns true if there is no element present in the Set.
6. **int size()**: It gives the number of elements of a Set.
7. **boolean(Object o)**: It removes the specified Object o from the Set.

1. **Adding duplicates and null values to HashSet.**

```
package Collections.HashSet;

import java.util.HashSet;

public class HashSet1 {
    public static void main(String[] args) {
        HashSet<String> hset = new HashSet<String>();
        // Adding elements to the HashSet
        hset.add("Apple");
        hset.add("Mango");
        hset.add("Grapes");
        hset.add("Orange");

        //Addition of duplicate elements
        hset.add("Apple");
        hset.add("Mango");

        //Addition of null values
        hset.add(null);
        hset.add(null);
        System.out.println("HashSet elements : " + hset);
    }
}
```

Output: HashSet elements : [null, Apple, Grapes, Mango, Orange]

2. **Adding, iterating and removing elements of HashSet.**

```
package Collections.HashSet;

import java.util.HashSet;
import java.util.Iterator;

public class HashSet2 {
    public static void main(String[] args) {
        HashSet<Integer> nums = new HashSet<>();
        nums.add(2);
        nums.add(6);
        nums.add(5);
        nums.add(10);
        nums.add(25);
        nums.add(15);
        System.out.println("HashSet: " + nums);

        // Calling iterator() method
        Iterator<Integer> iterate = nums.iterator();
        System.out.print("HashSet using Iterator: ");
        // Accessing elements
        while (iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }

        // Using remove() method
        boolean value1 = nums.remove(5);
        System.out.println("Is 5 removed? " + value1);
        boolean value2 = nums.removeAll(nums);
        System.out.println("Are all elements removed? " +
value2);
    }
}
```
Output: HashSet: [2, 5, 6, 25, 10, 15]
HashSet using Iterator: 2, 5, 6, 25, 10, 15,
 Is 5 removed? true
Are all elements removed? true

3. **Union and intersection of sets by using HashSet.**

```
package Collections.HashSet;

import java.util.HashSet;

public class HashSet3 {
    public static void main(String[] args) {
        HashSet<Integer> evenNums = new HashSet<>();
        evenNums.add(2);
        evenNums.add(4);
        evenNums.add(6);
        evenNums.add(8);
        evenNums.add(10);
        System.out.println("Even numbers : " + evenNums);

        HashSet<Integer> oddNums = new HashSet<>();
        oddNums.add(1);
        oddNums.add(3);
        oddNums.add(5);
        oddNums.add(7);
        oddNums.add(9);
        System.out.println("Odd Numbers : " + oddNums);

        //union of two sets
        evenNums.addAll(oddNums);
        System.out.println("Numbers after union : " +
evenNums);

        //intersection of two sets
        evenNums.retainAll(oddNums);
        System.out.println("Numbers after intersection : "
+ evenNums);

    }
}
```
Output: Even numbers : [2, 4, 6, 8, 10]
Odd Numbers : [1, 3, 5, 7, 9]

```
Numbers after union : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Numbers after intersection : [1, 3, 5, 7, 9]
```

4. **Checking difference of sets and sublist elements by using HashSet.**

```
package Collections.HashSet;

import java.util.HashSet;

public class HashSet4 {
    public static void main(String[] args) {
        HashSet<Integer> primeNums = new HashSet<>();
        primeNums.add(2);
        primeNums.add(3);
        primeNums.add(5);
        primeNums.add(7);
        primeNums.add(11);
        primeNums.add(13);
        System.out.println("Prime numbers : " +
primeNums);

        HashSet<Integer> oddNums = new HashSet<>();
        oddNums.add(1);
        oddNums.add(3);
        oddNums.add(5);
        oddNums.add(7);
        oddNums.add(9);
        System.out.println("odd numbers: " + oddNums);

        // Difference between Set1 and Set2
        primeNums.removeAll(oddNums);
        System.out.println("Difference : " + primeNums);

        //adding both sets
        primeNums.addAll(oddNums);
        System.out.println("Final set : " + primeNums);

        //checking sublist elements
        boolean result = primeNums.containsAll(oddNums);
        System.out.println("Is oddNums is subset of
primeNums? " + result);

    }
}
```

Output:
```
Prime numbers : [2, 3, 5, 7, 11, 13]
odd numbers: [1, 3, 5, 7, 9]
Difference : [2, 11, 13]
Final set : [1, 2, 3, 5, 7, 9, 11, 13]
```

Is oddNums is subset of primeNums? True

LinkedHashSet: LinkedHashSet is also an implementation of Set interface, it is similar to the HashSet and TreeSet except the below mentioned differences:

→ HashSet doesn't maintain any kind of order of its elements.
→ TreeSet sorts the elements in ascending order.
→ LinkedHashSet maintains the **insertion order**. Elements gets sorted in the same sequence in which they have been added to the Set.

In the LinkedHashset we can use all the methods which are present in the HashSet but it maintains the insertion order instead of Hashing order.

1.Adding elements to LinkedHashSet to check order of storing elements.

```
package Collections.linkedhashset;

import java.util.LinkedHashSet;

public class LkdHshSet1 {
    public static void main(String[] args) {
        LinkedHashSet<String> lhset = new
LinkedHashSet<String>();

        // Adding elements to the LinkedHashSet
        lhset.add("Z");
        lhset.add("PQ");
        lhset.add("N");
        lhset.add("O");
        lhset.add("KK");
        lhset.add("FGH");
        System.out.println(lhset);

        // LinkedHashSet of Integer Type
        LinkedHashSet<Integer> lhset2 = new
LinkedHashSet<Integer>();

        // Adding elements
        lhset2.add(99);
        lhset2.add(7);
        lhset2.add(0);
        lhset2.add(67);
        lhset2.add(89);
        lhset2.add(66);
        System.out.println(lhset2);
    }
}
```

Output:
```
[Z, PQ, N, O, KK, FGH]
[99, 7, 0, 67, 89, 66]
```

If we Observe the output: Both types of LinkedHashSet have preserved the insertion order.

2. Using iterator method and accessing elements in the LinkedHashSet.

```
package Collections.linkedhashset;

import java.util.Iterator;
import java.util.LinkedHashSet;

public class LinkedHashSet2 {
    public static void main(String[] args) {
        LinkedHashSet<Integer> numbers = new LinkedHashSet<>();
        numbers.add(2);
        numbers.add(5);
        numbers.add(6);
        System.out.println("LinkedHashSet: " + numbers);

        // Calling the iterator() method
        Iterator<Integer> iterate = numbers.iterator();

        System.out.print("LinkedHashSet using Iterator: ");

        // Accessing elements
        while(iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }
        // Using the remove() method
        boolean value1 = numbers.remove(5);
        System.out.println("\nIs 5 removed? " + value1);

        boolean value2 = numbers.removeAll(numbers);
        System.out.println("Are all elements removed? " + value2);
    }
}
```

Output:
```
LinkedHashSet: [2, 5, 6]
LinkedHashSet using Iterator: 2, 5, 6,
Is 5 removed? true
Are all elements removed? true
```

SortedSet interface: The SortedSet interface of the Java Collections framework is used to store elements with some order in a set. It extends the Set interface.

In order to use the functionalities of the SortedSet interface, we need to use the TreeSet class that implements it.

Methods of SortedSet: Besides methods included in the Set interface, the SortedSet interface also includes these methods:

1. **comparator()** - returns a comparator that can be used to order elements in the set
2. **first()** - returns the first element of the set
3. **last()** - returns the last element of the set
4. **headSet(element)** - returns all the elements of the set before the specified element
5. **tailSet(element)** - returns all the elements of the set after the specified element including the specified element
6. **subSet(element1, element2)** - returns all the elements between the element1 and element2 including element1.

Implementation of the sortedSet in TreeSet:

```
package Collections.interfaces;

import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo {
    public static void main(String[] args) {
        SortedSet<Integer> numbers = new TreeSet<>();

        // Insert elements to the set
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        numbers.add(4);
        System.out.println("SortedSet: " + numbers);

        // Access the element
        int firstNumber = numbers.first();
      System.out.println("First Number: " + firstNumber);

        int lastNumber = numbers.last();
        System.out.println("Last Number: " + lastNumber);

        // Remove elements
        boolean result = numbers.remove(2);
        System.out.println("Is the number 2 removed? " +
result);

//subset elements
System.out.println("subSet : "+numbers.subSet(2, 4));
//tail                                                    element
System.out.println("Tail element : "+numbers.tailSet(3));
}
}
```

Output: SortedSet: [1, 2, 3, 4]
First Number: 1
Last Number: 4
Is the number 2 removed? True
subSet : [3]
Tail element : [3, 4]

NavigableSet Interface: The NavigableSet interface of the Java Collections framework provides the features to navigate among the set elements. It is considered as a type of SortedSet.

Methods of NavigableSet:

1. **headSet(element, booleanValue):** The headSet() method returns all the elements of a navigable set before the specified element (which is passed as an argument). The booleanValue parameter is optional. Its default value is false. If true is passed as a booleanValue, the method returns all the elements before the specified element including the specified element.
2. **tailSet(element, booleanValue):** The tailSet() method returns all the elements of a navigable set after the specified element (which is passed as an argument) including the specified element. The booleanValue parameter is optional. Its default value is true. If false is passed as a booleanValue, the method returns all the elements after the specified element without including the specified element.
3. **subSet(e1, bv1, e2, bv2):** The subSet() method returns all the elements between e1 and e2 including e1. The bv1 and bv2 are optional parameters. The default value of bv1 is true, and the default value of bv2 is false. If false is passed as bv1, the method returns all the elements between e1 and e2 without including e1. If true is passed as bv2, the method returns all the elements between e1 and e2, including e1.
4. **descendingSet()** - reverses the order of elements in a set
5. **descendingIterator()** - returns an iterator that can be used to iterate over a set in reverse order
6. **ceiling()** - returns the lowest element among those elements that are greater than or equal to the specified element
7. **floor()** - returns the greatest element among those elements that are less than or equal to the specified element
8. **higher()** - returns the lowest element among those elements that are greater than the specified element
9. **lower()** - returns the greatest element among those elements that are less than the specified element

10.pollFirst() - returns and removes the first element from the set
11.pollLast() - returns and removes the last element from the set.

1.**Using first, last, poll methods in the NavigableSet.**

```
package Collections.interfaces;

import java.util.NavigableSet;
import java.util.TreeSet;

public class NavigableSetDemo {
    public static void main(String[] args) {
        NavigableSet<Integer> numbers = new TreeSet<>();

        // Insert elements to the set
        numbers.add(5);
        numbers.add(9);
        numbers.add(2);
        numbers.add(7);
        numbers.add(6);
        System.out.println("NavigableSet: " + numbers);

        // Access the first element
        int firstElement = numbers.first();
        System.out.println("First Number: " +
firstElement);

        // Access the last element
        int lastElement = numbers.last();
        System.out.println("Last Element: " + lastElement);

        // Remove the first element
        int number1 = numbers.pollFirst();
        System.out.println("Removed First Element: " +
number1);

        // Remove the last element
        int number2 = numbers.pollLast();
        System.out.println("Removed Last Element: " +
number2);

    }
}
```

Output:
```
NavigableSet: [2, 5, 6, 7, 9]
First Number: 2
Last Element: 9
Removed First Element: 2
Removed Last Element: 9
```

TreeSet: TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn’t maintain any order. TreeSet allows null element but like HashSet it doesn’t allow. Like most of the other collection classes this class is also not synchronized.
The important points about TreeSet class are:

- → TreeSet class contains unique elements only like HashSet.
- → TreeSet class access and retrieval times are quiet fast.
- → TreeSet class doesn't allow null element.
- → TreeSet class is non synchronized.
- → TreeSet class maintains ascending order.

All the methods of the SortedSet and NavigableSet are used in TreeSet.

1. **Iterating elements in descending order by using TreeSet.**

```
package                               Collections.treeset;

import                                 java.util.Iterator;
import                                  java.util.TreeSet;

public            class            TreeSet1            {
    public    static    void    main(String[]    args)    {
        TreeSet<String>    dist    =    new    TreeSet<>();
        dist.add("Karimnagar");
        dist.add("Hyderabad");
        dist.add("Warangal");
        dist.add("Siricilla");
        System.out.println("Districts    :"    +    dist);

        //iterating       in        descending        order
        Iterator    itr    =    dist.descendingIterator();
        while              (itr.hasNext())              {
            System.out.println(itr.next());
        }
    }
}
```

Output: Districts :[Hyderabad, Karimnagar, Siricilla, Warangal]
Warangal
Siricilla
Karimnagar
Hyderabad

2. **Using headSet(), tailSet() and subSet() Methods in TreeSet.**

```
package Collections.treeset;

import java.util.TreeSet;

public class TreeSet2 {
    public static void main(String[] args) {
        TreeSet<Integer> nums = new TreeSet<>();
        nums.add(5);
        nums.add(14);
        nums.add(7);
        nums.add(22);
        nums.add(19);
        nums.add(4);
        System.out.println("Initial list : " + nums);

        // Using headSet() with default boolean value
        System.out.println("Using headSet : " +
nums.headSet(5));

        // Using headSet() with specified boolean value
        System.out.println("Using headSet : " +
nums.headSet(5,
true));

        // Using tailSet() with default boolean value
        System.out.println("Using tailSet : " +
nums.tailSet(4));

        // Using tailSet() with specified boolean value
        System.out.println("Using tailSet : " +
nums.tailSet(4,
false));

        // Using subSet() with default boolean value
        System.out.println("Using subSet : " +
nums.subSet(4,
6));

        // Using subSet() with specified boolean value
        System.out.println("Using subSet : " +
nums.subSet(4,
false, 6, true));

    }
}
```

Output:
```
Initial list : [4, 5, 7, 14, 19, 22]
Using headSet : [4]
Using headSet : [4, 5]
Using tailSet : [4, 5, 7, 14, 19, 22]
Using tailSet : [5, 7, 14, 19, 22]
Using subSet : [4, 5]
```

Using subSet : [5]

3. **Using ceiling(), floor(), higher() and lower() Methods in TreeSet.**

```
package                                      Collections.treeset;

import                                         java.util.TreeSet;

public              class              TreeSet3              {
    public    static    void    main(String[]    args)    {
        TreeSet<Integer>   numbers   =   new   TreeSet<>();
        numbers.add(7);
        numbers.add(3);
        numbers.add(15);
        numbers.add(1);
        numbers.add(22);
        System.out.println("TreeSet:    "    +    numbers);

        //                  Using                 higher()
        System.out.println("Using     higher:     "      +
numbers.higher(4));

        //                  Using                  lower()
        System.out.println("Using      lower:     "      +
numbers.lower(4));

        //                 Using                 ceiling()
        System.out.println("Using     ceiling:     "     +
numbers.ceiling(4));

        //                  Using                  floor()
        System.out.println("Using      floor:     "      +
numbers.floor(3));
    }
}
```

Output:
```
TreeSet: [1, 3, 7, 15, 22]
Using higher: 7
Using lower: 3
Using ceiling: 7
Using floor: 3
```

4. **Using comparator method in TreeSet.**

```
package                                      Collections.treeset;

import                                    java.util.Comparator;
import                                         java.util.TreeSet;

public              class              TreeSet4              {
    public    static    void    main(String[]    args)    {
        TreeSet<String>   animals   =   new   TreeSet<>(new
CustomComparator());

        animals.add("Dog");
        animals.add("Zebra");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("TreeSet:    "    +    animals);
    }

    //       Creating       a       comparator       class
    public   static   class   CustomComparator   implements
Comparator<String>                                       {

        @Override
        public int compare(String animal1, String animal2)
{
            int   value   =     animal1.compareTo(animal2);

            //  elements  are  sorted  in  reverse  order
            if          (value          >          0)     {
                return                                 -1;
            }
            else      if      (value      <      0)       {
                return                                  1;
            }
            else                                          {
                return                                  0;
            }
        }
    }
}
```

Output: TreeSet: [Zebra, Horse, Dog, Cat]

AbstractCollection: The AbstractCollection class in Java is a part of the Java Collection Framework and implements the Collection interface. It is used to implement an unmodifiable collection, for which one needs to only extend this AbstractCollection Class and implement only the iterator and the size methods.

Constructors in Java AbstractCollection:

- **protected AbstractCollection()**: The default constructor, but being protected, it doesn't allow to create an AbstractCollection object.

Methods in AbstractCollection:

1. **add(E e):** This method ensures that this collection contains the specified element (optional operation).
2. **addAll(Collection c):** This method Adds all of the elements in the specified collection to this collection (optional operation).

3. **clear():** This method removes all of the elements from this collection (optional operation).
4. **contains(Object o):** This method returns true if this collection contains the specified element.
5. **containsAll(Collection c):** This method returns true if this collection contains all of the elements in the specified collection.
6. **isEmpty():** This method returns true if this collection contains no elements.
7. **iterator():** This method returns an iterator over the elements contained in this collection.
8. **remove(Object o):** This method removes a single instance of the specified element from this collection, if it is present (optional operation).
9. **size():** This method returns the number of elements in this collection.
10. **toArray():** This method returns an array containing all of the elements in this collection.
11. **toArray(T[] a):** This method returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
12. **toString ():** This method returns a string representation of this collection.

1. **Using add, addAll and clear methods in AbstractCollection.**

```
package Collections.abstractClasses;

import java.util.AbstractCollection;
import java.util.TreeSet;

public class AbstractCollection1 {
    public static void main(String[] args) {
   AbstractCollection<Integer> nums = new
TreeSet<Integer>();
        nums.add(10);
        nums.add(32);
        nums.add(89);
        nums.add(13);
        nums.add(17);
        System.out.println("First set values : " + nums);

        AbstractCollection<Integer> nums1 = new
TreeSet<Integer>();
        nums1.add(14);
        nums1.add(18);
        System.out.println("Second set values : " + nums);

        //using addAll() method
        nums.addAll(nums1);
        System.out.println("After addAll method : " +
nums);
```

```
		//using clear method
		nums.clear();
		System.out.println("Is collection empty ? " +
	nums.isEmpty());
	}
}
```
Output: First set values : [10, 13, 17, 32, 89]
Second set values : [10, 13, 17, 32, 89]
After addAll method : [10, 13, 14, 17, 18, 32, 89]
Is collection empty ? true

AbstractSet: The AbstractSet class in Java is a part of the Java Collection Framework which implements the Collection interface and extends the AbstractCollection class. It provides a skeletal implementation of the Set interface. This class does not override any of the implementations from the AbstractCollection class, but merely adds implementations for equals() and hashCode() method.

Constructors in AbstractSet:

protected AbstractSet(): The default constructor, but being protected, it doesn't allow to create an AbstractSet object.

Methods in AbstractSet:

1. **equals (Object o):** Compares the specified object with this set for equality.
2. **hashCode():** Returns the hash code value for this set.
3. **removeAll (Collection<?> c):** Removes from this set all of its elements that are contained in the specified collection (optional operation).

1.Using equals and hashCode methods in AbstractSet.

```
package Collections.abstractClasses;

import java.util.AbstractSet;
import java.util.TreeSet;

public class AbstractSetDemo {
	public static void main(String[] args) {
		AbstractSet<Integer> nums = new
TreeSet<Integer>();
		nums.add(32);
		nums.add(45);
		nums.add(43);
		nums.add(21);
		nums.add(56);
		System.out.println("First Set values : " + nums);

		AbstractSet<Integer> nums1 = new
TreeSet<Integer>();
        nums1.add(45);
        nums1.add(23);
        nums1.add(67);
        System.out.println("Second Set values : " +
nums1);

        //using equals() method
        System.out.println("Is two sets equals ?" +
nums.equals(nums1));

        //using hashCode method
        System.out.println("Hashcode of set1 : " +
nums.hashCode());
        System.out.println("Hashcode of set2 : " +
nums1.hashCode());
    }
}
```

```
Output: First Set values : [21, 32, 43, 45, 56]
Second Set values : [23, 45, 67]
Is two sets equals ?false
Hashcode of set1 : 197
Hashcode of set2 : 135
```

EnumSet: An enumSet is a specialized type of class which implements the Set interface needed to use the enum type.

Features of enumSet:

→ An enumSet extends AbstractSet and implements SetInterface.

→ An enumSet is a Java Collections member, it is not synchronized.

→ An enumSet is a high performing set implementation that works faster than the HashSet.

→ All the elements in an enumSet must be of a single enumeration type that is specified when the set is created.

Methods of enumSet:

1. **allOf (Class elementType):** This method is used to create an enum set containing all of the elements with a specified element type.
2. **noneOf (Class elementType):** This method is used to create an empty enum set with the specified element type.
3. **copyOf (Collection c):** This method is used to create an enum set that is initialized from the specified collection.
4. **of (E e):** This method is used to create an enum set initially, containing a single specified element.
5. **range (E from, E to):** This method is used to initially create an enum set that contains the range of specified elements.

6. **clone():** This method is used to return a copy of the specific set.

```
package Collections.EnumSet;

import java.util.EnumSet;

enum Set {
    ONE,
    TWO,
    THREE,
    FOUR,
    FIVE;
}

public class EnumSet1 {
    public static void main(String[] args) {

        //creating sets
        EnumSet<Set> set1;
        EnumSet<Set> set2;
        EnumSet<Set> set3;

        //Adding elements to sets
        set1 = EnumSet.of(Set.ONE, Set.TWO, Set.THREE);
        set2 = EnumSet.complementOf(set1);
        set3 = EnumSet.allOf(Set.class);
        System.out.println("Set 1 : " + set1);
        System.out.println("Set 2 : " + set2);
        System.out.println("Set 3 : " + set3);


    }
}
```

Output:
```
Set 1 : [ONE, TWO, THREE]
Set 2 : [FOUR, FIVE]
Set 3 : [ONE, TWO, THREE, FOUR, FIVE]
```

CopyOnWriteArrayList: It is a thread-safe variant of ArrayList in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array.

The CopyOnWriteArrayList class implements following interfaces – List, RandomAccess, Cloneable and Serializable.

CopyOnWriteArrayList Features:

→ Using CopyOnWriteArrayList is costly for update operations, because each mutation creates a cloned copy of underlying array and add/update element to it.

→ It is thread-safe version of ArrayList. Each thread accessing the list sees its own version of snapshot of backing array created while initializing the iterator for this list.

→ Because it gets snapshot of underlying array while creating iterator, it does not throw ConcurrentModificationException.

→ Mutation operations on iterators (remove, set, and add) are not supported. These methods throw UnsupportedOperationException.

→ CopyOnWriteArrayList is a concurrent replacement for a synchronized List and offers better concurrency when iterations outnumber mutations.

→ It allows duplicate elements and heterogeneous Objects (use generics to get compile time errors).

→ Because it creates a new copy of array everytime iterator is created, performance is slower than ArrayList.

```
package Collections.copyonwriteonarrayset;

import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList;

public class CopyOnWriteArraySet1 {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> set = new
CopyOnWriteArrayList<String>();
        set.add("Home");
        set.add("Office");
        set.add("Restaurant");

        //add, remove operator is not supported by
CopyOnWriteArrayList iterator
        Iterator<String> failSafeIterator =
set.iterator();
        while(failSafeIterator.hasNext()) {
            System.out.printf("Read from
CopyOnWriteArrayList : %s
%n", failSafeIterator.next());
            //failSafeIterator.remove(); //not supported

        }
    }
}
```

Output: Read from CopyOnWriteArrayList : Home
Read from CopyOnWriteArrayList : Office
Read from CopyOnWriteArrayList : Restaurant

ConcurrentSkipListSet: The ConcurrentSkipListSet has elements that are sorted by default. Also, its implementation is based on the

ConcurrentSkipListMap. The ConcurrentSkipListSet class also implements the Collection interface as well as the AbstractSet class. It is a part of the Java Collection Framework.

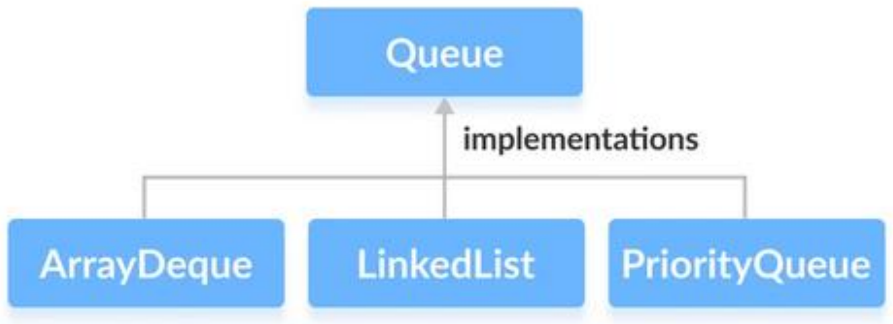
Methods of ConCurrentSkipList:

add(E e): Adds the specified element to this set if it is not already present.
clear(): Removes all of the elements from this set.
contains(Object o): Returns true if this set contains the specified element.
descendingIterator(): Returns an iterator over the elements in this set in descending order.
first(): Returns the first (lowest) element currently in this set.
isEmpty(): Returns true if this set contains no elements.
iterator(): Returns an iterator over the elements in this set in ascending order.
last(): Returns the last (highest) element currently in this set.
remove(Object o): Removes the specified element from this set if it is present.
size(): Returns the number of elements in this set.
spliterator(): Returns a Spliterator over the elements in this set.

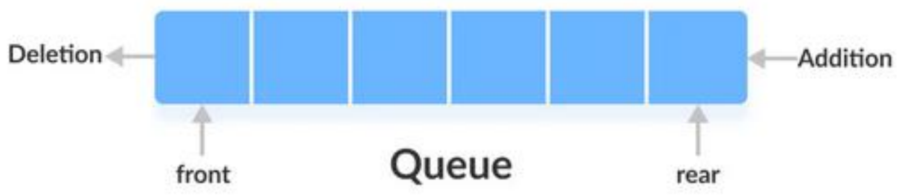
```
package Collections.concurrentskiplistset;

import java.util.concurrent.ConcurrentSkipListSet;

public class ConCurrentSkipListSet1 {
    public static void main(String[] args) {
        ConcurrentSkipListSet<Integer> nums = new
ConcurrentSkipListSet<Integer>();
        nums.add(7);
        nums.add(4);
        nums.add(1);
        nums.add(9);
        nums.add(3);
        System.out.println("Initial list : " + nums);

        //ceiling method
        int num = nums.ceiling(9);
        System.out.println("num- " + num);

        //floor method
        num = nums.floor(9);
        System.out.println("num- " + num);

        //lower method
        num = nums.lower(10);
        System.out.println("num- " + num);

    }
}
```

Output: Initial list : [1, 3, 4, 7, 9]
num- 9
num- 9
num- 9

Queue interface: The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface. Since the `Queue` is an interface, we cannot provide the direct implementation of it. In order to use the functionalities of `Queue`, we need to use classes that implement it:ArrayDeque, LinkedList and PriorityQueue.



In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front.



Methods of Queue:

1. **add()** - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
2. **offer()** - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
3. **element()** - Returns the head of the queue. Throws an exception if the queue is empty.
4. **peek()** - Returns the head of the queue. Returns null if the queue is empty.

5. **remove()** - Returns and removes the head of the queue. Throws an exception if the queue is empty.
6. **poll()** - Returns and removes the head of the queue. Returns null if the queue is empty.

1. **Implementing the Queue interface by using the LinkedList.**

```
package Collections.interfaces;

import java.util.LinkedList;
import java.util.Queue;

public class QueueDemo {
    public static void main(String[] args) {
        Queue<Integer> numbers = new LinkedList<>();

        // offer elements to the Queue
        numbers.offer(1);
        numbers.offer(3);
        numbers.offer(1);
        System.out.println("Queue: " + numbers);

        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " +
accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
 System.out.println("Removed Element: " + removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output: Queue: [1, 3, 1]

Accessed Element: 1

Removed Element: 1

Updated Queue: [3, 1]

2. **Implementing the queue interface by using the PriorityQueue**

```
package Collections.interfaces;

import java.util.PriorityQueue;
import java.util.Queue;

public class QueueDemo1 {
    public static void main(String[] args) {
        Queue<Integer> numbers = new PriorityQueue<>();

        // offer elements to the Queue
        numbers.offer(5);
        numbers.offer(1);
        numbers.offer(2);
        System.out.println("Queue: " + numbers);
        // Access elements of the Queue
        int accessedNumber = numbers.peek();
        System.out.println("Accessed Element: " +
accessedNumber);

        // Remove elements from the Queue
        int removedNumber = numbers.poll();
        System.out.println("Removed Element: " +
removedNumber);

        System.out.println("Updated Queue: " + numbers);
    }
}
```

Output:
```
Queue: [1, 5, 2]
Accessed Element: 1
Removed Element: 1
Updated Queue: [2, 5]
```

PriorityQueue: The PriorityQueue class provides the functionality of the heap data structure. It implements the Queue interface.

Unlike normal queues, priority queue elements are retrieved in sorted order. Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

Methods of PriorityQueue:

1. **add()** - Inserts the specified element to the queue. If the queue is full, it throws an exception.
2. **offer()** - Inserts the specified element to the queue. If the queue is full, it returns false.
3. **contains(element)**: Searches the priority queue for the specified element. If the element is found, it returns true, if not it returns false.
4. **size():** Returns the length of the priority queue.
5. **toArray():** Converts a priority queue to an array and returns it.

1. **Adding and accessing elements in PriorityQueue.**

```
package Collections.priorityqueue;

import java.util.PriorityQueue;

public class PriorityQueue1 {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new
PriorityQueue<>();
```

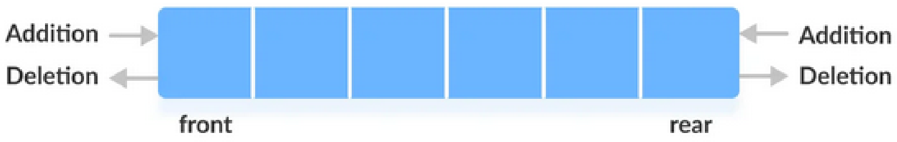
```
        // Using the add() method
        numbers.add(4);
        numbers.add(2);
        System.out.println("PriorityQueue: " + numbers);

        // Using the offer() method
        numbers.offer(1);
        System.out.println("Updated PriorityQueue: " +
numbers);

        // Using the peek() method
        int number = numbers.peek();
        System.out.println("Accessed Element: " + number);
    }
}
```

Output: PriorityQueue: [2, 4]
Updated PriorityQueue: [1, 4, 2]
Accessed Element: 1

2. **Removing and iterating elements over a PriorityQueue.**

```
package Collections.priorityqueue;

import java.util.Iterator;
import java.util.PriorityQueue;

public class PriorityQueue2 {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new
PriorityQueue<>();
        numbers.add(4);
        numbers.add(2);
        numbers.add(1);
        System.out.println("PriorityQueue: " + numbers);

        // Using the remove() method
        boolean result = numbers.remove(2);
        System.out.println("Is the element 2 removed? " +
result);

        // Using the poll() method
        int number = numbers.poll();
        System.out.println("Removed Element Using poll():
" +
number);

        //Using the iterator() method
        Iterator<Integer> iterate = numbers.iterator();
```

```
            while(iterate.hasNext()) {
                System.out.print(iterate.next());
                System.out.print(", ");
            }
        }
    }
```
Output:
```
PriorityQueue: [1, 4, 2]
Is the element 2 removed? true
Removed Element Using poll(): 1
4,
```
Deque interface: The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface. In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.



In order to use the functionalities of the Deque interface, we need to use classes that implement it: ArrayDeque and LinkedList.

Methods of Deque:

1. **addFirst()** - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.
2. **addLast()** - Adds the specified element at the end of the deque. Throws an exception if the deque is full.
3. **offerFirst()** - Adds the specified element at the beginning of the deque. Returns false if the deque is full.
4. **offerLast()** - Adds the specified element at the end of the deque. Returns false if the deque is full.
5. **getFirst()** - Returns the first element of the deque. Throws an exception if the deque is empty.
6. **getLast()** - Returns the last element of the deque. Throws an exception if the deque is empty.
7. **peekFirst()** - Returns the first element of the deque. Returns null if the deque is empty.
8. **peekLast()** - Returns the last element of the deque. Returns null if the deque is empty
9. **removeFirst()** - Returns and removes the first element of the deque. Throws an exception if the deque is empty.

10. **removeLast()** - Returns and removes the last element of the deque. Throws an exception if the deque is empty.
11. **pollFirst()** - Returns and removes the first element of the deque. Returns null if the deque is empty.
12. **pollLast()** - Returns and removes the last element of the deque. Returns null if the deque is empty.

1. **Implementation of Deque in ArrayDeque.**

```java
package Collections.interfaces;

import java.util.ArrayDeque;
import java.util.Deque;

public class DequeDemo {
    public static void main(String[] args) {
        Deque<Integer> numbers = new ArrayDeque<>();

        // add elements to the Deque
        numbers.offer(1);
        numbers.offerLast(2);
        numbers.offerFirst(3);
        System.out.println("Deque: " + numbers);

        // Access elements
        int firstElement = numbers.peekFirst();
        System.out.println("First Element: " +
firstElement);

        int lastElement = numbers.peekLast();
        System.out.println("Last Element: " + lastElement);

        // Remove elements
        int removedNumber1 = numbers.pollFirst();
        System.out.println("Removed First Element: " +
removedNumber1);

        int removedNumber2 = numbers.pollLast();
        System.out.println("Removed Last Element: " +
removedNumber2);

        System.out.println("Updated Deque: " + numbers);
    }
}
```

Output:
```
Deque: [3, 1, 2]
First Element: 3
Last Element: 2
Removed First Element: 3
Removed Last Element: 2
Updated Deque: [1]
```

ArrayDeque: In Java, we can use the ArrayDeque class to implement queue and deque data structures using arrays.

important features of ArrayDeque:

- → Array deques have no capacity restrictions and they grow as necessary to support usage.
- → They are not thread-safe which means that in the absence of external synchronization, ArrayDeque does not support concurrent access by multiple threads.
- → Null elements are prohibited in the ArrayDeque.
- → ArrayDeque class is likely to be faster than Stack when used as a stack.
- → ArrayDeque class is likely to be faster than LinkedList when used as a queue.

Methods of ArrayDeque:

1. **add()** - inserts the specified element at the end of the array deque
2. **addFirst()** - inserts the specified element at the beginning of the array deque
3. **addLast()** - inserts the specified at the end of the array deque (equivalent to add())
4. **offer()** - inserts the specified element at the end of the array deque
5. **offerFirst()** - inserts the specified element at the beginning of the array deque
6. **offerLast()** - inserts the specified element at the end of the array deque
7. **getFirst()** - returns the first element of the array deque.
8. **getLast()** - returns the last element of the array deque.
9. **peek()** - returns the first element of the array deque
10. **peekFirst()** - returns the first element of the array deque (equivalent to peek())
11. **peekLast()** - returns the last element of the array deque.
12. **poll()** - returns and removes the first element of the array deque
13. **pollFirst()** - returns and removes the first element of the array deque (equivalent to poll())
14. **pollLast()** - returns and removes the last element of the array deque.

1. **Inserting elements into ArrayDeque by using various methods.**

```
package Collections.arraydeque;

import java.util.ArrayDeque;

public class ArrayDeque1 {
    public static void main(String[] args) {
        ArrayDeque<String> animals = new ArrayDeque<>();

        // Using add()
        animals.add("Dog");

        // Using addFirst()
        animals.addFirst("Cat");

        // Using addLast()
        animals.addLast("Horse");

        // Using offer()
        animals.offer("Lion");

        // Using offerFirst()
        animals.offerFirst("Tiger");

        // Using offerLast()
        animals.offerLast("Crocodile");
        System.out.println("ArrayDeque: " + animals);

    }
}
```

Output: ArrayDeque: [Tiger, Cat, Dog, Horse, Lion, Crocodile]

2. **Accessing elements in ArrayDeque by using various methods.**

```
package Collections.arraydeque;

import java.util.ArrayDeque;

public class ArrayDeque2 {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Get the first element
        String firstElement = animals.getFirst();
        System.out.println("First Element: " +
firstElement);

        // Get the last element
        String lastElement = animals.getLast();
        System.out.println("Last Element: " +
lastElement);
```

```
        // Using peek()
        String element = animals.peek();
        System.out.println("Head Element: " + element);

        // Using peekFirst()
        String firstElement1 = animals.peekFirst();
        System.out.println("First Element: " +
firstElement1);

        // Using peekLast
        String lastElement2 = animals.peekLast();
        System.out.println("Last Element: " +
lastElement2);
    }
}
```

Output: ArrayDeque: [Dog, Cat, Horse]

```
First Element: Dog
Last Element: Horse
Head Element: Dog
First Element: Dog
Last Element: Horse
```

3. **Removing elements in ArrayDeque by using various methods.**

```
package Collections.arraydeque;

import java.util.ArrayDeque;

public class ArrayDeque3 {
    public static void main(String[] args) {
        ArrayDeque<String> animals= new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Cow");
        animals.add("Horse");
        System.out.println("ArrayDeque: " + animals);

        // Using remove()
        String element = animals.remove();
        System.out.println("Removed Element: " + element);

        System.out.println("New ArrayDeque: " + animals);

        // Using removeFirst()
        String firstElement = animals.removeFirst();
        System.out.println("Removed First Element: " +
firstElement);
```

```
        // Using removeLast()
        String lastElement = animals.removeLast();
        System.out.println("Removed Last Element: " +
lastElement);

        // Using poll()
        String element1 = animals.poll();
        System.out.println("Removed Element: " +
element1);
        System.out.println("New ArrayDeque: " + animals);

        // Using pollFirst()
        String firstElement1 = animals.pollFirst();
        System.out.println("Removed First Element: " +
firstElement1);

        // Using pollLast()
        String lastElement2 = animals.pollLast();
        System.out.println("Removed Last Element: " +
lastElement2);
    }
}
```

Output:
```
ArrayDeque: [Dog, Cat, Cow, Horse]
Removed Element: Dog
New ArrayDeque: [Cat, Cow, Horse]
Removed First Element: Cat
Removed Last Element: Horse
Removed Element: Cow
New ArrayDeque: []
Removed First Element: null
Removed Last Element: null
```

4. Iterating the ArrayDeque.

```
package Collections.arraydeque;

import java.util.ArrayDeque;
import java.util.Iterator;

public class ArrayDeque4 {
    public static void main(String[] args) {
        ArrayDeque<String> animals = new ArrayDeque<>();
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");

        System.out.print("ArrayDeque: ");
        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        while (iterate.hasNext()) {
            System.out.print(iterate.next());
            System.out.print(", ");
        }

        System.out.print("\nArrayDeque in reverse order: 
");
        // Using descendingIterator()
        Iterator<String> desIterate = 
animals.descendingIterator();
        while (desIterate.hasNext()) {
            System.out.print(desIterate.next());
            System.out.print(", ");
        }
    }
}
```

Output: ArrayDeque: Dog, Cat, Horse,
ArrayDeque in reverse order: Horse, Cat, Dog,

BlockingQueue: The BlockingQueue interface of the Java Collections framework extends the Queue interface. It allows any operation to wait until it can be successfully performed.

For example, if we want to delete an element from an empty queue, then the blocking queue allows the delete operation to wait until the queue contains some elements to be deleted.

Implementation of BlockingQueue in ArrayBlockingQueue.

```
package Collections.blockingqueue;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueue1 {
    public static void main(String[] args) {
        BlockingQueue<Integer> numbers = new 
ArrayBlockingQueue<>(5);

        try {
            // Insert element to blocking queue
            numbers.put(2);
            numbers.put(1);
            numbers.put(3);
            System.out.println("BLockingQueue: " + 
numbers);
```

```
            // Remove Elements from blocking queue
            int removedNumber = numbers.take();
            System.out.println("Removed Number: " +
removedNumber);
        } catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: BLockingQueue: [2, 1, 3]

Removed Number: 2

In Java, BlockingQueue is considered as the thread-safe collection. It is because it can be helpful in multi-threading operations.

Suppose one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread runs slower, then the blocking queue can make the second thread wait until the first thread completes its operation.

ArrayBlockingQueue: The ArrayBlockingQueue class of the Java Collections framework provides the blocking queue implementation using an array. It implements the Java BlockingQueue interface.

Using some of the methods in ArrayBlockingQueue.

```
package Collections.arrayblockingqueue;

import java.util.Iterator;
import java.util.concurrent.ArrayBlockingQueue;

public class ArrayBlockingQueue1 {
    public static void main(String[] args) {
        ArrayBlockingQueue<String> animals = new
ArrayBlockingQueue<>(10);

        // Add elements
        animals.add("Dog");
        animals.add("Cat");
        animals.add("Horse");
        System.out.println("ArrayBlockingQueue: " +
animals);

        // Using peek()
        String element = animals.peek();
        System.out.println("Accessed Element: " + element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("ArrayBlockingQueue Elements: ");
        while              (iterate.hasNext())              {
            System.out.print(iterate.next());
            System.out.print(",                              ");
        }

        //                  Using                     remove()
        String      element1      =      animals.remove();
        System.out.println("Removed              Element:");
        System.out.println("Using remove(): " + element1);

        //                   Using                      poll()
        String       element2       =       animals.poll();
        System.out.println("Using  poll():  "  +  element2);

        //                   Using                     clear()
        animals.clear();
        System.out.println("Updated ArrayBlockingQueue: " +
animals);
    }
}
```

Output: ArrayBlockingQueue: [Dog, Cat, Horse]
Accessed Element: Dog
ArrayBlockingQueue Elements: Dog, Cat, Horse, Removed Element:
Using remove(): Dog
Using poll(): Cat
Updated ArrayBlockingQueue: []

The ArrayBlockingQueue uses arrays as its internal storage.
It is considered as a thread-safe collection. Hence, it is generally used in multi-threading applications.
Suppose, one thread is inserting elements to the queue and another thread is removing elements from the queue.
Now, if the first thread is slower than the second thread, then the array blocking queue can make the second thread waits until the first thread completes its operations.

LinkedBlockingQueue: The LinkedBlockingQueue class of the Java Collections framework provides the blocking queue implementation using a linked list. It implements the Java BlockingQueue interface.

Using some of the methods in LinkedBlockingQueue.

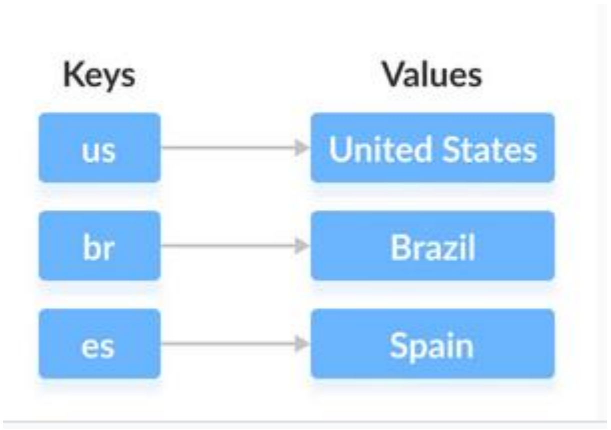
```
package Collections.linkedblockingqueue;

import java.util.Iterator;
import java.util.concurrent.LinkedBlockingQueue;

public class LinkedBlockingQueue1 {
    public static void main(String[] args) {
        LinkedBlockingQueue<String> animals = new
LinkedBlockingQueue<>(5);

        // Using add()
        animals.add("Dog");
        animals.add("Cat");

        // Using offer()
        animals.offer("Horse");
        System.out.println("LinkedBlockingQueue: " +
animals);

        // Using peek()
        String element = animals.peek();
        System.out.println("Accessed Element: " +
element);

        // Using iterator()
        Iterator<String> iterate = animals.iterator();
        System.out.print("LinkedBlockingQueue Elements:
");

        // Using remove()
        String element1 = animals.remove();
        System.out.println("Removed Element:");
        System.out.println("Using remove(): " + element1);

        // Using poll()
        String element2 = animals.poll();
        System.out.println("Using poll(): " + element2);

        // Using clear()
        animals.clear();
        System.out.println("Updated LinkedBlockingQueue "
+
animals);
    }
}
```

Output:
```
LinkedBlockingQueue: [Dog, Cat, Horse]
Accessed Element: Dog
LinkedBlockingQueue Elements: Removed Element:
Using remove(): Dog
Using poll(): Cat
Updated LinkedBlockingQueue []
```

The LinkedBlockingQueue uses linked lists as its internal storage.

It is considered as a thread-safe collection. Hence, it is generally used in multi-threading applications.
Suppose, one thread is inserting elements to the queue and another thread is removing elements from the queue.
Now, if the first thread is slower than the second thread, then the linked blocking queue can make the second thread waits until the first thread completes its operations.

Map Interface: In Java, elements of `Map` are stored in key/value pairs. Keys are unique values associated with individual Values.
A map cannot contain duplicate keys. And, each key is associated with a single value.



We can access and modify values using the keys associated with them. In the above diagram, we have values: United States, Brazil, and Spain. And we have corresponding keys: us, br, and es. Now, we can access those values using their corresponding keys.
Classes that implement Map are HashMap, EnumMap, LinkedHashMap and TreeMap. Interfaces that extend Map are SortedMap, NavigableMap and ConcurrentMap.

Methods of Map:

1. **put(K, V)** - Inserts the association of a key <u>K</u> and a value <u>V</u> into the map. If the key is already present, the new value replaces the old value.
2. **putAll()** - Inserts all the entries from the specified map to this map.
3. **putIfAbsent(K, V)** - Inserts the association if the key <u>K</u> is not already associated with the value <u>V</u>.
4. **get(K)** - Returns the value associated with the specified key <u>K</u>. If the key is not found, it returns null.
5. **getOrDefault(K, defaultValue)** - Returns the value associated with the specified key <u>K</u>. If the key is not found, it returns the <u>defaultValue</u>.
6. **containsKey(K)** - Checks if the specified key <u>K</u> is present in the map or not.

7. **containsValue(V)** - Checks if the specified value V is present in the map or not.
8. **replace(K, V)** - Replace the value of the key K with the new specified value V.
9. **replace(K, oldValue, newValue)** - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
10. **remove(K)** - Removes the entry from the map represented by the key K.

11. **remove(K, V)** - Removes the entry from the map that has key K associated with value V.
12. **keySet()** - Returns a set of all the keys present in a map.
13. **values()** - Returns a set of all the values present in a map.
14. **entrySet()** - Returns a set of all the key/value mapping present in a map.

1. **Implemetation of the Map interface by using haspMap class**

```
package Collections.maps;

import java.util.HashMap;
import java.util.Map;

public class Map1 {
    public static void main(String[] args) {
        Map<String, Integer> numbers = new HashMap<>();

        // Insert elements to the map
        numbers.put("One", 1);
        numbers.put("Two", 2);
        System.out.println("Map: " + numbers);

        // Access keys of the map
        System.out.println("Keys: " + numbers.keySet());

        // Access values of the map
        System.out.println("Values: " + numbers.values());

        // Access entries of the map
        System.out.println("Entries: " +
numbers.entrySet());

        // Remove Elements from the map
        int value = numbers.remove("Two");
        System.out.println("Removed Value: " + value);
    }
}
```

Output: Map: {One=1, Two=2}

```
Keys: [One, Two]
Values: [1, 2]
Entries: [One=1, Two=2]
Removed Value: 2
```

2. **Implementation of the Map interface by using TreeMap class.**

```
package Collections.maps;

import java.util.Map;
import java.util.TreeMap;

public class Map2 {
    public static void main(String[] args) {
        // Creating Map using TreeMap
        Map<String, Integer> values = new TreeMap<>();

        // Insert elements to map
        values.put("Second", 2);
        values.put("First", 1);
      System.out.println("Map using TreeMap: " + values);

        // Replacing the values
        values.replace("First", 11);
        values.replace("Second", 22);
        System.out.println("New Map: " + values);

        // Remove elements from the map
        int removedValue = values.remove("First");
        System.out.println("Removed Value: " +
removedValue);
    }
}
```

Output:
```
Map using TreeMap: {First=1, Second=2}
New Map: {First=11, Second=22}
Removed Value: 11
```

HashMap: The HashMap class of the Java collections framework provides the functionality of the hash table data structure. It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map. The HashMap class implements the Map interface.

1. **Adding, accessing and removing elements in hashMap.**

```
package Collections.hashmap;

import java.util.HashMap;

public class Hashmap1 {
    public static void main(String[] args) {
        HashMap<Integer, String > numbers = new HashMap<>();

        System.out.println("Initial HashMap: " + numbers);
        // put() method to add elements
        numbers.put(1, "One");
        numbers.put(2, "Two");
        numbers.put(3, "Three");
        System.out.println("HashMap after put(): " + numbers);

        // get() method to get value
        String value = numbers.get(2);
        System.out.println("Value at index 1: " + value);
        // using keySet()
        System.out.println("Keys: " + numbers.keySet());

        // using values()
        System.out.println("Values: " + numbers.values());

        // using entrySet()
        System.out.println("Key/Value mappings: " + numbers.entrySet());

        // remove element associated with key 2
        String value1 = numbers.remove(2);
        System.out.println("Removed value: " + value1);

        System.out.println("Updated HashMap: " + numbers);
    }
}
```

Output:
```
Initial HashMap: {}
HashMap after put(): {1=One, 2=Two, 3=Three}
Value at index 1: Two
Keys: [1, 2, 3]
Values: [One, Two, Three]
Key/Value mappings: [1=One, 2=Two, 3=Three]
Removed value: Two
Updated HashMap: {1=One, 3=Three}
```

2. **Iterating the HashMap.**

```
package Collections.hashmap;

import java.util.HashMap;
import java.util.Map;

public class HashMap2 {
    public static void main(String[] args) {
        HashMap<Integer, String> cities = new HashMap<>();
        cities.put(1,                              "Hyderabad");
        cities.put(2,                              "Bangalore");
        cities.put(3,                                 "Kerala");
        System.out.println("HashMap:    "    +    cities);

        //      iterate       through       keys       only
        System.out.print("Keys:                            ");
        for    (Integer    key    :    cities.keySet())    {
            System.out.print(key);
            System.out.print(",                            ");
        }

        //      iterate      through      values      only
        System.out.print("\nValues:                          ");
        for    (String    value    :    cities.values())    {
            System.out.print(value);
            System.out.print(",                            ");
        }

        //     iterate     through     key/value     entries
        System.out.print("\nEntries:                         ");
        for    (Map.Entry<Integer,    String>    entry    :
cities.entrySet())                                         {
            System.out.print(entry);
            System.out.print(",                            ");
        }
    }
}
```

Output: HashMap: {1=Hyderabad, 2=Bangalore, 3=Kerala}
Keys: 1, 2, 3,
Values: Hyderabad, Bangalore, Kerala,
Entries: 1=Hyderabad, 2=Bangalore, 3=Kerala,

LinkedHashMap: The LinkedHashMap class of the Java collections framework provides the hash table and linked list implementation of the Map interface. The LinkedHashMap interface extends the HashMap class to store its entries in a hash table. It internally maintains a doubly-linked list among all of its entries to order its entries.

here is how we can create linked hashmaps in Java.

```
LinkedHashMap<Key, Value> numbers = new LinkedHashMap<>(8, 0.6f);
```

In the above code, we have created a linked hashmap named numbers.

Here,

- Key - a unique identifier used to associate each element (value) in a map
- Value - elements associated by the keys in a map.

Here, the first parameter is **capacity** and the second parameter is **loadFactor**.

- **capacity** - The capacity of this linked hashmap is 8. Meaning, it can store 8 entries.
- **loadFactor** - The load factor of this linked hashmap is 0.6. This means, whenever our hash map is filled by 60%, the entries are moved to a new hash table of double the size of the original hash table.

1. **Adding, accessing and removing elements from the LinkedHAshMap.**

```
package Collections.linkedhashmap;

import java.util.LinkedHashMap;

public class LinkedHashMap1 {
    public static void main(String[] args) {
        LinkedHashMap<String, Integer> evenNumbers = new
LinkedHashMap<>();

        // Using put()
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);
        System.out.println("Original LinkedHashMap: " +
evenNumbers);

        // Using putIfAbsent()
        evenNumbers.putIfAbsent("Six", 6);
        System.out.println("Updated LinkedHashMap(): " +
evenNumbers);

        //Creating LinkedHashMap of numbers
        LinkedHashMap<String, Integer> numbers =
new
LinkedHashMap<>();
        numbers.put("One", 1);

        // Using putAll()
        numbers.putAll(evenNumbers);
        System.out.println("New LinkedHashMap: " +
numbers);

        // Using entrySet()
        System.out.println("Key/Value mappings: " +
numbers.entrySet());

        // Using keySet()
        System.out.println("Keys: " + numbers.keySet());

        // Using values()
        System.out.println("Values: " + numbers.values());
        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Returned Number: " + value2);

        // remove method with single parameter
        int value = numbers.remove("Two");
        System.out.println("Removed value: " + value);

        // remove method with two parameters
        boolean result = numbers.remove("Three", 3);
        System.out.println("Is the entry Three removed? "
+ result);

        System.out.println("Updated LinkedHashMap: " +
numbers);
    }
}
```

Output:
```
Original LinkedHashMap: {Two=2, Four=4}
Updated LinkedHashMap(): {Two=2, Four=4, Six=6}
New LinkedHashMap: {One=1, Two=2, Four=4, Six=6}
Key/Value mappings: [One=1, Two=2, Four=4, Six=6]
Keys: [One, Two, Four, Six]
Values: [1, 2, 4, 6]
Returned Number: 5
Removed value: 2
Is the entry Three removed? false
Updated LinkedHashMap: {One=1, Four=4, Six=6}
```

Enum HashMap: The EnumMap class of the Java collections framework provides a map implementation for elements of an enum. In EnumMap, enum elements are used as **keys**. It implements the Map interface.

1. **Insertion, accessing, removing and replacing of Enum HashMap elements.**

```
package                                 Collections.enumhashmap;

import                                        java.util.EnumMap;

enum                            Size                          {
    SMALL,          MEDIUM,          LARGE,          EXTRALARGE
}

public            class            EnumHashMap1               {
    public    static    void    main(String[]    args)    {
        EnumMap<Size,   Integer>  sizes1   =   new   EnumMap<>
(Size.class);
```

```
        // Using the put() Method
        sizes1.put(Size.SMALL, 28);
        sizes1.put(Size.MEDIUM, 32);
        System.out.println("EnumMap1: " + sizes1);

        EnumMap<Size, Integer> sizes2 = new EnumMap<>
(Size.class);

        // Using the putAll() Method
        sizes2.putAll(sizes1);
        sizes2.put(Size.LARGE, 36);
        System.out.println("EnumMap2: " + sizes2);

        // Using the entrySet() Method
        System.out.println("Key/Value mappings: " +
sizes1.entrySet());

        // Using the keySet() Method
        System.out.println("Keys: " + sizes1.keySet());

        // Using the values() Method
        System.out.println("Values: " + sizes1.values());

        // Using the get() Method
        int value = sizes1.get(Size.MEDIUM);
        System.out.println("Value of MEDIUM: " + value);

        // Using the remove() Method
        int value1 = sizes1.remove(Size.MEDIUM);
        System.out.println("Removed Value: " + value1);

        boolean result = sizes1.remove(Size.SMALL, 28);
        System.out.println("Is the entry {SMALL=28}
removed? " + result);

sizes1.replace(Size.MEDIUM, 30);
sizes1.replace(Size.LARGE, 36, 34);
System.out.println("EnumMap using replace(): " + sizes1);

// Using the replaceAll() Method
sizes1.replaceAll((key, oldValue) -> oldValue + 3);
System.out.println("EnumMap using replaceAll(): " +
sizes1);

        System.out.println("Updated EnumMap: " + sizes1);
```

```
    }
}
```

Output: EnumMap1: {SMALL=28, MEDIUM=32}
EnumMap2: {SMALL=28, MEDIUM=32, LARGE=36}
Key/Value mappings: [SMALL=28, MEDIUM=32]
Keys: [SMALL, MEDIUM]
Values: [28, 32]
Value of MEDIUM: 32
Removed Value: 32
Is the entry {SMALL=28} removed? True
EnumMap using replace(): {}
EnumMap using replaceAll(): {}
Updated EnumMap: {}

SortedMap Interface: The SortedMap interface of the Java collections framework provides sorting of keys stored in a map. It extends the Map interface.

Methods of SortedMap:

1. **comparator()** - returns a comparator that can be used to order keys in a map

2. **firstKey()** - returns the first key of the sorted map
3. **lastKey()** - returns the last key of the sorted map
4. **headMap(key)** - returns all the entries of a map whose keys are less than the specified key
5. **tailMap(key)** - returns all the entries of a map whose keys are greater than or equal to the specified key
6. **subMap(key1, key2)** - returns all the entries of a map whose keys lies in between key1 and key2 including key1.

Implementation of the SortedMap in TreeMap Class.

```
package Collections.sortedmap;

import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMap1 {
    public static void main(String[] args) {
    SortedMap<String, Integer> numbers = new TreeMap<>();

        // Insert elements to map
        numbers.put("Two", 2);
        numbers.put("One", 1);
        System.out.println("SortedMap: " + numbers);


        // Access the first key of the map
        System.out.println("First      Key:      "      +
numbers.firstKey());

        //   Access   the   last   key   of   the   map
        System.out.println("Last      Key:      "      +
numbers.lastKey());

        //    Remove    elements    from    the    map
        int       value       =       numbers.remove("One");
        System.out.println("Removed  Value:  "  +  value);
    }
}
```

Output: SortedMap: {One=1, Two=2}

```
First Key: One
Last Key: Two
Removed Value: 1
```

NavigableMap Interface: The NavigableMap interface of the Java collections framework provides the features to navigate among the map entries. It is considered as a type of SortedMap.

Methods of NavigableMap:

The NavigableMap is considered as a type of SortedMap. It is because NavigableMap extends the SortedMap interface. Hence, all SortedMap methods are also available in NavigableMap. To learn how these methods are defined in SortedMap, visit Java SortedMap.

1. **headMap(key, booleanValue):** The headMap() method returns all the entries of a navigable map associated with all those keys before the specified key (which is passed as an argument). The booleanValue is an optional parameter. Its default value is false. If true is passed as a booleanValue, the method returns all the entries associated with all those keys before the specified key, including the entry associated with the specified key.
2. **tailMap(key, booleanValue):** The tailMap() method returns all the entries of a navigable map associated with all those keys after the specified key (which is passed as an argument) including the entry associated with the specified key.The booleanValue is an optional parameter. Its default value is true.If false is passed as a booleanValue, the method returns all the entries associated with those keys after the specified key, without including the entry associated with the specified key.
3. **subMap(k1, bv1, k2, bv2):** The subMap() method returns all the entries associated with keys between k1 and k2 including the entry associated with k1. The bv1 and bv2 are optional parameters. The default value of bv1 is true and the default value of bv2 is false. If false is passed as bv1, the method returns all the entries associated with keys between k1 and k2, without including the entry associated with k1. If true is passed as bv2, the method

returns all the entries associated with keys between k1 and k2, including the entry associated with k1.

4.

Implemetation of NavigableMap in TreeMap class.

```
package Collections.navigablemap;

import java.util.NavigableMap;
import java.util.TreeMap;

public class NavigableMap1 {
    public static void main(String[] args) {
        NavigableMap<String, Integer> numbers = new TreeMap<>();

        // Insert elements to map
        numbers.put("Two", 2);
        numbers.put("One", 1);
        numbers.put("Three", 3);
        System.out.println("NavigableMap: " + numbers);

        // Access the first entry of the map
        System.out.println("First Entry: " + numbers.firstEntry());

        // Access the last entry of the map
        System.out.println("Last Entry: " + numbers.lastEntry());

        // Remove the first entry from the map
        System.out.println("Removed First Entry: " + numbers.pollFirstEntry());

        // Remove the last entry from the map
        System.out.println("Removed Last Entry: " + numbers.pollLastEntry());
    }
}
```

Output:
```
NavigableMap: {One=1, Three=3, Two=2}
First Entry: One=1
Last Entry: Two=2
Removed First Entry: One=1
Removed Last Entry: Two=2
```

TreeMap: The TreeMap class of the Java collections framework provides the tree data structure implementation. It implements the NavigableMap interface.

Methods in TreeMap:

1. **put()** - inserts the specified key/value mapping (entry) to the map
2. **putAll()** - inserts all the entries from specified map to this map
3. **putIfAbsent()** - inserts the specified key/value mapping to the map if the specified key is not present in the map.
4. **entrySet()** - returns a set of all the key/values mapping (entry) of a treemap

5. **keySet()** - returns a set of all the keys of a tree ma
6. **values()** - returns a set of all the maps of a tree map.
7. **get()** - Returns the value associated with the specified key. Returns null if the key is not found.
8. **getOrDefault()** - Returns the value associated with the specified key. Returns the specified default value if the key is not found.
9. **remove(key)** - returns and removes the entry associated with the specified key from a TreeMap
10. **remove(key, value)** - removes the entry from the map only if the specified key is associated with the specified value and returns a boolean value.

1. **Insertion, accessing, removing and replacing of TreeMap elements.**

```
package Collections.treemap;

import java.util.TreeMap;

public class TreeMap1 {
    public static void main(String[] args) {
        TreeMap<String, Integer> evenNumbers = new
TreeMap<>();

        // Using put()
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);

        // Using putIfAbsent()
        evenNumbers.putIfAbsent("Six", 6);
        System.out.println("TreeMap of even numbers: " +
evenNumbers);

        //Creating TreeMap of numbers
        TreeMap<String, Integer> numbers = new TreeMap<>();
        numbers.put("One", 1);

        // Using putAll()
        numbers.putAll(evenNumbers);
        System.out.println("TreeMap of numbers: " +
numbers);

        // Using entrySet()
        System.out.println("Key/Value mappings: " + numbers.entrySet());

        // Using keySet()
        System.out.println("Keys: " + numbers.keySet());

        // Using values()
        System.out.println("Values: " + numbers.values());


       // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Using getOrDefault(): " + value2);

        // remove method with single parameter
        int value = numbers.remove("Two");
        System.out.println("Removed value: " + value);

        // remove method with two parameters
        boolean result = numbers.remove("Three", 3);
        System.out.println("Is the entry {Three=3} removed? " + result);

        System.out.println("Updated TreeMap: " + numbers);

        // Using replace()
        numbers.replace("Second", 22);
        numbers.replace("Third", 3, 33);
        System.out.println("TreeMap using replace: " + numbers);

        // Using replaceAll()
        numbers.replaceAll((key, oldValue) -> oldValue + 2);
        System.out.println("TreeMap using replaceAll: " + numbers);
    }
}
```

Output:
```
TreeMap of even numbers: {Four=4, Six=6, Two=2}
TreeMap of numbers: {Four=4, One=1, Six=6, Two=2}
Key/Value mappings: [Four=4, One=1, Six=6, Two=2]
Keys: [Four, One, Six, Two]
Values: [4, 1, 6, 2]
Using getOrDefault(): 5
Removed value: 2
```

```
Is the entry {Three=3} removed? false
Updated TreeMap: {Four=4, One=1, Six=6}
TreeMap using replace: {Four=4, One=1, Six=6}
TreeMap using replaceAll: {Four=6, One=3, Six=8}.
```

2. **Using navigation, floor, ceiling, higher and lower methods in TreeMap.**

```
package Collections.treemap;

import java.util.TreeMap;

public class TreeMap2 {
    public static void main(String[] args) {
        TreeMap<String, Integer> numbers = new
TreeMap<>();
        numbers.put("First", 1);
        numbers.put("Second", 2);
        numbers.put("Third", 3);
        System.out.println("TreeMap: " + numbers);

        // Using the firstKey() method
        String firstKey = numbers.firstKey();
        System.out.println("First Key: " + firstKey);

        // Using the lastKey() method
        String lastKey = numbers.lastKey();
        System.out.println("Last Key: " + lastKey);

        // Using firstEntry() method
        System.out.println("First Entry: "
+
numbers.firstEntry());


        // Using the lastEntry() method
        System.out.println("Last Entry: " +
numbers.lastEntry());

        // Using higher()
        System.out.println("Using higherKey(): "
+
numbers.higherKey("Fourth"));
        System.out.println("Using higherEntry(): " +
numbers.higherEntry("Fourth"));

        // Using lower()
        System.out.println("\nUsing lowerKey(): " +
numbers.lowerKey("Fourth"));
        System.out.println("Using lowerEntry(): " +
numbers.lowerEntry("Fourth"));

        // Using ceiling()
        System.out.println("\nUsing ceilingKey(): " +
numbers.ceilingKey("Fourth"));
        System.out.println("Using ceilingEntry(): " +
numbers.ceilingEntry("Fourth"));

        // Using floor()
        System.out.println("\nUsing floorKey(): " +
numbers.floorKey("Fourth"));
        System.out.println("Using floorEntry(): " +
numbers.floorEntry("Fourth"));

    }
}
```

Output:
```
TreeMap: {First=1, Second=2, Third=3}
First Key: First
Last Key: Third
First Entry: First=1
Last Entry: Third=3
Using higherKey(): Second
Using higherEntry(): Second=2

Using lowerKey(): First
Using lowerEntry(): First=1

Using ceilingKey(): Second
Using ceilingEntry(): Second=2

Using floorKey(): First
Using floorEntry(): First=1
```

Java ConcurrentMap interface: The ConcurrentMap interface of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map. ConcurrentMap is known as a synchronized map. It extends the Map interface.

Methods of ConcurrentMap:

1. **putIfAbsent()** - Inserts the specified key/value to the map if the specified key is not already associated with any value.
2. **compute()** - Computes an entry (key/value mapping) for the specified key and its previously mapped value.

3. **computeIfAbsent()** - Computes a value using the specified function for the specified key if the key is not already mapped with any value.
4. **computeIfPresent()** - Computes a new entry (key/value mapping) for the specified key if the key is already mapped with the specified value.
5. **forEach()** - Access all entries of a map and perform the specified actions.

6. **merge()** - Merges the new specified value with the old value of the specified key if the key is already mapped to a certain value. If the key is not already mapped, the method simply associates the specified value to our key.

Implementation of ConcurrentMap

```
package Collections.concurrentmap;

import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;

public class ConcurrentMap1 {
    public static void main(String[] args) {
        // Creating ConcurrentMap using ConcurrentHashMap
        ConcurrentMap<String, Integer> numbers = new
ConcurrentHashMap<>();

        // Insert elements to map
        numbers.put("Two", 2);
        numbers.put("One", 1);
        numbers.put("Three", 3);
        System.out.println("ConcurrentMap: " + numbers);

        // Access the value of specified key
        int value = numbers.get("One");
        System.out.println("Accessed Value: " + value);

        // Remove the value of specified key
        int removedValue = numbers.remove("Two");
        System.out.println("Removed Value: " +
removedValue);
    }
}
```

Output:
```
ConcurrentMap: {One=1, Two=2, Three=3}
Accessed Value: 1
Removed Value: 2
```

ConcurrentHashMap: The ConcurrentHashMap class of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map. It implements the ConcurrentMap interface.

Using some methods by in ConcurrentHashMap class

Example 1:

```
package Collections.concurrenthashmap;

import java.util.HashMap;
import java.util.concurrent.ConcurrentHashMap;

public class ConCurrentHashMap1 {
    public static void main(String[] args) {
       HashMap<String, Integer> evenNumbers = new HashMap<>();
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);
        System.out.println("HashMap: " + evenNumbers);

        // Creating a concurrent hashmap from other map
        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>(evenNumbers);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);

        // Using put()
        evenNumbers.put("Two", 2);
        evenNumbers.put("Four", 4);

        // Using putIfAbsent()
        evenNumbers.putIfAbsent("Six", 6);
        System.out.println("ConcurrentHashMap of even numbers:
" +
evenNumbers);

        // Using putAll()
        numbers.putAll(evenNumbers);
        System.out.println("ConcurrentHashMap of numbers: " +
numbers);

        // Using entrySet()
        System.out.println("Key/Value mappings: " +
numbers.entrySet());

        // Using keySet()
        System.out.println("Keys: " + numbers.keySet());

        // Using values()
        System.out.println("Values: " + numbers.values());

        // Using get()
        int value1 = numbers.get("Three");
        System.out.println("Using get(): " + value1);
        // Using getOrDefault()
        int value2 = numbers.getOrDefault("Five", 5);
        System.out.println("Using getOrDefault(): " + value2);

        // remove method with single parameter
        int value = numbers.remove("Two");
        System.out.println("Removed value: " + value);

        // remove method with two parameters
        boolean result = numbers.remove("Three", 3);
        System.out.println("Is the entry {Three=3} removed? " +
result);

        System.out.println("Updated ConcurrentHashMap: " +
numbers);

    }
}
```

Output:
```
HashMap: {Four=4, Two=2}
ConcurrentHashMap: {Four=4, Two=2, Three=3}
ConcurrentHashMap of even numbers: {Six=6, Four=4, Two=2}
ConcurrentHashMap of numbers: {Six=6, Four=4, Two=2, Three=3}
Key/Value mappings: [Six=6, Four=4, Two=2, Three=3]
Keys: [Six, Four, Two, Three]
Values: [6, 4, 2, 3]
Using get(): 3
Using getOrDefault(): 5
Removed value: 2
Is the entry {Three=3} removed? true
Updated ConcurrentHashMap: {Six=6, Four=4}
```

Example2:
```
package Collections.concurrenthashmap;

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHasMap2 {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> numbers = new
ConcurrentHashMap<>();
        numbers.put("One", 1);
        numbers.put("Two", 2);
        numbers.put("Three", 3);
        System.out.println("ConcurrentHashMap: " + numbers);
        // forEach() without transformer function
        numbers.forEach(4, (k, v) -> System.out.println("key: "
+ k + "                                                     value: " +
v));

        // forEach() with transformer function
        System.out.print("Values are ");
        numbers.forEach(4, (k, v) -> v, (v) ->
System.out.print(v
                                                              + ","));

        //                    Using                         search()
        String key = numbers.search(4, (k, v) -> {return v == 3
? k:                                                        null;});
        System.out.println("Searched   value:   "   +   key);

   }
}
```

Output: ConcurrentHashMap: {One=1, Two=2, Three=3}
key: One value: 1
key: Two value: 2
key: Three value: 3
Values are 1, 2, 3, Searched value: Three

Generics: Generics is a technical term denoting a set of language features related to the definition and use of generic types and methods. In java, Generic types or methods differ from regular types and methods in that they have type parameters.

Generic types are instantiated to form parameterized types by providing actual type arguments that replace the formal type parameters. A class like LinkedList<E> is a generic type, that has a type parameter E. Instantiations, such as LinkedList<Integer> or a LinkedList<String>, are called parameterized types, and String and Integer are the respective actual type arguments.

Why Generics ?

If you closely look at java collection framework classes then you will observe that most classes take parameter/argument of type Object and return values from methods as Object. Now, in this form, they can take any java type as argument and return the same. They are essentially heterogeneous i.e. not of a particular similar type.

```
     package                                                 generics;

     import                                        java.util.Iterator;
```

```
import java.util.LinkedList;

class GenericExp {
    public static void main(String args[]) {
        LinkedList<String> list = new LinkedList<>();
        list.add("Name");
        list.add("Name1");
        //list.add(32);//compile time error

        System.out.println("element is: " + list.get(1));

        Iterator<String> itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

Output: element is: Name1

Name

Name1

How Generics works:

Let’s understand with an example.

```
List<Integer> list = new ArrayList<Integer>();

list.add(1000);     //works fine

list.add("lokesh"); //compile time error;
```

When you write above code and compile it, you will get below error:

“The method add(Integer) in the type List<Integer> is not applicable for the arguments (String)“. This exactly is generics sole purpose i.e. Type Safety.

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.

Types of Generics

Generic Type Class or Interface:

A class is generic if it declares one or more type variables. These type variables are known as the type parameters of the class.

1. Generic Type class

```
package generics;

import java.util.ArrayList;

public class MyCustomList<T> {
ArrayList<T> list = new ArrayList<>();

public void addElement(T element) {
list.add(element);
}

public void removeElement(T element) {
list.remove(element);
}

@Override
public String toString() {
return "MyCustomList{" +
"list=" + list +
'}';
}

public T get(int index) {
return list.get(index);
}
}
```

```
package generics;

import generics.MyCustomList;

public class genericsExp1 {
public static void main(String[] args) {
MyCustomList<String> list = new MyCustomList<>();
list.addElement("Element 1");
list.addElement("Element 2");
System.out.println(list);
System.out.println("element at index 0 is " +
list.get(0));

MyCustomList<Integer> list2 = new MyCustomList<>();
list2.addElement(Integer.valueOf(5));
list2.addElement(Integer.valueOf(10));
System.out.println(list2);
System.out.println("number at index 1 is " +
list2.get(1));

}
}
```

Output:

```
MyCustomList{list=[Element 1, Element 2]}

element at index 0 is Element 1

MyCustomList{list=[5, 10]}

number at index 1 is 10
```

2. Generic Type interface

```
package generics;

public interface GenericInterface<T1> {

public void addElement1(T1 element);

public void addElement2(T1 element);

}
```

```
package generics;

import java.util.LinkedList;

public class ImplementsInterface<T1> implements
GenericInterface<String>{
LinkedList<T1> list = new LinkedList<T1>();

@Override
public void addElement1(String element) {
list.add((T1) "element1");
}

@Override
public void addElement2(String element) {
list.add((T1) "element2");

}

@Override
public String toString() {
return "ImplementsInterface{" +
"list=" + list +
'}';
}

public T1 get(int index) {
return list.get(index);
}
}
package generics;

public class GenericExp2 {
public static void main(String[] args) {
ImplementsInterface<String> list = new
ImplementsInterface<>();
list.addElement1("Element 1");
list.addElement2("Element 2");
System.out.println(list);
System.out.println("element at index 0 is " +
list.get(0));

}
}
```

Output:
```
ImplementsInterface{list=[element1, element2]}

element at index 0 is element1
```

Generic Type Method or Constructor: Generic methods are much similar to generic classes. They are different only in one aspect that the scope of type information is only inside the method (or constructor). Generic methods are methods that introduce their own type parameters.

1. Generic method example

```
package generics;

public class Employee {
private Integer id;
private String name;
private Double salary;

public Integer getId() {
return id;
}

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

public String getName() {
return name;
}

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

public Double getSalary() {
return salary;
}

public void setSalary(Double salary) {
this.salary = salary;
}

@Override
public String toString() {
return "Employee{" +
"id=" + id +
", name='" + name + '\'' +
", salary=" + salary +
'}';
}
}

package generics;
```

```
public class GenericExp3 {
public static void main(String[] args) {
Employee employee1 = new Employee();
employee1.setId(101);
employee1.setName("Name1");
employee1.setSalary(500000.00);

Employee employee2 = new Employee();
employee2.setId(102);
employee2.setName("Name2");
employee2.setSalary(400000.00);

Employee empArry[] = new Employee[2];
empArry[0] = employee1;
empArry[1] = employee2;
display(empArry);

Integer[] intArray = { 10, 20, 30, 40, 50 };
display(intArray);
}

//Generic method
private static <E> void display(E[] elements) {
for (E e : elements) {
System.out.println(e);
}
}
}
```

Output:
```
Employee{id=101, name='Name1', salary=500000.0}
Employee{id=102, name='Name2', salary=400000.0}
10
20
30
40
50
```

2. Generic constructor example

```
package generics;
```

```
public class GenericExp4 {
private double val;

//Generic constructor
<T extends Number> GenericExp4(T arg) {
val = arg.doubleValue();
}
public void showVal() {
System.out.println("Value is : "+val);
}

public static void main(String[] args) {
GenericExp4 obj = new GenericExp4(100.00);
GenericExp4 onj1 = new GenericExp4(200.00);
obj.showVal();
onj1.showVal();

}
}
```

Output: Value is : 100.0

Value is : 200.0

Bounded-types: Whenever you want to restrict the type parameter to subtypes of a particular class you can use the bounded type parameter. If you just specify a type (class) as bounded parameter, only sub types of that particular class are accepted by the current generic class. These are known as bounded-types in generics in Java.

You can declare a bound parameter just by extending the required class with the type-parameter, within the angular braces as

```
class Sample <T extends Number>
```

In the following example the generic class restricts the type parameter to the sub classes of the Number classes using the bounded parameter.

```
package generics;

public class BoundGen1<T extends Number> {
T data;

BoundGen1(T data) {
this.data = data;
}

public                T               getData()               {
return                                                    data;
}

public         void         setData(T         data)          {
this.data                     =                           data;
}

public              void              display()              {
System.out.println("Data   value   is   :   "   +   this.data);
}

}
package                                               generics;

public              class             GenericExp5            {
public     static     void     main(String[]     args)      {
BoundGen1<Integer>     obj     =     new     BoundGen1<>(30);
obj.display();
BoundGen1<Double>     obj1     =     new     BoundGen1<>(33.3);
obj1.display();
BoundGen1<Float>     obj2     =     new     BoundGen1<>(33.33f);
obj2.display();
}
}
```

Output:
```
Data value is : 30

Data value is : 33.3

Data value is : 33.33
```

Generics with Wildcards: In generic code, the question mark (?), called the wildcard, represents an unknown type. A wildcard parameterized type is an instantiation of a generic type where at least one type argument is a wildcard. Examples of wildcard parameterized types are **Collection<?<, List<? extends Number<, Comparator<? super String>** and **Pair<String,?>.** The wildcard can be used in a variety of situations: as the type of a parameter, field, or local variable; sometimes as a return type (though it is better programming practice to be more specific). The wildcard is never used as a type argument for a generic method invocation, a generic class instance creation, or a supertype.

Having wild cards at difference places have different meanings as well. e.g.

- → **Collection** denotes all instantiations of the Collection interface regardless of the type argument.
- → **List** denotes all list types where the element type is a subtype of Number.
- → **Comparator<? super String<** denotes all instantiations of the Comparator interface for type argument types that are supertypes of String.

A wildcard parameterized type is not a concrete type that could appear in a new expression. It just hints the rule enforced by java generics that which types are valid in any particular scenario where wild cards have been used.

```
package generics;

public class Wildcard1<T> {
T[] list;

public Wildcard1(T[] list) {
this.list = list;
}
public void greaterThan(Wildcard1<?> otherWildcard) {
if (list.length > otherWildcard.list.length){
System.out.println("True");
}else {
System.out.println("False");
}
}
}
```

```
package generics;

public class GenericExp6 {
public static void main(String[] args) {
var countries = new String[]{"India", "USA", "England",
"New zealand"};
var scores = new Integer[]{110, 189, 214};
var names = new String[]{"Virat", "Buttler", "Stokes"};

var wildcard2 = new Wildcard1<>(countries);
var wildcard3 = new Wildcard1<>(scores);
var wildcard4 = new Wildcard1<>(names);

wildcard2.greaterThan(wildcard3);
wildcard3.greaterThan(wildcard4);
wildcard2.greaterThan(wildcard4);
wildcard4.greaterThan(wildcard2);
wildcard3.greaterThan(wildcard2);
}
}
```

Output: True

False

True

False

False

Bounded wildcard parameterized type: Bounded wildcards put some restrictions over possible types, you can use to instantiate a parametrized type. This restriction is enforced using keywords "super" and "extends". To differentiate more clearly, let's divide them into upper bounded wildcards and lower bounded wildcards.

Upper bounded wildcards: For example, say you want to write a method that works on List<String>, List<Integer>, and List<double> you can achieve this by using an upper bounded wildcard e.g. you would specify List<? extends Number>. Here Integer, Double are subtypes of Number class. In layman's terms, if you want the generic expression to accept all subclasses of a particular type, you will use upper bound wildcard using "**extends**" keyword.

```
package generics;

import java.util.Arrays;
import java.util.List;

public class GenericExp7<T> {
public static void main(String[] args) {
//List of Integers
List<Integer> ints = Arrays.asList(1, 2, 3, 4, 5);
System.out.println("Sum of the numbers " + sum(ints));

//List of Doubles
List<Double> doubles = Arrays.asList(1.5d, 2d, 3d);
System.out.println("Sum of the double digits " +
sum(doubles));
//List<String> strings = Arrays.asList("1","2");
// System.out.println(sum(strings));
//This will give compilation error as :: The method
//sum(List<? extends Number>) in the
//type GenericsExample<T> is not applicable for the
//arguments (List<String>)

}

//Method will accept

// Upper bound Wildcard - ? extends Type.
private static Number sum(List<? extends Number> numbers)
{
double s = 0.0;
for (Number n : numbers)
s += n.doubleValue();
return s;
}
}
```

Output: Sum of the numbers 15.0

Sum of the double digits 6.5

Lower Bounded wildcards: If you want a generic expression to accept all types which are "super" type of a particular type OR parent class of a particular class then you will use a lower bound wildcard for this purpose, using 'super' keyword.

```
package                                                  generics;

import                                            java.util.Arrays;
import                                              java.util.List;

class                        GenericExp8                          {
public     static     void     main(String[]     args)     {
//Lower            Bounded            Integer            List
List<Integer>   list1   =   Arrays.asList(4,   5,   6,   7);

//Integer     list     object     is     being     passed
System.out.println("First      list      :      "+list1);

//Number                                                     list
List<Number>   list2   =   Arrays.asList(4,   5,   6,   7);
//Integer list object is being passed
System.out.println("Second list : "+list2);
}

// Lower bound Wildcard - ? super Type.
public static void SuperClass(List<? super Integer> list) {
System.out.println(list);
}
}
```

Output: First list : [4, 5, 6, 7]

Second list : [4, 5, 6, 7]

Unbounded wildcard parameterized type: A generic type where all type arguments are the unbounded wildcard "?” without any restriction on type variables.

These are useful in the following cases

→ When writing a method which can be employed using functionality provided in Object class.
→ When the code is using methods in the generic class that don’t depend on the type parameter.

```
package generics;

import java.util.Arrays;
import java.util.List;

class GenericExp9 {
public static void main(String[] args) {

//String List
List<String> list1 = Arrays.asList("Name1", "Name2");

//Integer list
List<Integer> list2 = Arrays.asList(11, 22, 33);

printlist(list1);

printlist(list2);
}
// Unbounded Wildcard - ?
private static void printlist(List<?> list) {

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

Output: [Name1, Name2]

[11, 22, 33]

Examples:

1. **Generic class**

```
package generics;

public class Box<T> {
private T t;

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

public T get() {
return t;
}

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

Box<Integer> type1 = new Box<>();
type1.set(100);

System.out.println("String : "+type.get());
System.out.println("Integer "+type1.get());
}
}
```

Output: String : String

Integer 100

2. Generic class with generic methods

```
package generics;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class GenericExp10<X> {

static <X> X doubleValue(X value) {
return value;
}
static <X extends List> void duplicate(X list){
list.addAll(list);
}


public static void main(String[] args) {

ArrayList<Integer> numbers = new ArrayList<>
(List.of(1,2,3));
duplicate(numbers);
System.out.println("numbers : "+numbers);

LinkedList<String> strings = new LinkedList<>
(List.of("String1", "String2"));
duplicate(strings);
System.out.println("Strings : "+strings);

}
}
```

Output: numbers : [1, 2, 3, 1, 2, 3]

Strings : [String1, String2, String1, String2]

3. Using ArrayList with Generics

```
package generics;

class Employee1<T> {
private T name;
private T id;
private T salary;

public Employee1(T name, T id, T salary) {
```

```
this.name = name;
this.id = id;
this.salary = salary;
}

public T getName() {
return name;
}

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

public T getId() {
return id;
}

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

public T getSalary() {
return salary;
}

public void setSalary(T salary) {
this.salary = salary;
}

@Override
public String toString() {
return "Employee1{" +
"name=" + name +
", id=" + id +
", salary=" + salary +
'}';
}
}
package generics;

import java.util.ArrayList;

public class GenericExp12 {
public static void main(String[] args) {
```

```
ArrayList<Employee1<String>> list = new
ArrayList<Employee1<String>>();
Employee1 employee01 = new Employee1("Employee01", 10,
500000);
Employee1 employee02 = new Employee1("Employee02", 11,
550000);
list.add(employee01);
list.add(employee02);
for (Employee1<String> employee1 : list) {
System.out.println(employee1);
}
}
}
```

Output: Employee1{name=Employee01, id=10, salary=500000}

Employee1{name=Employee02, id=11, salary=550000}

4. LinkedList by using Generics

```
package generics;

public class Student<T> {
private T studentName;
private T studentId;

public Student(T studentName, T studentId) {
this.studentName = studentName;
this.studentId = studentId;
}

public T getStudentName() {
return studentName;
}

public void setStudentName(T studentName) {
this.studentName = studentName;
}

public T getStudentId() {
return studentId;
}

public void setStudentId(T studentId) {
this.studentId = studentId;
}

@Override
public String toString() {
return "Name=" + studentName +
", Id=" + studentId;
}
}
package generics;

import java.util.LinkedList;

public class GenericExp13 {
public static void main(String[] args) {
LinkedList<Student<String>> students = new
LinkedList<Student<String>>();
Student student1 = new Student("Student1", 201);
Student student2 = new Student("Student2", 202);

//adding elements
students.add(student1);
students.add(student2);
System.out.println("initial list : "+students);

//removing elements
students.remove(student1);
System.out.println("List after removing :"+students);

//getting size
System.out.println("Size of list "+students.size());
}
}
```

Output: initial list : [Name=Student1, Id=201, Name=Student2, Id=202]

List after removing :[Name=Student2, Id=202]

Size of list 1

5. Vector by using Generics

```
package generics;

public class BankAccount<T> {
private T balance;
private T AccountType;
private T customerName;
private T accountNo;

public BankAccount(T balance, T accountType, T
customerName, T accountNo) {
this.balance = balance;
AccountType = accountType;
this.customerName = customerName;
this.accountNo = accountNo;
}

public T getBalance() {
return balance;
}

public void setBalance(T balance) {
this.balance = balance;
}

public T getAccountType() {
return AccountType;
}

public void setAccountType(T accountType) {
AccountType = accountType;
}

public T getCustomerName() {
return customerName;
}

public void setCustomerName(T customerName) {
this.customerName = customerName;
}

public T getAccountNo() {
return accountNo;
}

public void setAccountNo(T accountNo) {
this.accountNo = accountNo;
}
```

```
@Override
public String toString() {
return "\n{" +
"balance=" + balance +
", AccountType=" + AccountType +
", customerName=" + customerName +
", accountNo=" + accountNo +
'}';
}
}
package generics;

import java.util.Vector;

public class GenericExp14 {
public static void main(String[] args) {
Vector<BankAccount<String>> bankAccounts = new
Vector<BankAccount<String>>();
BankAccount bankAccount1 = new BankAccount("239.00",
"Savings ","Name1",2322);
BankAccount bankAccount2 = new BankAccount("339.00",
"Regular ","Name2",2332);
bankAccounts.add(bankAccount1);
bankAccounts.add(bankAccount2);
System.out.println("Bank Accounts"+bankAccounts);

}
}
```

Output: Bank Accounts[

{balance=239.00, AccountType=Savings , customerName=Name1, accountNo=2322},

{balance=339.00, AccountType=Regular , customerName=Name2, accountNo=2332}]

6. ArrayDeque by using generics.

```
package generics;

public class Hospital<T> {
private T hsptlName;
private T noOfPatients;
private T noOfDoctors;
private T noOfWorkers;

public Hospital(T hsptlName, T noOfPatients, T
noOfDoctors, T noOfWorkers) {
this.hsptlName = hsptlName;
this.noOfPatients = noOfPatients;
this.noOfDoctors = noOfDoctors;
this.noOfWorkers = noOfWorkers;
}

public T getHsptlName() {
return hsptlName;
}

public void setHsptlName(T hsptlName) {
this.hsptlName = hsptlName;
}

public T getNoOfPatients() {
return noOfPatients;
}

public void setNoOfPatients(T noOfPatients) {
this.noOfPatients = noOfPatients;
}

public T getNoOfDoctors() {
return noOfDoctors;
}

public void setNoOfDoctors(T noOfDoctors) {
this.noOfDoctors = noOfDoctors;
}

public T getNoOfWorkers() {
return noOfWorkers;
}

public void setNoOfWorkers(T noOfWorkers) {
this.noOfWorkers = noOfWorkers;
}

@Override
public String toString() {
```

```
return "\n{" +
"hsptlName=" + hsptlName +
", noOfPatients=" + noOfPatients +
", noOfDoctors=" + noOfDoctors +
", noOfWorkers=" + noOfWorkers +
'}';
}
}
```

```
package generics;

import java.util.ArrayDeque;

public class GenericExp15 {
    public static void main(String[] args) {
        ArrayDeque<Hospital<String>> hospitals = new
ArrayDeque<Hospital<String>>();
        Hospital hospital1 = new Hospital("Hospital1", 23,
12, 15);
        Hospital hospital2 = new Hospital("Hospital1", 23,
12, 15);
        hospitals.add(hospital1);
        hospitals.add(hospital2);
        System.out.println("Hospital details "+hospitals);

    }
}
```

Output:
```
Hospital details [
{hsptlName=Hospital1, noOfPatients=23, noOfDoctors=12,
noOfWorkers=15},
{hsptlName=Hospital1, noOfPatients=23, noOfDoctors=12,
noOfWorkers=15}]
```

6. HashSet by using Generics.

```
package generics;

public class School<T> {
private T schoolName;
private T noOfStudents;
private T noOfFaculty;
public School(T schoolName, T noOfStudents, T noOfFaculty)
{
this.schoolName = schoolName;
this.noOfStudents = noOfStudents;
this.noOfFaculty = noOfFaculty;
}

public T getSchoolName() {
return schoolName;
}

public void setSchoolName(T schoolName) {
this.schoolName = schoolName;
}

public T getNoOfStudents() {
return noOfStudents;
}

public void setNoOfStudents(T noOfStudents) {
this.noOfStudents = noOfStudents;
}

public T getNoOfFaculty() {
return noOfFaculty;
}

public void setNoOfFaculty(T noOfFaculty) {
this.noOfFaculty = noOfFaculty;
}

@Override
public String toString() {
return "\n{" +
"schoolName=" + schoolName +
", noOfStudents=" + noOfStudents +
", noOfFaculty=" + noOfFaculty +
'}';
}
}
package generics;

import java.util.*;
```

```
public class GenericExp16 {
public static void main(String[] args) {
HashSet<School<String>> schools = new
HashSet<School<String>>();
School school1 = new School("SchoolName1", 350, 20);
School school2 = new School("SchoolName2", 300, 22);
schools.add(school1);
schools.add(school2);
System.out.println("Schools details "+schools);
}
}
```

Output: Schools details [

{schoolName=SchoolName2, noOfStudents=300, noOfFaculty=22},

{schoolName=SchoolName1, noOfStudents=350, noOfFaculty=20}]

7. Generics by using Maps

```
package generics;

import java.util.HashMap;

public class Person<T, K> {
    HashMap<T, K> list = new HashMap<T, K>();

    public void putElement(T element, K number) {
        list.put(element, number);
    }

    public void removeElement(T element, K number) {
        list.remove(element, number);
    }

    @Override
    public String toString() {
        return "\n{" + list + '}';
    }

}

package generics;

import java.util.HashMap;
```

```
public class GenericExp17 {

    public static void main(String[] args) {
        Person<String, Integer> names = new Person<String,
Integer>();
        names.putElement("Name1", 01);
        names.putElement("Name2", 02);
        System.out.println("data is : " + names);

        names.removeElement("Name2", 02);
        System.out.println("After removing : " + names);
    }
}
```

Output: data is :

{{Name1=1, Name2=2}}

After removing :

{{Name1=1}}

Files: The File class of the java.io package is used to perform various operations on files and directories. To use the File class, create an object of the class, and specify the filename or directory name.

The File class has many methods for creating and getting information about files.

1. **createNewFile():** Creates an empty file
2. **canRead():** Tests whether the file is readable or not
3. **canWrite():** Tests whether the file is writable or not
4. **exists():** Tests whether the file exists
5. **getName():** Returns the name of the file
6. **getAbsolutePath():** Returns the absolute pathname of the file
7. **length():** Returns the size of the file in bytes
8. **list():** Returns an array of the files in the directory
9. **delete():** Deletes a file
10. **mkdir():** Creates a directory

Creating a file: To create a new file, we can use the **createNewFile()** method. It returns boolean value

→ true if a new file is created.

→ false if the file already exists in the specified location.

Note that the method is enclosed in a try...catch block. This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason).

```
package                                                  files;

import                                            java.io.File;
import                                     java.io.IOException;

public            class            CreateFile                 {
public     static     void     main(String[]     args)        {
try                                                           {
File    myObj    =    new    File("D:/Files/textfile.txt");
if                (myObj.createNewFile())                     {
System.out.println("File  created:  "  +  myObj.getName());
}                            else                             {
System.out.println("File          already          exists.");
}
}          catch          (IOException          e)            {
System.out.println("An          error          occurred.");
}
}
}
```

Output: For first Execution: File created: textfile.txt

For second Execution: File already exists.

Writing to a file: we use the **FileWriter** class together with its **write()** method to write some text to the file we created. Note that when you are done writing to the file, you should close it with the **close()** method.

```
package files;

import java.io.FileWriter;

import java.io.IOException;

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

String data = "This is the data in the file";
try {
FileWriter output = new
FileWriter("D:/Files/WriteFile.txt");

// Writes string to the file
output.write(data);
System.out.println("Data is written to the file.");

// Closes the writer
output.close();
}
catch (IOException e) {
System.out.println("An error occurred.");
e.getStackTrace();
}
}
}
```

Output: Data is written to the file.

Reading a file: we use the **Scanner** class to read the contents of the text file we created.

```
package files;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFromFile {
public static void main(String[] args) {
try {
File obj = new File("D:/Files/WriteFile.txt");
Scanner reader = new Scanner(obj);
while (reader.hasNextLine()) {
String data = reader.nextLine();
System.out.println(“Data in the file : “+data);
}
reader.close();
} catch (FileNotFoundException e) {
System.out.println("An error occurred.");
e.printStackTrace();
}
}
}
```

Output: Data in the file : This is the data in the file

Getting file information:

```
package files;

import java.io.File;

public class GetFileInfo {
public static void main(String[] args) {
File myObj = new File("D:/Files/WriteFile.txt");
if (myObj.exists()) {
System.out.println("File name: " + myObj.getName());
System.out.println("Absolute path: " +
myObj.getAbsolutePath());
System.out.println("Writeable: " + myObj.canWrite());
System.out.println("Readable " + myObj.canRead());
System.out.println("File size in bytes " +
myObj.length());
} else {
System.out.println("The file does not exist.");
}
}
}
```

Output:
```
File name: WriteFile.txt
Absolute path: D:\Files\WriteFile.txt
Writeable: true
Readable true
File size in bytes 29
```

Deteting a file: We can use the **delete()** method of the **<u>File</u>** class to delete the specified file or directory. It returns boolean value

→ true if the file is deleted.
→ false if the file does not exist.

```
package files;

import java.io.File;

public class DeleteFile {
```

```
public static void main(String[] args) {
File myObj = new File("D:/Files/WriteFile.txt");
if (myObj.delete()) {
System.out.println("Deleted the file: " +
myObj.getName());
} else {
System.out.println("Failed to delete the file.");
}
}
}
```

Output: Deleted the file: WriteFile.txt

Creating Directories: The Java File class provides the **mkdir()** and **mkdirs()** methods to create a new directory. The method returns Boolean value

→ true if the new directory is created
→ false if the directory already exists

```
package files;

import java.io.File;

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

// creates a file object with specified path
File file = new File("D:/Programs Direc");

// tries to create a new directory
boolean value = file.mkdir();

//boolean value = file.mkdirs();
if (value) {
System.out.println("The new directory is created.");
} else {
System.out.println("The directory already exists.");
}
}
}
```

Output: First Execution: The new directory is created.

Second Execution: The directory already exists.

Renaming a file: The Java File class provides the **renameTo()** method to change the name of the file. It returns true if the renaming operation succeeds otherwise returns false.

```
package files;
import java.io.File;
class RenameFile {
public static void main(String[] args) {
File file = new File("D:/Files/textfile.txt");
try {
file.createNewFile();
} catch (Exception e) {
e.getStackTrace();
}

//object that contains the new name of file
File newFile = new File("D:/Files/textfile1.txt");

// changing the name of file
boolean value = file.renameTo(newFile);

if (value) {
System.out.println("The name of the file is changed.");
} else {
System.out.println("The name cannot be changed.");
}
}
}
```

Output: The name of the file is changed.

Listing files: The **list()** method of the Java File class is used to list all the files and subdirectories present inside a directory. It returns all the files and directories as a string array.

```
package files;
import java.io.File;
class ListFile {
public static void main(String[] args) {
File file = new File("D:/Files");

// returns an array of all files
String[] fileList = file.list();

System.out.println("FIles in the folder ");
for (String str : fileList) {
System.out.println(str);
}
}
}
```

Output: FIles in the folder

textfile1.txt

WriteFile.txt

Coping files: The Java File class doesn't provide any method to copy one file to another. However, we can use Java I/O Streams to read content from one file and write to another.

```
package files;
import java.io.FileInputStream;
import java.io.FileOutputStream;
class CopyFIles {
    public static void main(String[] args) {

     byte[] array = new byte[50];
     try {
            FileInputStream sourceFile =
new
FileInputStream("D:/Copyfile.txt");
            FileOutputStream destFile = new
FileOutputStream("D:/Files/Copyfile.txt");

            // reads all data from input.txt
            sourceFile.read(array);

            // writes all data to newFile
            destFile.write(array);
            System.out.println("The input.txt file is
copied to
newFile.");

            // closes the stream
            sourceFile.close();
            destFile.close();
        } catch (Exception e) {
            e.getStackTrace();
```

```
                }
            }
        }
```

Output: The input.txt file is copied to newFile.

Java IO Stream: Java performs I/O through Streams. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Types of Streams: Depending upon the data a stream holds, it can be classified into

→ Byte Stream

→ Character Stream

Byte Stream: Byte stream is used to read and write a single byte (8 bits) of data. All byte stream classes are derived from base abstract classes called InputStream and OutputStream.

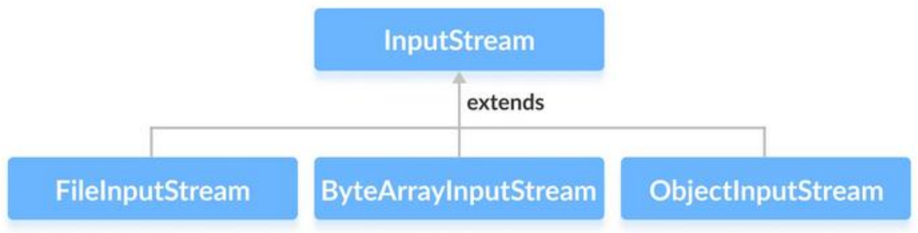
Java InputStream Class: An input stream is used to read data from the source. The InputStream class of the java.io package is an abstract superclass that represents an input stream of bytes. Since InputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

In order to use the functionality of InputStream, we can use its subclasses. Some of them are:

→ FileInputStream

→ ByteArrayInputStream

→ ObjectInputStream



Methods of inputStream:

1. **read()** - reads one byte of data from the input stream
2. **read(byte[] array)** - reads bytes from the stream and stores in the specified array
3. **available()** - returns the number of bytes available in the input stream
4. **mark()** - marks the position in the input stream up to which data has been read

5. **reset()** - returns the control to the point in the stream where the mark was set
6. **markSupported()** - checks if the mark() and reset() method is supported in the stream
7. **skips()** - skips and discards the specified number of bytes from the input stream.

FileInputStream: The FileInputStream class of the java.io package can be used to read data (in bytes) from files. It extends the InputStream abstract class.

Methods of FileInputStream:

1. **read()**: - reads a single byte from the file
2. **read(byte[] array):** - reads the bytes from the file and stores in the specified array read(byte[] array, int start, int length) - reads the number of bytes equal to length from the file and stores in the specified array starting from the position start
3. **Available():** To get the number of available bytes, we can use the available() method.
4. **close():** To close the file input stream, we can use the close() method. Once the close() method is called, we cannot use the input stream to read data.
5. **skip():** To discard and skip the specified number of bytes, we can use the skip() method.

Reading the file by using FileInputStream:

```
package files.inputstream;

import java.io.FileInputStream;

public class ReadFile {

    public static void main(String args[]) {

        try {
            FileInputStream input = new
FileInputStream("D:/Files/WriteFile.txt");

            System.out.println("Data in the file: ");

            // Reads the first byte
            int i = input.read();

            while (i != -1) {
                System.out.print((char) i);

                // Reads next byte from the file
                i = input.read();
            }
            input.close();
        } catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: Data in the file:

This is the data in the file

getting the number of available bytes:

```
package files.inputstream;

import java.io.FileInputStream;

public class CheckBytes {

    public static void main(String args[]) {

        try {
            FileInputStream input = new
FileInputStream("D:/Files/WriteFile.txt");

            System.out.println("Available bytes at the
beginning: "
+input.available());

            // Reads 3 bytes from the file
            input.read();
            input.read();
            input.read();

            // Returns the number of available bytes
            System.out.println("Available bytes at the
end: " + input.available());

            input.close();
        }

        catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: Available bytes at the beginning: 29

Available bytes at the end: 26

Skipping the number of specified bytes:

```
package files.inputstream;

import java.io.FileInputStream;

public class SkipBytes {

    public static void main(String args[]) {

        try {
            FileInputStream input = new
FileInputStream("D:/Files/WriteFile.txt");

            // Skips the 5 bytes
            input.skip(5);
            System.out.println("Input stream after
skipping 5 bytes:");

            // Reads the first byte
            int i = input.read();
            while (i != -1) {
                System.out.print((char) i);

                // Reads next byte from the file
                i = input.read();
            }
            input.close();
        } catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: Input stream after skipping 5 bytes:
is the data in the file

ByteArrayInputStream: The ByteArrayInputStream class of the java.io package can be used to read an array of input data (in bytes). It extends the InputStream abstract class. The ByteArrayInputStream class provides implementations for different methods present in the InputStream class.

ByteArrayInputStream to read data:

```
package files.inputstream;

import java.io.ByteArrayInputStream;

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

        // Creates an array of byte
        byte[] array = {1, 2, 3, 4};

        try {
            ByteArrayInputStream input = new
ByteArrayInputStream(array);

            System.out.print("The bytes read from the
input stream:
");

            for (int i = 0; i < array.length; i++) {

                // Reads the bytes
                int data = input.read();
                System.out.print(data + ", ");
            }
            input.close();
        } catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: The bytes read from the input stream: 1, 2, 3, 4,

getting the number of available bytes in the InputByteStream:

```
package files.inputstream;

import java.io.ByteArrayInputStream;

public class CheckByteArray {

    public static void main(String args[]) {

        // Creates an array of bytes
        byte[] array = {1, 2, 3, 4};

        try {
            ByteArrayInputStream input = new
ByteArrayInputStream(array);
            System.out.println("Available bytes at the
beginning: " + input.available());

            // Reads 2 bytes from the input stream
            input.read();
            input.read();
            System.out.println("Available bytes at the
end: " + input.available());
```

```
            input.close();
        } catch (Exception e) {
            e.getStackTrace();
        }
    }
}
```

Output: Available bytes at the beginning: 4

Available bytes at the end: 2

ObjectInputStream: The ObjectInputStream class of the java.io package can be used to read objects that were previously written by ObjectOutputStream. It extends the InputStream abstract class.

Methods of ObjectInputStream:

1. **read()** - reads a byte of data from the input stream
2. **readBoolean()** - reads data in boolean form
3. **readChar()** - reads data in character form
4. **readInt()** - reads data in integer form
5. **readObject()** - reads the object from the input stream

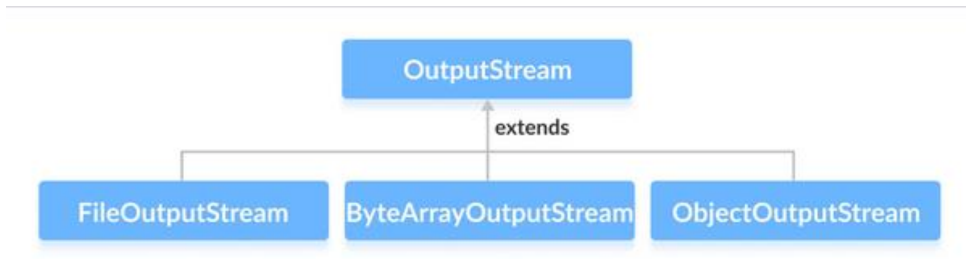
ObjectInputStream class to read objects written by the ObjectOutputStream class.

```
package files.inputstream;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

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

int data1 = 5;
String data2 = "This is data";

try {
FileOutputStream file = new
FileOutputStream("D:/Files/WriteFile.txt");
ObjectOutputStream output = new ObjectOutputStream(file);
output.writeInt(data1);
output.writeObject(data2);
FileInputStream fileStream = new
FileInputStream("D:/Files/WriteFile.txt");
ObjectInputStream objStream = new
ObjectInputStream(fileStream);

//Using the readInt() method
System.out.println("Integer data :" +
objStream.readInt());

// Using the readObject() method
System.out.println("String data: " +
objStream.readObject());

output.close();
objStream.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Integer data :5

String data: This is data

OutputStream: The OutputStream class of the java.io package is an abstract superclass that represents an output stream of bytes. Since OutputStream is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.



Methods of OutputStream:

1. **write()** - writes the specified byte to the output stream
2. **write(byte[] array)** - writes the bytes from the specified array to the output stream
3. **flush()** - forces to write all data present in output stream to the destination
4. **close()** - closes the output stream.

FileOutputStream: The FileOutputStream class of the java.io package can be used to write data (in bytes) to the files. It extends the OutputStream abstract class.

FileOutputStream to write data to a File

```
package files.outputstream;

import java.io.FileOutputStream;

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

String data = "Data inside the copyFIle";

try {
FileOutputStream output = new
FileOutputStream("D:/Files/Copyfile.txt");

byte[] array = data.getBytes();

// Writes byte to the file
output.write(array);
System.out.println("Data written inside the file : ");
output.close();
}
catch(Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data written inside the file :

ByteArrayOutputStream: The ByteArrayOutputStream class of the java.io package can be used to write an array of output data (in bytes). It extends the OutputStream abstract class.

Methods of ByteArrayOutputStream:

1. **write(int byte)** - writes the specified byte to the output stream
2. **write(byte[] array)** - writes the bytes from the specified array to the output stream
3. **write(byte[] arr, int start, int length)** - writes the number of bytes equal to length to the output stream from an array starting from the position start

4. **writeTo(ByteArrayOutputStream out1)** - writes the entire data of the current output stream to the specified output stream
5. **toByteArray()** - returns the array present inside the output stream
6. **toString()** - returns the entire data of the output stream in string form.

ByteArrayOutputStream to write data:

```
package files.outputstream;

import java.io.ByteArrayOutputStream;

class WriteToFile1 {
public static void main(String[] args) {

String data = "This is a line of text inside the file.";

try {
// Creates an output stream
ByteArrayOutputStream out = new ByteArrayOutputStream();
byte[] array = data.getBytes();

// Writes data to the output stream
out.write(array);

String streamData = out.toString();
System.out.println("Output stream: " + streamData);

out.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Output stream: This is a line of text inside the file.

ObjectOutputStream: The ObjectOutputStream class of the java.io package can be used to write objects that can be read by ObjectInputStream. It extends the OutputStream abstract class.

```
package files.outputstream;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

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

int data1 = 5;
String data2 = "This is data";

try {
FileOutputStream file = new
FileOutputStream("D:/Files/Copyfile1.txt");
ObjectOutputStream output = new ObjectOutputStream(file);

output.writeInt(data1);
output.writeObject(data2);

// Reads data using the ObjectInputStream
FileInputStream fileStream = new
FileInputStream("D:/Files/Copyfile1.txt");
ObjectInputStream objStream = new
ObjectInputStream(fileStream);

System.out.println("Integer data :" +
objStream.readInt());
System.out.println("String data: " +
objStream.readObject());

output.close();
objStream.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Integer data :5

String data: This is data

ObjectOutputStream by using Serializable interface:

```
package files.outputstream;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

class ObjOutputStr1 {
public static void main(String[] args) {

int data1 = 5;
String data2 = "This is data";

try {
FileOutputStream file = new
FileOutputStream("D:/Files/Copyfile1.txt");
ObjectOutputStream output = new ObjectOutputStream(file);

output.writeInt(data1);
output.writeObject(data2);

// Reads data using the ObjectInputStream
FileInputStream fileStream = new
FileInputStream("D:/Files/Copyfile1.txt");
ObjectInputStream objStream = new
ObjectInputStream(fileStream);

System.out.println("Integer data :" +
objStream.readInt());
System.out.println("String data: " +
objStream.readObject());

output.close();
objStream.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

OutPut: Integer data :5

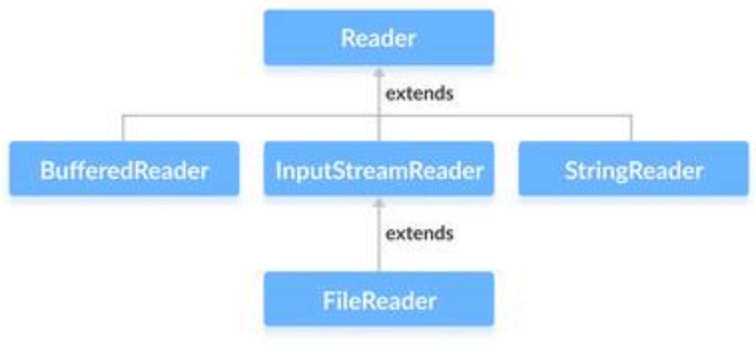
String data: This is data

Character Stream: Character stream is used to read and write a single character of data. All the character stream classes are derived from base abstract classes Reader and Writer.

Reader Class: The Reader class of the java.io package is an abstract superclass that represents a stream of characters. Since Reader is an abstract class, it is not useful by itself. However, its subclasses can be used to read data.

Subclasses of Reader: In order to use the functionality of Reader, we can use its subclasses. Some of them are:

→ BufferedReader
→ InputStreamReader
→ FileReader
→ StringReader



Methods of Reader:

1. **ready()** - checks if the reader is ready to be read
2. **read(char[] array)** - reads the characters from the stream and stores in the specified array
3. **read(char[] array, int start, int length)** - reads the number of characters equal to length from the stream and stores in the specified array starting from the start.
4. **mark()** - marks the position in the stream up to which data has been read.
5. **reset()** - returns the control to the point in the stream where the mark is set.
6. **skip()** - discards the specified number of characters from the stream.

BufferReader: The BufferedReader class of the java.io package can be used with other readers to read data (in characters) more efficiently. It extends the abstract class Reader. The BufferedReader maintains an internal buffer of 8192 characters.

During the read operation in BufferedReader, a chunk of characters is read from the disk and stored in the internal buffer. And from the internal buffer characters are read individually. Hence, the number of communication to the disk is reduced. This is why reading characters is faster using BufferedReader.

Reading the file by using BufferReader:

```
package files.readerclass;

import java.io.FileReader;
import java.io.BufferedReader;

public class ReaderFile {

public static void main(String args[]) {

char[] array = new char[100];
try {
FileReader file = new
FileReader("D:/Files/WriteFile.txt");

// Creates a BufferedReader
BufferedReader input = new BufferedReader(file);
input.skip(5);
input.read(array);

System.out.println("Data after skipping 5 characters:");
System.out.println(array);
input.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output:
```
Data after skipping 5 characters:
is the data in the  file
```

InputStreamReader: The InputStreamReader class of the java.io package can be used to convert data in bytes into data in characters. It extends the abstract class Reader.

The InputStreamReader class works with other input streams. It is also known as a bridge between byte streams and character streams. This is because the InputStreamReader reads bytes from the input stream as characters.

For example, some characters required 2 bytes to be stored in the storage. To read such data we can use the input stream reader that reads the 2 bytes together and converts into the corresponding character.

Reading and encoding the file:

```
package files.readerclass;

import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.nio.charset.Charset;

class Main {
public static void main(String[] args) {
char[] array = new char[100];

try {
FileInputStream file = new
FileInputStream("D:/Files/WriteFile.txt");
InputStreamReader input = new InputStreamReader(file);
input.read(array);
System.out.println("Data in the stream:");
System.out.println(array);

// Creates an InputStreamReader specifying the encoding
InputStreamReader input1 = new InputStreamReader(file,
Charset.forName("UTF8"));

// Returns the character encoding of the input stream
System.out.println("Character encoding of input1: " +
input.getEncoding());
System.out.println("Character encoding of input2: " +
input1.getEncoding());
input.close();
}

catch(Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data in the stream:

```
This is the data in the  file
Character encoding of input1: UTF8
Character encoding of input2: UTF8
```

FileReader: The FileReader class of the java.io package can be used to read data (in characters) from files. It extends the InputStreamReader class.

Reading and encoding file by using FileReader:

```
package files.readerclass;

import java.io.FileReader;
import java.nio.charset.Charset;

class FileReader1 {
public static void main(String[] args) {

char[] array = new char[100];
try {

FileReader input = new
FileReader("D:/Files/WriteFile.txt");
Input.read(array);
System.out.println("Data in the file: ");
System.out.println(array);
FileReader input1 = new
FileReader("D:/Files/WriteFile.txt", Charset.forName("
UTF8"));

// Returns the character encoding of the file reader
System.out.println("Character encoding of input1: " +
input.getEncoding());
System.out.println("Character encoding of input2: " +
input1.getEncoding());
input.close();

Input1.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data in the file:

This is the data in the file

Character encoding of input1: UTF8

Character encoding of input2: UTF8

StringReader: The StringReader class of the java.io package can be used to read data (in characters) from strings. It extends the abstract class Reader. In StringReader, the specified string acts as a source from where characters are read individually.

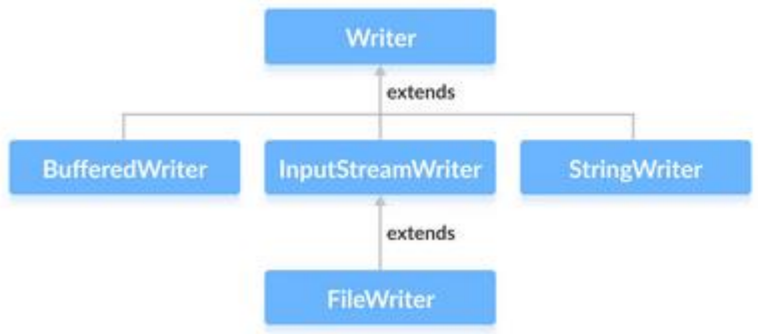
StringReader Example

```
package files.readerclass;

import java.io.StringReader;

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

String data = "This is the text from StringReader.";
char[] array = new char[100];

try {
StringReader input = new StringReader(data);

//Use the read method
input.read(array);
System.out.println("Data read from the string:");
System.out.println(array);

input.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data read from the string:

This is the text from StringReader.

Writer Class: The Writer class of the java.io package is an abstract superclass that represents a stream of characters. Since Writer is an abstract class, it is not useful by itself. However, its subclasses can be used to write data.



Methods of Writer:

1. **write(char[] array)** - writes the characters from the specified array to the output stream
2. **write(String data)** - writes the specified string to the writer.
3. **append(char c)** - inserts the specified character to the current writer
4. **flush()** - forces to write all the data present in the writer to the corresponding destination
5. **close()** - closes the writer.

BufferedWriter: The BufferedWriter class of the java.io package can be used with other writers to write data (in characters) more efficiently. It extends the abstract class Writer. The BufferedWriter maintains an internal **buffer of 8192 characters.**

During the write operation, the characters are written to the internal buffer instead of the disk. Once the buffer is filled or the writer is closed, the whole characters in the buffer are written to the disk.

Hence, the number of communication to the disk is reduced. This is why writing characters is faster using BufferedWriter.

Writing the data to the file:

```
package files.writerclass;

import java.io.FileWriter;
import java.io.BufferedWriter;

public class WriteToFIle {
```

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

String data = "This is the data in the output file";

try {
FileWriter file = new
FileWriter("D:/Files/WriteFile.txt");
BufferedWriter output = new BufferedWriter(file);

// Writes the string to the file
output.write(data);
System.out.println("Data is written to the file.");
output.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data is written to the file.

OutputStreamWriter: The OutputStreamWriter class of the java.io package can be used to convert data in character form into data in bytes form. It extends the abstract class Writer.

OutputStreamWriter to write data to a File:

```
package files.writerclass;

import java.io.FileOutputStream;
import java.io.OutputStreamWriter;

public class WriteToFile1 {

public static void main(String args[]) {

String data = "This is a line of text inside the file.";

try {
FileOutputStream file = new
FileOutputStream("D:/Files/WriteFile.txt");
OutputStreamWriter output = new OutputStreamWriter(file);
output.write(data);
System.out.println("Data is written to the file.");
output.close();
}

catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data is written to the file.

FileWriter: The FileWriter class of the java.io package can be used to write data (in characters) to files. It extends the OutputStreamWriter class.

However, since Java 11 we can specify the type of character encoding (UTF8 or UTF16) as well.

FileWriter to write and encoding data to a File:

```
package files.writerclass;

import java.io.FileWriter;
import java.nio.charset.Charset;

public class FileWriter1 {

public static void main(String args[]) {

String data = "This is the data in the output file";

try {
FileWriter output = new
FileWriter("D:/Files/WriteFile.txt");
output.write(data);
System.out.println("Data is written to the file.");
FileWriter output1 = new
FileWriter("D:/Files/WriteFile.txt",
Charset.forName("UTF8"));

System.out.println("Character encoding of output1: " +
output.getEncoding());
System.out.println("Character encoding of output2: " +
output1.getEncoding());

output.close();
output1.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data is written to the file.

Character encoding of output1: UTF8

Character encoding of output2: UTF8

StringWriter: The StringWriter class of the java.io package can be used to write data (in characters) to the string buffer. It extends the abstract class Writer.

In Java, string buffer is considered as a mutable string. That is, we can modify the string buffer. To convert from string buffer to string, we can use the toString() method.

StringWriter Example:

```
package files.writerclass;

import java.io.StringWriter;

public class StringWriter1 {
public static void main(String[] args) {

String data = "This is the text in the string.";

try {
StringWriter output = new StringWriter();
output.write(data);
System.out.println("Data in the StringWriter: " + output);
StringBuffer stringBuffer = output.getBuffer();
System.out.println("StringBuffer: " + stringBuffer);

// Returns the string buffer in string form
String string = output.toString();
System.out.println("String: " + string);
output.close();
}

catch(Exception e) {
e.getStackTrace();
}
}
}
```

Output: Data in the StringWriter: This is the text in the string.

StringBuffer: This is the text in the string.

String: This is the text in the string.

PrintWriter: The PrintWriter class of the java.io package can be used to write output data in a commonly readable form (text). It extends the abstract class Writer.

Unlike other writers, PrintWriter converts the primitive data (int, float, char, etc.) into the text format. It then writes that formatted data to the writer.

Also, the PrintWriter class does not throw any input/output exception. Instead, we need to use the checkError() method to find any error in it.

The PrintWriter class also has a feature of auto flushing. This means it forces the writer to write all data to the destination if one of the println() or printf() methods is called.

Methods of PrintWriter:

1. **print()** - prints the specified data to the writer.
2. **println()** - prints the data to the writer along with a new line character at the end.
3. **printf()** - The printf() method can be used to print the formatted string. It includes 2 parameters: formatted string and argument.

PrintWriter Example1

```
package files.printWriter;

import java.io.PrintWriter;

class PrintWriter1 {
public static void main(String[] args) {

String data = "This is a text inside the file.";

try {
PrintWriter output = new
PrintWriter("D:/Files/print1.txt");
int age = 25;
output.print(data);
output.printf("I am %d years old.", age);
System.out.println("Successfully printed in the file ");
output.close();
} catch (Exception e) {
e.getStackTrace();
}
}
}
```

Output: Successfully printed in the file

PrintWriter Example2

```
package files.printWriter;

import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriter2 {

public static void main(String[] args) throws IOException
{
FileWriter fw = new FileWriter("D:/Files/print.txt");
PrintWriter out = new PrintWriter(fw);
out.write(97);
out.println(100);
out.println(true);
out.print('c');
out.println("code in side the file");
System.out.println("Successfully printed in the file ");
out.close();
}
}
```

Output: Successfully printed in the file.

Example for counting lines of a particular file.

```
package files.realtimeexp;

import java.io.*;

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

```
try {
BufferedReader bf = new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Please enter file name with
extension:");
String str = bf.readLine();
File file = new File(str);
if (file.exists()) {
FileReader fr = new FileReader(file);
LineNumberReader ln = new LineNumberReader(fr);
int count = 0;
while (ln.readLine() != null) {
count++;
}
System.out.println("Total line no: " + count);
ln.close();
} else {
System.out.println("File does not exists!");
}
} catch (IOException e) {
e.printStackTrace();
}
}
}
```

Output: Please enter file name with extension:

D:\Intellij\src\files\realtimeexp/CountLines.java

Total line no: 28

Creating a temporary File

```
package files.realtimeexp;

import java.io.File;
import java.io.IOException;

public class TempFile {
    public static void main(String[] args) {
        File tempFile = null;

        try {
            tempFile = File.createTempFile("MyFile.txt",
".tmp");
```

```
            System.out.print("Created temporary file with
name\n ");

System.out.println(tempFile.getAbsolutePath());
        } catch (IOException ex) {

            System.err.println("Cannot create temp file: "
+ ex.getMessage());
        } finally {
            if (tempFile != null) {
            }
        }
    }
}
```

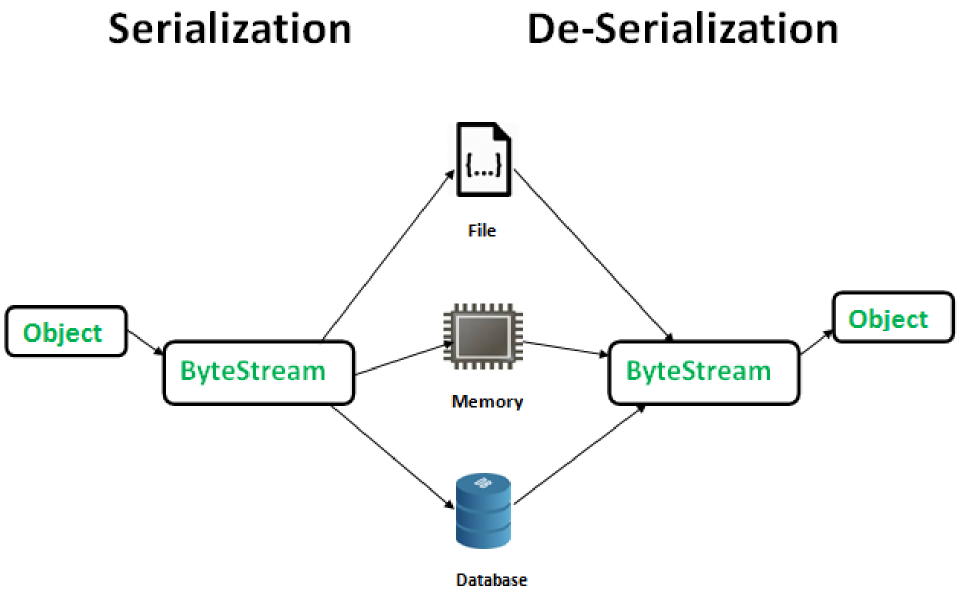
Output: Created temporary file with name

C:\Users\manisai\AppData\Local\Temp\MyFile.txt14461912954368542791.tmp

Serialization and Deserialization: **Serialization** is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

A class must implement **Serializable** interface present in java.io package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behavior to the class implementing it.

To implement serialization and deserialization, Java provides two classes ObjectOutputStream and ObjectInputStream.



Example for Serialization:

```
package files.serialization;

import java.io.Serializable;

class StudentInfo implements Serializable {
String name;
int rNo;
String standard;

StudentInfo(String n, int r, String s) {
this.name = n;
this.rNo = r;
this.standard = s;

}
}
package files.serialization;

import files.serialization.StudentInfo;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

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

```
StudentInfo si = new StudentInfo("Name", 101, "Fifth");
FileOutputStream file = new
FileOutputStream("D:/Files/student.txt");
ObjectOutputStream oos = new ObjectOutputStream(file);
oos.writeObject(si);
System.out.println("Data stored in the file");
System.out.println("Student Name : "+si.name);
System.out. println("Roll no : "+si.rNo);
System.out.println("Standard : "+si.standard);
oos.flush();
oos.close();
} catch (Exception e) {
System.out.println(e);
}
}
}
```

Output:
```
Data stored in the file

Student Name : Name

Roll no : 101

Standard : Fifth
```

Example for Deserialization:

```
package files.serialization;

import java.io.Serializable;

class StudentInfo implements Serializable {
String name;
int rNo;
String standard;

StudentInfo(String n, int r, String s) {
this.name = n;
this.rNo = r;
this.standard = s;

}
}
package files.serialization;

import files.serialization.StudentInfo;
import java.io.FileInputStream;
import java.io.ObjectInputStream;


class Main1 {
public static void main(String[] args) {
StudentInfo si = null;
try {
FileInputStream fis = new
            FileInputStream("D:/Files/student.txt");
ObjectInputStream ois = new ObjectInputStream(fis);
si = (StudentInfo) ois.readObject();
} catch (Exception e) {
e.printStackTrace();
}
System.out.println("Student Name : " + si.name);
System.out.println("Roll no : " + si.rNo);
System.out.println("Standard : " + si.standard);
}
}
```

Output: Student Name : Name

Roll no : 101

Standard : Fifth

Example for serialization and deserialization

```
package files.serialization;

import java.io.*;

public class Employee1 implements Serializable {
int id;
String name;

public Employee1(int id, String name) {
this.id = id;
this.name = name;
}

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

```
Employee1 emp1 = new Employee1(211, "Name1");
Employee1 emp2 = new Employee1(212, "Name2");
Employee1 emp3 = new Employee1(213, "Name3");
FileOutputStream fout = new
FileOutputStream("D:/Files/Emp.txt");
ObjectOutputStream out = new ObjectOutputStream(fout);
out.writeObject(emp1);
out.writeObject(emp2);
out.writeObject(emp3);
out.flush();
out.close();
System.out.println("Serialization is successfully
executed");
} catch (Exception e) {
System.out.println(e);
}
try {
ObjectInputStream in = new ObjectInputStream(new
FileInputStream("D:/Files/Emp.txt"));
Employee1 e1 = (Employee1) in.readObject();
Employee1 e2 = (Employee1) in.readObject();
Employee1 e3 = (Employee1) in.readObject();
System.out.println("Deserialization");
System.out.println(e1.id + " " + e1.name);
System.out.println(e2.id + " " + e2.name);
System.out.println(e3.id + " " + e3.name);
in.close();
} catch (Exception e) {
System.out.println(e);
}
}
}
```

Output: Serialization is successfully executed

Deserialization

211 Name1

212 Name2

213 Name3

Lambda Expressions: The lambda expression was introduced for the first time in Java 8. Its main objective is to increase the expressive power of the language.

A lambda expression is an anonymous function. A function that doesn't have a name and doesn't belong to any class. The concept of lambda expression was first introduced in LISP programming language.

But, before getting into lambdas, we first need to understand functional interfaces.

Functional Interface: If a Java interface contains one and only one abstract method then it is termed as functional interface. This only one method specifies the intended purpose of the interface. For example, the Runnable interface from package java.lang; is a functional interface because it constitutes only one method i.e. run().

```
package              lambdaexpressions.functionalinterfaces;

import                            java.lang.FunctionalInterface;

@FunctionalInterface
public           interface           MyInterf             {
//         the         single        abstract         method
double                                           getValue();
}
```

In the above example, the interface MyInterface has only one abstract method getValue(). Hence, it is a functional interface.

In Java 7, functional interfaces were considered as Single Abstract Methods or SAM type. SAMs were commonly implemented with Anonymous Classes in Java 7.

Implement SAM with anonymous classes in java

```
package lambdaexpressions.functionalinterfaces;

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

// anonymous class
new Thread(new Runnable() {
@Override
public void run() {
System.out.println("Runnable Functional Interface.");
}
}).start();
}
}
```

OutPut: Runnable Functional Interface.

Defining the lambda expressions: The new operator (->) used is known as an arrow operator or a lambda operator. The syntax might not be clear at the moment.

```
(parameter list) -> lambda body
```

Suppose, we have a method like this:

```
double getPiValue() {

return 3.1415;

}
```

We can write this method using lambda expression as:

```
() -> 3.1415
```

In Java, the lambda body is of two types.

A body with a single expression:

```
() -> System.out.println("Lambdas are great");
```

A body that consists of a block of code.

```
() -> {
double pi = 3.1415;
  return pi;
};
```

This type of the lambda body is known as a block body. The block body allows the lambda body to include multiple statements. These statements are enclosed inside the braces and you have to add a semi-colon after the braces.

Above Example by using the Lambda expression:

```
package lambdaexpressions;


import java.lang.FunctionalInterface;

@FunctionalInterface
interface MyInterface{

// abstract method
double getPiValue();
}

public class LambdaExp1 {

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

```
// declare a reference
MyInterface ref;

// lambda expression
ref = () -> 3.1415;
//MyInterface ref = () -> 3.1415;

System.out.println("Value of Pi = " + ref.getPiValue());
}
}
```

Output: Value of Pi = 3.1415

Using lambda expression with parameters:

```
package lambdaexpressions;

import java.lang.FunctionalInterface;

@FunctionalInterface
interface MyInterface1 {

// abstract method
String reverse(String n);
}

public class ReverseStr {

public static void main(String[] args) {

// declare a reference to MyInterface
// assign a lambda expression to the reference
MyInterface1 ref = (str) -> {

String result = "";
for (int i = str.length() - 1; i >= 0; i--)
result += str.charAt(i);
return result;
};

// call the method of the interface
System.out.println("reversed = " +
ref.reverse("Hyderabad"));
}
}
```

Output: reversed = dabaredyH

Lambda Expression with Multiple Parameters:

```
package lambdaexpressions;

import java.lang.FunctionalInterface;

@FunctionalInterface
interface StringConcat {

// Abstract method
public String concat(String a, String b);
}

public class LambdaExp2 {

public static void main(String args[]) {
// lambda expression with multiple arguments
StringConcat s = (str1, str2) -> str1 + str2;
System.out.println("Result: " + s.concat("Lambda ",
"Expression"));
}
}
```

Output: Result: Lambda Expression

Iterating collections using Lambda Expression:

```
package lambdaexpressions;

import java.util.*;
public class ListExp{
public static void main(String[] args) {
List<String> list=new ArrayList<String>();
list.add("Virat");
list.add("Rohit");
list.add("Rahul");
list.add("Bumrah");
list.add("Shami");
System.out.println("List of Indian cricketer's :");
list.forEach(
// lambda expression
(names)->System.out.println(names)
);
}
}
```

Output: List of Indian cricketer's :

Virat

Rohit

Rahul

Bumrah

Shami

Iterating Map by using Lambda expression:

```
package lambdaexpressions;

import java.util.HashMap;
import java.util.Map;

public class MapExp {
public static void main(String[] args) {
Map<String, Integer> prices = new HashMap<>();
prices.put("Apple", 50);
prices.put("Orange", 20);
prices.put("Banana", 10);
prices.put("Grapes", 40);
prices.put("Papaya", 50);
System.out.println("Fruits and their Prices : ");

prices.forEach((k, v) -> System.out.println(" " + k + ",
Price: " + v));

}
}
```

Output: Fruits and their Prices :

Apple, Price: 50

Grapes, Price: 40

Papaya, Price: 50

Orange, Price: 20

```
Banana, Price: 10
```

Calculator by using Lambda expressions:

```
package lambdaexpressions;

import java.lang.FunctionalInterface;

@FunctionalInterface
interface MathOperation {
int operation(int a, int b);
}

public class LambdaExp3 {

private int operate(int a, int b, MathOperation
mathOperation) {
return mathOperation.operation(a, b);
}

public static void main(String args[]) {
LambdaExp3 tester = new LambdaExp3();

//with type declaration
MathOperation addition = (int a, int b) -> a + b;

//without type declaration
MathOperation subtraction = (a, b) -> a - b;

//with return statement along with curly braces
MathOperation multiplication = (int a, int b) -> {
return a * b;
};

//without return statement and without curly braces
MathOperation division = (int a, int b) -> a / b;

System.out.println("10 + 5 = " + tester.operate(10, 5,
addition));
System.out.println("10 - 5 = " + tester.operate(10, 5,
subtraction));
System.out.println("10 x 5 = " + tester.operate(10, 5,
multiplication));
System.out.println("10 / 5 = " + tester.operate(10, 5,
division));

}
}
```

Output: 10 + 5 = 15

10 - 5 = 5

10 x 5 = 50

10 / 5 = 2

Sorting elements by using the Lambda Expressions:

```
package lambdaexpressions;

public class Movie {

String movieName;
long duration;

public Movie(String movieName, long duration) {
this.movieName = movieName;
this.duration = duration;
}

public String getMovieName() {
return movieName;
}

public void setMovieName(String movieName) {
this.movieName = movieName;
}

public long getDuration() {
return duration;
}

public void setDuration(long duration) {
this.duration = duration;
}

@Override
public String toString() {
return "Movie Name: " + this.getMovieName() + "|| "
+ "Movie duration: " + this.getDuration();

}
}
package lambdaexpressions;

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

public class LambdaExp4 {

public static void main(String[] args) {

Movie m1 = new Movie("Inception", 110);
Movie m2 = new Movie("GodFather", 200);
Movie m3 = new Movie("Forest Gump", 130);
Movie m4 = new Movie("Avengers", 150);

List<Movie> listOfMovies = new ArrayList<>();
listOfMovies.add(m1);
listOfMovies.add(m2);
listOfMovies.add(m3);
listOfMovies.add(m4);

// Using lambda expression
Collections.sort(listOfMovies, (o1, o2) ->
o1.getMovieName().compareTo(o2.getMovieName()));

System.out.println("After Sort by name: ");
for (int i = 0; i < listOfMovies.size(); i++) {
Movie movie = (Movie) listOfMovies.get(i);
System.out.println(movie);
}
}
}
```

Output:
```
After Sort by name:

Movie Name: Avengers|| Movie duration: 150

Movie Name: Forest Gump|| Movie duration: 130

Movie Name: GodFather|| Movie duration: 200
Movie Name: Inception|| Movie duration: 110
```

Sorting elements by using the Lambda Expressions:

```
package lambdaexpressions;

class Person {
private String name;
private int age;

public Person(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

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

public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
}
package lambdaexpressions;

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

public class LambdaExp5 {

public static void main(String[] args) {

List<Person> listOfPerson = new ArrayList<Person>();
listOfPerson.add(new Person("Name1", 27));
listOfPerson.add(new Person("Name2", 26));
listOfPerson.add(new Person("Name3", 28));
listOfPerson.add(new Person("Name4", 27));

Collections.sort(listOfPerson, (Person o1, Person o2) -> {
return o1.getAge() - o2.getAge();
});
System.out.println(" sort persons by age in ascending
order by their ages ");
listOfPerson.forEach(
(person) -> System.out.println(" Person name : " +
person.getName()));
}
}
```

Output: sort persons by age in ascending order by their ages

Person name : Name2

Person name : Name1

Person name : Name4

Person name : Name3

Generic Functional interface by using Lambda expression:

```
package lambdaexpressions;

interface MyFunc<T> {
T func(T t);
}

public class LambdaExp6 {
public static void main(String args[]) {

// Use a String-based version of MyFunc.
MyFunc<String> reverse = (str) -> {
StringBuilder sb = new StringBuilder(str);
return sb.reverse().toString();
};

System.out.println("Reverse of Generics :
"+reverse.func("Generics"));

// use an Integer-based version of MyFunc.
MyFunc<Integer> factorial = (n) -> {
```

```
int result = 1;

for (int i = 1; i <= n; i++)
result = i * result;

return result;
};

System.out.println("The factoral of 5 is " +
factorial.func(5));
}
}
```

Output: Reverse of Generics : scireneG

The factoral of 5 is 120

Function: The Java Function interface (java.util.function.Function) interface is one of the most central functional interfaces in Java. The Function interface represents a function (method) that takes a single parameter T and returns a single value as R.

```
package lambdaexpressions.functionalinterfaces;

import java.util.function.Function;

public class FunctionInterf {
public static void main(String[] args) {
Function<Integer,Double> squareRootGenerator = (input) -> {return
Math.sqrt(input);};
int input = 16;
double result = squareRootGenerator.apply(input);
System.out.println("Square Root of "+input+" is "+result);
}
}
```

Output: Square Root of 16 is 4.0

This Function implementation accepts an integer number and returns its square root. Function<Integer, Double> specifies that the input is of type Integer, and output is of type Double. The lambda expression needs to be assigned to an instance of the functional interface.

Supplier Interface: The Java Supplier interface is a functional interface that represents an function that supplies a value of some sorts. The Supplier interface can also be thought of as a factory interface. The supplier has a single method called get. It does not accept any arguments. It returns an object of any data type.

```
package lambdaexpressions.functionalinterfaces;

import java.util.Random;
import java.util.function.Supplier;

public class SupplierInterf {

public static void main(String[] args) {
Supplier<Integer[]> randomNumberGenerator = () -> {
Integer[] result = new Integer[5];
for (int i = 0; i < result.length; i++) {
result[i] = new Random().nextInt(100);
}
return result;
};
System.out.println("Result:");
Integer[] result = randomNumberGenerator.get();
for (int num : result)
System.out.println(num);
}
}
```

Output:

```
Result:

36

85

4

14

68
```

This Supplier returns an Integer array of 5 random numbers. As before, the get method is implemented via a lambda expression which creates an Integer array of length 5 and populates it with 5 random numbers.

Consumer Interface: The Java Consumer interface is a functional interface that represents an function that consumes a value without returning any value. A Java Consumer implementation could be printing out a value, or writing it to a file, or over the network etc.

```
package lambdaexpressions.functionalinterfaces;

public class Person {

private String name;
private int age;

Person(String name, int age) {
this.name = name;
this.age = age;
}

public String getName() {
return name;
}

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

public int getAge() {
return age;
}

public void setAge(int age) {
this.age = age;
}
}
package lambdaexpressions.functionalinterfaces;

import java.util.function.Consumer;

public class ConsumerInterf {
public static void main(String[] args) {
Consumer<Person> ageIncr = (input) -> {
int newAge = input.getAge() + 5;
input.setAge(newAge);
};

Person person = new Person("abc", 25);
ageIncr.accept(person);
System.out.println("New age:" + person.getAge());
}
}
```

Output: New age:30

Predicate: The Java Pred icate interface, java.util.function.Predicate, represents a simple function that takes a single value as parameter, and returns true or false.

Methods in Predicate Interface:

1. **boolean test(T t) –** This method takes a single generic argument and returns **true** or **false**
2. **default Predicate<T> and(Predicate<? super T> other) –** This is a default method, returns a composed predicate that by performing **short-circuiting logical AND** of **current predicate** and **another predicate**.
3. **default Predicate<T> or(Predicate<? super T> other) –** This is also a default method, returns a composed predicate that by performing **short-circuiting logical OR** of **current predicate** and **another predicate.**
4. **default Predicate<T> negate() –** This also a default method, returns a predicate after performing logical negation(!) on the **current predicate**.
5. **static <T> Predicate<T> isEqual(Object targetRef) –** This static method returns a predicate which test the equality of the arguments passed.

```
package lambdaexpressions.functionalinterfaces;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

public class PredicateInterf1 {
public static void main(String[] args) {
List<Integer> numbers = Arrays.asList(12, 2, 4, 1, 2, 0,
9, 3, 5);
Predicate<Integer> checker = number -> number > 3;

System.out.println("Numbers greater than 3 ");
numbers.stream().filter(number -> checker.test(number))
.forEach(System.out::println);
}
}
```

Output: Numbers greater than 3

```
12

4

9

5
```

predicate that gives us the name of the Student with mark greater than 50.

```
package lambdaexpressions.functionalinterfaces;

public class Student {
private int id;
private int mark;
private String name;

public Student(int id, int mark, String name) {
super();
this.id = id;
this.mark = mark;
this.name = name;
}

public int getId() {
return id;
}

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

public int getMark() {
return mark;
}

public void setMark(int mark) {
this.mark = mark;
}

public String getName() {
return name;
}

public void setName(String name) {
```

```
this.name = name;
}
}
package lambdaexpressions.functionalinterfaces;

import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

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

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

studentList.add(new Student(1, 45, "Rajesh"));
studentList.add(new Student(2, 65, "Sravan"));
studentList.add(new Student(3, 80, "Raj"));
studentList.add(new Student(4, 20, "Avi"));

Predicate<Integer> markPredicate = mark -> mark > 50 ;

System.out.println("Student with marks greater than 50");
studentList.stream()
.filter(student -> markPredicate.test(student.getMark()))
.forEach(student ->
System.out.println(student.getName()));
}
}
```

Output: Student with marks greater than 50

Sravan

Raj

UnaryOperator: The Java UnaryOperator interface is a functional interface that represents an operation which takes a single parameter and returns a parameter of the same type. The apply() method from Function interface and default methods: andThen() and compose() are inherited from the UnaryOperator interface.

The UnaryOperator interface can be used to represent an operation that takes a specific object as parameter, modifies that object, and returns it again - possibly as part of a functional stream processing chain.

Methods of unaryOperator:

1. **andThen(Function<? super R, ? extends V> after)**– It is a default interface method which takes another Function as argument and returns a composed Function that performs, in sequence, first the operation of the calling Function followed by the after operation.
2. **compose(Function<? super V, ? extends T> before)**– It is a default method in Function interface which takes another Function as argument and returns a composed Function that performs, in sequence, first the before operation then the operation of the calling Function.
3. **R apply(T t)**– Here T is the type of the argument passed to the method and it returns a value of type R. This is the abstract method in this functional interface. If you are writing a Lambda expression that takes single argument of one type and returns a value of the same type then that lambda expression can be written as an implementation of UnaryOperator built-in functional interface where lambda expression implements the apply() method.

```
package lambdaexpressions.functionalinterfaces;

import java.util.function.UnaryOperator;

public class UnaryOpInterf {
public static void main(String[] args) {
UnaryOperator<Integer> operator1 = t -> t + 5;
UnaryOperator<Integer> operator2 = t -> t * 5;
System.out.println("Using andthen method ");
// Using andThen() method
int a = operator1.andThen(operator2).apply(5);
System.out.println(a);
int b = operator1.andThen(operator2).apply(3);
System.out.println(b);
System.out.println("Using compose method ");
//using compose() method
int c = operator1.compose(operator2).apply(5);
System.out.println(c);
int d = operator1.compose(operator2).apply(3);
System.out.println(d);
}
}
```

Output: Using andthen method

```
50

40

Using compose method

30

20
```

BinaryOperator: The Java BinaryOperator interface is a functional interface that represents an operation which takes two parameters and returns a single value. Both parameters and the return type must be of the same type. It is useful when implementing functions that sum, subtract, divide, multiply etc. two elements of the same type, and returns a third element of the same type.

```
package lambdaexpressions.functionalinterfaces;

import java.util.function.BinaryOperator;

public class BinaryOperInterf1 {
    public static void main(String[] args) {
        BinaryOperator<Integer> operator1 = (a, b) -> a +
b;
        System.out.println("Sum : "+operator1.apply(5,
7));

        BinaryOperator<String> operator2 = (s1, s2) -> s1
+ s2;
        System.out.println("Concate :
"+operator2.apply("Binary ", "Operator"));
    }
}
```

Output:
```
Sum : 12

Concate : Binary Operator
```

Binary operator by using comparator:

```
package lambdaexpressions.functionalinterfaces;

import java.util.Comparator;
import java.util.function.BinaryOperator;

public class BinaryOperInterf2 {
    public static void main(String[] args) {

        Comparator<Integer> comparator = (a, b) ->
(a.compareTo(b));

        // Using maxBy()
        BinaryOperator<Integer> opMax =
BinaryOperator.maxBy(comparator);
        System.out.println("Max: " + opMax.apply(5, 6));
        System.out.println("Max: " + opMax.apply(9, 6));

        // Using minBy()
        BinaryOperator<Integer> opMin =
BinaryOperator.minBy(comparator);
        System.out.println("Min: " + opMin.apply(5, 6));
        System.out.println("Min: " + opMin.apply(9, 6));
    }
}
```

Output:
```
Max: 6

Max: 9

Min: 5

Min: 6
```

Streams: By using streams we can perform various aggregate operations on the data returned from collections, arrays, Input/Output operations.All the classes and interfaces of stream API is in the java.util.stream package.

Stream represents a sequence of objects from a source, which supports aggregate operations. Following are the characteristics of a Stream.

→ **Sequence of elements** – A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.

→ **Source** – Stream takes Collections, Arrays, or I/O resources as input source.

→ **Aggregate operations** – Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.

→ **Pipelining** – Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. collect() method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.

→ **Automatic iterations** − Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

Creating the streams:

```
package streams;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Stream;

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

//by using stream.of() method
System.out.println("By using stream.of() method :");
Stream<Integer> stream = Stream.of(1, 2, 3, 4, 5, 6, 7, 8,
9);
stream.forEach(p -> System.out.println(p));

// by using stream.of(array) method
System.out.println("By using stream.of(array) method :");
Stream<Integer> stream1 = Stream.of( new Integer[]{6, 7,
8, 9,10} );
stream1.forEach(p -> System.out.println(p));

//by using List.stream() method
System.out.println("By using List.stream() method :");
List<Integer> list = new ArrayList<Integer>();
for(int i = 11; i< 16; i++){
list.add(i);
}
Stream<Integer> stream2 = list.stream();
stream2.forEach(p -> System.out.println(p));

//by using Stream.generate() method
System.out.println("By using Stream.generate() method :");
Stream<Integer> randomNumbers = Stream
.generate(() -> (new Random()).nextInt(100));
//limit() function to restrict 5 numbers
randomNumbers.limit(5)
.forEach(System.out::println);

}
}
```

Output:
```
By using stream.of() method :

1

2

3

4

5

6

7

8

9

By using stream.of(array) method :

6

7

8

9

10

By using List.stream() method :

11

12

13

14

15
```

```
By using Stream.generate() method :
52
32
16
79
94
```

Stream Operations: In java.util.Stream interface represents a stream on which one or more operations can be performed. **Stream operations are either intermediate or terminal**.
Intermediate Operations: Intermediate operations return the stream itself so you can chain multiple methods calls in a row.

Stream filter() Method: Stream.filter(Predicate) method to traverse all the elements and filter all elements which match a given condition through Predicate argument. Predicate is a functional interface and represents the condition to filter out the non-matching elements from the stream.

- → filter() is a intermediate Stream operation.
- → It returns a Stream consisting of the elements of the given stream that match the given predicate.
- → The filter() argument should be stateless predicate which is applied to each element in the stream to determine if it should be included or not.
- → Predicate is a functional interface. So, we can also pass lambda expression also.
- → It returns a new Stream so we can use other operations applicable to any stream.

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;
import java.util.stream.Collectors;

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

List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9, 10);

//Filtering a Stream using Lambda Expression
System.out.println("By using Lambda Expression");
list.stream()
.filter(n -> n % 2 == 0)
.forEach(System.out::println);

//Filtering a Stream using Predicate
System.out.println("By using Predicate");
Predicate<Integer> condition = new Predicate<Integer>() {
@Override
public boolean test(Integer n) {
if (n % 2 == 0) {
return true;
}
return false;
}
};
list.stream().filter(condition).forEach(System.out::printl
n);

//Filtering a Stream and Collecting into a List
System.out.println("By using collect");
List<Integer> evenNumbers = list.stream()
.filter(n -> n % 2 == 0)
.collect(Collectors.toList());

System.out.println(evenNumbers);

//Stream filter() and map() Example
System.out.println("By using filter() and map()");
List<Integer> evenNumbers1 = list.stream()
.filter(n -> n % 2 == 0)
.map(n -> n * n)
.collect(Collectors.toList());

System.out.println(evenNumbers1);
}
}
```

Output:

```
By using Lambda Expression
2
```

```
4
6
8
10
By using Predicate
2
4
6
8
10
By using collect
[2, 4, 6, 8, 10]
By using filter() and map()
[4, 16, 36, 64, 100]
```

Stream map() method: Stream.map() converts Stream<X> to Stream<Y>. For each object of type X, a new object of type Y is created and put in the new Stream.

Stream map() method has following syntax.

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

→ R represents the element type of the new stream.
→ mapper is a non-interfering, stateless function to apply to each element which produces a stream of new values.
→ The method returns a new stream of objects of type R.

Stream interface has three more similar methods which produce IntStream, LongStream and DoubleStream respectively after the map operation.

→ The map() is an intermediate operation. It returns a new Stream as return value.
→ The map() operation takes a Function, which is called for each value in the input stream and produces one result value, which is sent to the output stream.

→ The mapper function used for transformation is a stateless function (does not store the information of previously processed objects) and returns only a single value.
→ The map() method is used when we want to convert a Stream of X to Stream of Y.
→ The mapped stream is closed after its contents have been placed into the new output stream.
→ map() operation does not flatten the stream as flatMap() operation does.

converting a Stream of Strings to a Stream of Integers:

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamMap1 {
public static void main(String[] args) {
List<String> listOfStrings = Arrays.asList("1", "2", "3",
"4", "5");

List<Integer> listOfIntegers = listOfStrings.stream()
.map(Integer::valueOf)
.collect(Collectors.toList());

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

Output: [1, 2, 3, 4, 5]

finding all distinct salaries among employees:

```
package streams;

public class Employee {
private int empId;
private String empName;
private double salary;

public Employee(int empId, String empName, double salary)
{
this.empId = empId;
this.empName = empName;
this.salary = salary;
```

```
}

public int getEmpId() {
return empId;
}

public void setEmpId(int empId) {
this.empId = empId;
}

public String getEmpName() {
return empName;
}

public void setEmpName(String empName) {
this.empName = empName;
}

public double getSalary() {
return salary;
}

public void setSalary(int salary) {
this.salary = salary;
}
}
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamMap2 {
public static void main(String[] args) {
List<Employee> employeesList = Arrays.asList(
new Employee(1, "Name1", 100),
new Employee(2, "Name2", 100),
new Employee(3, "Name3", 200),
new Employee(4, "Name4", 200),
new Employee(5, "Name5", 300),
new Employee(6, "Name6", 300)
);

List<Double> distinctSalaries = employeesList.stream()
```

```
.map(e -> e.getSalary())
.distinct()
.collect(Collectors.toList());

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

Output: [100.0, 200.0, 300.0]

Stream flatMap() method: Stream flatMap() method is used to flatten a Stream of collections to a stream of objects. The objects are combined from all the collections in the original Stream. The flatMap() operation has the effect of applying a one-to-many transformation to the elements of the stream and then flattening the resulting elements into a new stream. In very layman terms, flattening is referred to as merging multiple collections/arrays into one.

→ flatMap() is an intermediate operation and return another stream as method output return value.

→ It returns a stream consisting of the results of replacing each element of the given stream with the contents of a mapped stream produced by applying the provided mapping function to each element.

→ The mapper function used for transformation in flatMap() is a stateless function and returns only a stream of new values.

→ Each mapped stream is closed after its contents have been placed into new Stream.

→ flatMap() operation flattens the stream; opposite to map() operation which does not apply flattening.

Converting nested lists into List:

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamFlatMap {
public static void main(String[] args) {
List<Integer> list1 = Arrays.asList(1, 2, 3);
List<Integer> list2 = Arrays.asList(4, 5, 6);
List<Integer> list3 = Arrays.asList(7, 8, 9);

List<List<Integer>> listOfLists = Arrays.asList(list1,
list2, list3);

List<Integer> listOfAllIntegers = listOfLists.stream()
.flatMap(x -> x.stream())
.collect(Collectors.toList());

System.out.println(listOfAllIntegers);
}
}
Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Converting nested arrays into List:

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamFlatMap1
{
public static void main(String[] args)
{
String[][] dataArray = new String[][]{{"a", "b"}, {"c",
"d"}, {"e", "f"}, {"g", "h"}};

List<String> listOfAllChars = Arrays.stream(dataArray)
.flatMap(x -> Arrays.stream(x))
.collect(Collectors.toList());

System.out.println(listOfAllChars);
}
}
Output: [a, b, c, d, e, f, g, h]
```

Stream distinct() method: Stream.distinct() method is used for filtering or collecting all the distinct elements from a stream. The distict() method is one of the stateful intermediate operation which uses the state from previously seen elements when processing new elements.

This method returns a stream consisting of the distinct elements from the given stream. The checking the objects equality, equals() method is used. For ordered streams, the element appearing first in the encounters order is preserved. For unordered streams, no stability guarantees are made.

Finding all distinct objects by the field value:

```
package streams;

class Person {
private Integer id;
private String fname;
private String lname;

public Person(Integer id, String fname, String lname) {
super();
this.id = id;
this.fname = fname;
this.lname = lname;
}

public Integer getId() {
return id;
}

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

public String getFname() {
return fname;
}

public void setFname(String fname) {
this.fname = fname;
}

public String getLname() {
return lname;
}

public void setLname(String lname) {
this.lname = lname;
}

@Override
public String toString() {
return "\nPerson [id=" + id + ", fname=" + fname + ",
lname=" + lname + "]";
}
```

```
}
package streams;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;
import java.util.function.Predicate;
import java.util.stream.Collectors;

public class DistinctExp {
public static void main(String[] args) {
Person virat = new Person(1, "Virat", "Kohli");
Person brian = new Person(2, "Brian", "lara");
Person alex = new Person(3, "Alex", "cary");

//Add some random persons
Collection<Person> list = Arrays.asList(virat, brian,
alex, virat, brian, virat);

// Get distinct objects by key
List<Person> distinctElements = list.stream()
.filter(distinctByKey(p -> p.getId()))
.collect(Collectors.toList());

// Let's verify distinct elements
System.out.println(distinctElements);
}

//Utility function
public static <T> Predicate<T> distinctByKey(Function<?
super T, Object> keyExtractor) {
Map<Object, Boolean> map = new ConcurrentHashMap<>();
return t -> map.putIfAbsent(keyExtractor.apply(t),
Boolean.TRUE) == null;
}
}
Output: [
```

```
Person [id=1, fname=Virat, lname=Kohli],

Person [id=2, fname=Brian, lname=lara],
```

Person [id=3, fname=Alex, lname=cary]]

Stream sorted() method: Stream sorted() method to sort the elements in a Stream by their natural order. We can also sort the elements using the provided Comparator.

Stream interface provides two methods for sorting the Stream elements.

→ **sorted()** – Provides the default sorting
→ **sorted(Comparator)** – Sorting based on provided comparator.

Stream sorted():

→ sorted() is a stateful intermediate operation which returns a new Stream.
→ It returns a stream consisting of the elements of this stream, sorted according to natural order.
→ If the elements of this stream are not Comparable, a java.lang.ClassCastException may be thrown when the terminal operation is executed.
→ For ordered streams, the sort is stable.
→ For unordered streams, no stability guarantees are made

Stream sorted(Comparator):

→ This is a stateful intermediate operation which returns a new stream.
→ It returns a stream consisting of the elements of this stream, sorted according to the provided Comparator..
→ For ordered streams, the sort is stable.
→ For unordered streams, no stability guarantees are made.

Sorting the elements of the Stream:

```
package streams;

import java.util.Arrays;
import java.util.Comparator;
import java.util.List;
import java.util.stream.Collectors;

public class StreamSorting {
public static void main(String[] args) {
List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6,
8);

//by using sorted() method
System.out.println("Sorting by using sorted()");
List<Integer> sortedList = list.stream()
.sorted()
.collect(Collectors.toList());

System.out.println(sortedList);

//By using sorted(Comparator)
System.out.println("Sorting by using sorted(Comparator)");
List<Integer> sortedList1 = list.stream()
.sorted(Comparator.reverseOrder())
.collect(Collectors.toList());

System.out.println(sortedList1);

//Sorting by using custom comparator
System.out.println("Sorting by using custom comparator");
Comparator<Integer> reverseComparator = new
Comparator<Integer>() {
@Override
public int compare(Integer i1, Integer i2) {
return i2.compareTo(i1);
}
};

List<Integer> sortedList2 = list.stream()
.sorted(reverseComparator)
.collect(Collectors.toList());

System.out.println(sortedList2);

//by using Lambda expression
System.out.println("Sorting by using Lambda expression");
List<Integer> sortedList3 = list.stream()
.sorted((i1, i2) -> i2.compareTo(i1))
.collect(Collectors.toList());

System.out.println(sortedList3);
}
}
```

Output:
```
Sorting by using sorted()

[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
Sorting by using sorted(Comparator)

[9, 8, 7, 6, 5, 4, 3, 2, 1]

Sorting by using custom comparator

[9, 8, 7, 6, 5, 4, 3, 2, 1]

Sorting by using Lambda expression

[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Stream peek() method: peek() method returns a new Stream consisting of all the elements from the original Stream after applying a given Consumer action. The peek() method is an intermediate Stream operation. So process the Stream element through peek(), we must use a terminal operation.

→ Stream peek() method is an intermediate operation.
→ It returns a Stream consisting of the elements of current stream.
→ It additionally perform the provided action on each element as elements.
→ For parallel stream pipelines, the action may be called at whatever time and in whatever thread the element is made available by the upstream operation.
→ If the action modifies shared state, it is itself responsible for providing the required synchronization.
→ peek() exists mainly to support debugging, where we want to see the elements as they flow past a certain point in a pipeline.

```
package                                                    streams;

import                                          java.util.Arrays;
import                                            java.util.List;
import                               java.util.stream.Collectors;

public            class            StreamPeek                   {
public     static     void     main(String[]     args)          {
List<Integer>  list  =  Arrays.asList(1,  2,  3,  4,  5);

//using                        peek()                      method
List<Integer>        newList        =        list.stream()
.peek(System.out::println)
.collect(Collectors.toList());

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

Output: 1

2

3

4

5

[1, 2, 3, 4, 5]

Stream limit() method: Stream limit(maxSize) is used to retrieve a number of elements from the Stream while the count must not be greater than a certain limit. The limit() method returns a Stream consisting of the elements of the given stream, truncated to be no longer than maxSize in length.

Here maxSize the number of elements the stream should be limited to; and the method return value is a new Stream consisting of elements picked from the original stream.

Stream.limit() method is short-circuiting intermediate operation. An intermediate operation is short-circuiting if, when presented with infinite input, it may produce a finite stream as a result. Please note that a terminal operation is short-circuiting if, when presented with infinite input, it may terminate in finite time.

→ It returns a stream consisting of the maximum elements, no longer than given size in length, of current stream.

→ Generally, limit() is cheap operation but may sometimes be expensive if maxSize has a large value and stream is parallely processed.

→ Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with BaseStream.unordered() may result in significant speedups of limit() in parallel pipelines.

→ limit() returns the first n elements in the encounter order.

Getting first 10 even numbers from an infinite stream of even numbers:

```
package streams;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamLimit {
public static void main(String[] args) {
Stream<Integer> evenNumInfiniteStream = Stream.iterate(0,
n -> n + 2);

List<Integer> newList = evenNumInfiniteStream.limit(10)
.collect(Collectors.toList());
System.out.println(newList);

}
}
```

Output: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

Stream skip() method: Stream skip(n) method is used to skip the first 'n' elements from the given Stream. The skip() method returns a new Stream consisting of the remaining elements of the original stream, after the specified n elements have been skipped in the encounter order.

→ Stream skip() method is stateful intermediate operation. Stateful operations, such as distinct and sorted, may incorporate state from previously seen elements when processing new elements.

→ Returns a stream consisting of the remaining elements of the stream after discarding the first n elements of the stream.

→ If the stream contains fewer than n elements then an empty stream will be returned.

→ Generally skip() is a cheap operation, it can be quite expensive on ordered parallel pipelines, especially for large values of n.

→ Using an unordered stream source (such as generate(Supplier)) or removing the ordering constraint with BaseStream.unordered() may result in significant speedups of skip() in parallel pipelines.

→ skip() skips the first n elements in the encounter order.

```
package                                              streams;

import                                        java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class StreamSkip {
public static void main(String[] args)
{
Stream<Integer> evenNumInfiniteStream = Stream.iterate(0, n
-> n + 2);

List<Integer> newList = evenNumInfiniteStream
.skip(5)
.limit(10)
.collect(Collectors.toList());
System.out.println(newList);
}
}
```

Output: [10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

Terminal Operations: Terminal operations return a result of a certain type after processing all the stream elements. Once the terminal operation is invoked on a Stream, the iteration of the Stream and any of the chained streams will get started. Once the iteration is done, the result of the terminal operation is returned.

Stream forEach() method: Stream forEach(action) method is used to iterate over all the elements of the given Stream and to perform an Consumer action on the each element of the Stream.

→ The forEach() method is a terminal operation. It means that it does not return an output of type Stream.

→ After forEach() is performed, the stream pipeline is considered consumed, and Stream can no longer be used.

→ If we need to traverse the same data source again (the collection backing the Stream), we must return to the data source to get a new stream.

→ For *parallel streams*, the forEach() operation does not guarantee the order of elements in the stream, as doing so would sacrifice the benefit of parallelism.

→ If the provided Consumer action accesses the shared state between the Stream elements the action is responsible for providing the required synchronization.

```
package streams;
```

```
import                                              java.util.Arrays;
import                                          java.util.Comparator;
import                                                java.util.List;

public           class           StreamForEach           {
public    static    void    main(String[]    args)    {
List<Integer>  list  =  Arrays.asList(2,  4,  6,  8,  10);

//by            using            simple            forEach
System.out.println("By          using          ForEach");
list.stream()
.forEach(System.out::println);

//by                  using                  comparator
System.out.println("By     using     comparator     ");
list.stream()
.sorted(Comparator.reverseOrder())
.forEach(System.out::println);
}
}
```

Output:
```
By using ForEach
2
4
6
8
10
By using comparator
10
8
6
4
2
```

Stream forEachOrdered() method: Stream forEachOrdered(action) method is used to iterate over all the elements of the given Stream and to perform an

Consumer action on the each element of the Stream, in the encounter order of the Stream if the Stream has a defined encounter order.

→ The forEachOrdered() method is a terminal operation. It means that it does not return an output of type Stream.
→ After forEachOrdered() is performed, the stream pipeline is considered consumed, and can no longer be used.
→ If we need to traverse the same data source again, we must return to the data source to get a new stream.
→ Performs an action for each element of this stream, in the encounter order of the stream if the stream has a defined encounter order.
→ Performing the action for one element happens-before performing the action for subsequent elements. But for any given element, the action may be performed in whatever Thread the library chooses.

```
package streams;

import java.util.Arrays;
import java.util.List;

public class StreamForEachOrdered {
public static void main(String[] args) {
List<Integer> list = Arrays.asList(2, 4, 6, 8, 10);

//using for forEach
System.out.println("Using forEach ");
list.stream().parallel()
.forEach(System.out::println);

//using forEachOrdered
System.out.println("Using forEachOrdered");
list.stream().parallel()
.forEachOrdered(System.out::println);
}
}
```

Output: Using forEach

6

10

8

```
2

4

Using forEachOrdered

2

4

6

8

10
```

Stream toArray() method: The toArray() method returns an array containing the elements of the given stream. This is a terminal operation. toArray() method is an overloaded method. The second method uses a generator function to allocate the returned array. The generator function takes an integer, which is the size of the desired array and produces an array of the desired size.

Using Stream filter and collecting employees based on salary to an Array:

```
package streams;

public class Employee {
private int empId;
private String empName;
private double salary;

public Employee(int empId, String empName, double salary)
{
this.empId = empId;
this.empName = empName;
this.salary = salary;
}

public int getEmpId() {
return empId;
}

public void setEmpId(int empId) {
this.empId = empId;
}

public String getEmpName() {
```

```
return empName;
}

public void setEmpName(String empName) {
this.empName = empName;
}

public double getSalary() {
return salary;
}

public void setSalary(int salary) {
this.salary = salary;
}

public String toString() {
return "\nEmployee{" +
"empId=" + empId +
", empName='" + empName + '\'' +
", salary=" + salary +
'}';
}
}
package streams;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class StreamToArray {
public static void main(String[] args) {
List<Employee> employeeList = new
ArrayList<>(Arrays.asList(
new Employee(1, "AB", 100),
new Employee(2, "BC", 200),
new Employee(3, "CD", 300),
new Employee(4, "DE", 400),
new Employee(5, "EF", 500),
new Employee(6, "FG", 600)));

Employee[] employeesArray = employeeList.stream()
.filter(e -> e.getSalary() < 400)
.toArray(Employee[]::new);
```

```
System.out.println(Arrays.toString(employeesArray));
}
}
```

Output: [

Employee{empId=1, empName='AB', salary=100.0},

Employee{empId=2, empName='BC', salary=200.0},

Employee{empId=3, empName='CD', salary=300.0}]

Stream count() method: Stream count() method returns the count of elements in the stream. To count the number of elements in stream, we can use Collectors.counting() method as well.

```
package streams;

import java.util.stream.Collectors;
import java.util.stream.Stream;

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

long count = Stream.of("Stream", "Count", "method",
"java")
.collect(Collectors.counting());
System.out.printf("There are %d words in the stream %n",
count);

count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
.collect(Collectors.counting());
System.out.printf("There are %d integers in the stream
%n", count);

count = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
.filter(i -> i % 2 == 0)
.collect(Collectors.counting());
System.out.printf("There are %d even numbers in the stream
%n", count);
}
}
```

Output: There are 4 words in the stream

There are 9 integers in the stream

There are 4 even numbers in the stream

Stream min() and max() methods: The **Stream min()** method is used to select the minimum/smallest element in the Stream according to the Comparator used for comparing the elements.The Comparator imposes a total ordering on the Stream elements which may not have a natural ordering.

Stream max() method is used to select the largest element in the Stream according to the Comparator used for comparing the elements. The Comparator imposes a total ordering on the Stream elements which may not have a natural ordering.

```
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class StreamMinAndMax {
public static void main(String[] args) {
List<Integer> list = Arrays.asList(2, 4, 1, 3, 7, 5, 9, 6,
8);

//using min() method
Optional<Integer> minNumber = list.stream()
.min((i, j) -> i.compareTo(j));

System.out.println("Min number : " + minNumber.get());

//using max() method
Optional<Integer> maxNumber = list.stream()
.max((i, j) -> i.compareTo(j));

System.out.println("Max Number : " + maxNumber.get());
}
}
```

Output: Min number : 1

Max Number : 9

Stream.match() methods:
Stream anyMatch(): Stream anyMatch(predicate) is terminal short-circuit operation. It is used to check if the stream contains at least one element whic satisfies the given predicate.

The anyMatch() method returns true if at least one element satisfies the condition provided by predicate, else false.

Stream allMatch(): Java Stream allMatch (predicate) is a short-circuiting terminal operation which is used to check if all the elements in the stream satisfy the provided predicate. Here predicate a non-interfering, stateless predicate to apply to all the elements of the stream. The allMatch() method returns always a true or false, based on the result of evaluation.

Stream noneMatch(): Java Stream noneMatch(predicate) method is *short-circuiting terminal operation*. It is used to check if no element of the Stream match the provided Predicate.

The noneMatch() returns:

→ **true** – if no element in the stream matches the given predicate, or the stream is empty.
→ **false** – if at least one element matches the given predicate.

Here predicate a non-interfering, stateless predicate to apply to elements of the stream.

```
package streams;

import java.util.function.Predicate;
import java.util.stream.Stream;

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

Stream<String> stream = Stream.of("one", "two", "three",
"four");

//using any match method
boolean match = stream.anyMatch(s -> s.contains("four"));
System.out.println(match);

Stream<String> stream1 = Stream.of("one", "two", "three",
"four");
//using allMatch method
Predicate<String> containsDigit = s -> s.contains("\\d+")
== false;
boolean match1 = stream1.allMatch(containsDigit);
System.out.println(match1);
Stream<String> stream2 = Stream.of("one", "two", "three",
"four");
//using noneMatch method
boolean match2 = stream2.noneMatch(s ->
s.contains("\\d+"));
System.out.println(match2);
}
}
```

Output: true

true

True

Collect() Method: the collect() method on a Stream, with a Collector instance passed as a parameter ends that Stream's processing and returns back the final result. Stream.collect() method is thus a terminal operation. In other words, Stream.collect() method is used to receive the elements from a stream and store them in a collection.

The collect() operation accumulates the elements in a stream into a container such as a collection. It performs mutable reduction operation in which the reduced (final) value is a mutable result container such as an ArrayList. This method takes a Collector implementation that provides useful reduction operations.

Finding the Employees Whose Salaries Are Above 10000.

```
package streams;

public class Employee {
private int empId;
private String empName;
private double salary;

public Employee(int empId, String empName, double salary) {
this.empId = empId;
this.empName = empName;
this.salary = salary;
}

public int getEmpId() {
return empId;
}
public void setEmpId(int empId) {
this.empId = empId;
}

public String getEmpName() {
return empName;
}

public void setEmpName(String empName) {
this.empName = empName;
}

public double getSalary() {
return salary;
}

public void setSalary(int salary) {
this.salary = salary;
}

@Override
public String toString() {
return "\nEmployee{" +
"empId=" + empId +
", empName='" + empName + '\'' +
", salary=" + salary +
'}';
}
}
```

```
package streams;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class StreamCollect {

public static void main(String[] args) {

List<Employee> empList = new ArrayList<Employee>();
empList.add(new Employee(01, "Employee1", 7000.00));
empList.add(new Employee(02, "Employee2", 10500.00));
empList.add(new   Employee(03,   "Employee3",   8000.00));
empList.add(new   Employee(04,   "Employee4",   12000.00));
empList.add(new   Employee(05,   "Employee2",   9000.00));

// Finding the Employees Whose Salaries Are Above 10000
List<Employee>     filteredList     =     empList.stream()
.filter(emp     ->     emp.getSalary()     >     10000)
.collect(Collectors.toList());
filteredList.forEach(System.out::println);
}
}
```

Output: Employee{empId=2, empName='Employee2', salary=10500.0}

Employee{empId=4, empName='Employee4', salary=12000.0}

Reduce() method: The reduce() method performs a reduction on the elements of the stream with the given function. The result is an Optional holding the reduced value. In the given example, we are reducing all the strings by concatenating them using a separator #.

```
package streams;

public class Employee {
private int empId;
private String empName;
private double salary;

public Employee(int empId, String empName, double salary)
{
this.empId = empId;
this.empName = empName;
this.salary = salary;
}

public int getEmpId() {
return empId;
}

public void setEmpId(int empId) {
this.empId = empId;
}

public String getEmpName() {
return empName;
```

```java
}

public void setEmpName(String empName) {
this.empName = empName;
}

public double getSalary() {
return salary;
}

public void setSalary(int salary) {
this.salary = salary;
}

@Override
public String toString() {
return "\nEmployee{" +
"empId=" + empId +
", empName='" + empName + '\'' +
", salary=" + salary +
'}';
}
}
package streams;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class StreamReduce {

static List<Employee> employeeList = Arrays.asList(
new Employee(01, "Name1", 7000.00),
new Employee(02, "Name2", 10000.00),
new Employee(03, "Name3", 8000.00),
new Employee(04, "Name4", 12000.00),
new Employee(05, "Name5", 9000.00));

public static void main(String[] args) {

//Total Salary Expenses
Double totalSalaryExpense = (Double) employeeList.stream()
.map(emp -> emp.getSalary())
.reduce(0.00,(a,b) -> a+b);
```

```
System.out.println("Total Salary Expense?= " +
totalSalaryExpense + "\n");

//Employee Details Having Maximum Salary
Optional<Employee> maxSalaryEmp = employeeList.stream()
.reduce((Employee a, Employee b) -> a.getSalary() <
b.getSalary() ? b:a);
if(maxSalaryEmp.isPresent()) {
System.out.println("Employee with Max. Salary?= "+
maxSalaryEmp.get());
}
}
}
```

Output: Total Salary Expense?= 46000.0

Employee with Max. Salary?=

Employee{empId=4, empName='Name4', salary=12000.0}

Stream findFirst() and findAny() methods:

findFirst() and **findAny()**. Both method looks very much similar but they may behave differently in certain conditions.

Stream findFirst() method: This method returns an Optional describing the first element of this stream. In case of stream has :

→ defined encounter order – first element in encounter order in stream.
→ no encounter order – any element may be returned.

Stream findAny() method: This method returns an Optional describing the any element of this stream. In case of stream has :

→ defined encounter order – any element may be returned.
→ no encounter order – any element may be returned.

```
package streams;

import java.util.stream.Stream;

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

//Sequential Stream
//findFirst() method
Stream.of("one", "two", "three", "four")
.findFirst()
.ifPresent(System.out::println);

//findAny() method
Stream.of("one", "two", "three", "four")
.findAny()
.ifPresent(System.out::println);

//parallel stream
//findFirst() method
Stream.of("one", "two", "three", "four")
.parallel()
.findFirst()
.ifPresent(System.out::println);

//findAny() method
Stream.of("one", "two", "three", "four")
.parallel()
.findAny()
.ifPresent(System.out::println);
}
}
```

Output:
```
one
one
one
three
```

mergeing streams of numbers:

```
package                                            streams;

import                          java.util.stream.Collectors;
import                              java.util.stream.Stream;

public           class           StreamsMerge             {
public     static     void     main(String[]     args)    {
Stream<Integer> firstStream = Stream.of(1, 2, 3, 4, 5, 6);
Stream<Integer> secondStream = Stream.of(4, 5, 6, 7, 8, 9);

Stream<Integer>              resultingStream              =
Stream.concat(firstStream,                    secondStream)
.distinct();

System.out.println(resultingStream.collect(Collectors.toLi
st()));
}
}
```

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

Method References: Method references are a special form of the lambda expression. Since the lambda expressions are doing nothing other than invoking existing behavior (methods), we can achieve the same result by referring to it by name.

- → :: is used to refer to a method.
- → Method type arguments are inferred by JRE at runtime from the context it is defined.

Types of Method References: There are four types of method references.

1. Static method reference
2. Instance method reference of a particular object
3. Instance method reference of an arbitrary object of a particular type
4. Constructor reference

Static Method Reference: When we refer to the static method of Containing a class.

e.g. ClassName::staticMethodName

Instance Method Reference of a Particular Object: When you refer to the instance method of the particular object, you will use e.g. containingObjectReference::someInstanceMethodName

Instance Method Reference of an Arbitrary Object: When you refer to the instance method of a class with the ClassName , you will get the instance method reference of an arbitrary object of a particular type, such as

e.g. ClassName::someInstanceMethod;

Constructor Reference: When you refer to a constructor of a class in lambda, you will get a constructor reference, such as

e.g. ClassName::new.

Lambda Expressions	Equivalent Method References
(String s) -> Integer.parseInt(s)	Integer::parseInt
(String s) -> s.toLowerCase()	String::toLowerCase
(int i) -> System.out.println(i)	System.out::println
(Student s) -> s.getName()	Student::getName
() -> s.getName()	s::getName where *'s'* refers to *Student* object which already exist.
() -> new Student()	Student::new

Example for all the four types of method references:

```
package methodreferences;

public class MathOperation {

//constructor operation
public MathOperation(int a, int b) {
System.out.println("Sum of " + a + " and " + b + " is " +
(a + b));
}

//is Even or odd
public static boolean isEven(int n) {
return n % 2 == 0;
}

// Addition
public int add(int a, int b) {
return a + b;
}

//Subtraction
```

```
public int sub(int a, int b) {
return a - b;
}

}
package methodreferences;

import java.util.ArrayList;

import java.util.Arrays;
import java.util.List;
import java.util.function.BiConsumer;
import java.util.function.BiFunction;

public class MethodRef1 {
public static void main(String[] args) {

System.out.println("Sum of numbers by normal operation :
");
MathOperation op = new MathOperation(10, 20);

//reference to static method
System.out.println("Refering a static Method : ");
List<Integer> numbers = Arrays.asList(20, 15);
numbers.stream().map(MathOperation::isEven)
        .forEach(System.out::println);

//reference to an an Instance Method of a Particular
Object
System.out.println("Reference to instance Method : ");
BiFunction<Integer, Integer, Integer> add2 = op::add;
System.out.println("Addtion = " + add2.apply(4, 5));

//reference to an an Instance Method of a Particular
//Object
BiFunction<Integer, Integer, Integer> sub2 = op::sub;
System.out.println("Subtraction = " + sub2.apply(58, 5));

//Reference to a Constructor
System.out.println("Refering a constructor : ");
BiConsumer<Integer, Integer> addtion2 =
MathOperation::new;
addtion2.accept(10, 30);
//Reference to an Instance Method of an Arbitrary Object
of //a Particular Type
List<String> weeks = new ArrayList<String>();
weeks.add("Monday");
weeks.add("Tuesday");
weeks.add("Wednesday");
weeks.add("Thursday");
weeks.add("Friday");
weeks.add("Saturday");
weeks.add("Sunday");

System.out.println("Refering a Arbitrary Object of a
Particular Type : ");
weeks.stream()
.map(String::toUpperCase)
.forEach(System.out::println);

}
}
```

Output: Sum of numbers by normal operation :

Sum of 10 and 20 is 30

Refering a static method :

The given num is even :

true

false

Reference to instance Method :

Addtion = 9

Subtraction = 53

Refering a constructor :

Sum of 10 and 30 is 40

Refering Arbitrary Object of a Particular Type :

MONDAY

TUESDAY

WEDNESDAY

THURSDAY

FRIDAY

SATURDAY

SUNDAY