JAVA

JVM (java virtual machine)

What is mean by platform independent language?

Access modifier in java

Non access modifier in java

Primitive data types

Non-primitive data types

How to find maximum and minimum value of primitive data types

Casting

Ternary Operator

toString()

Difference between abstract class and interface

Comparable interface

Comparator interface

( diff bet run time and compile time , method overloading, method overriding)

OOPs (Object-Oriented Programming)

Object

Class

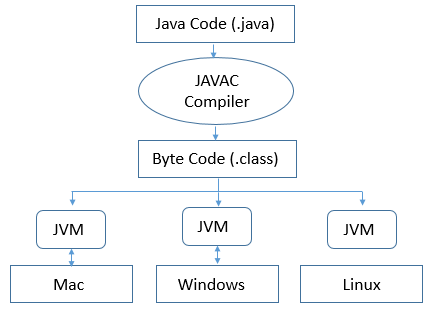
Inheritance

Stream()

Collections

JVM (java virtual machine)

* JVM acts as a run-time engine to run Java applications.
* JVM allow java programs to run on any operating system
* Due to JVM java is a platform independent language
* JVM is a part of JRE(java runtime environment)



What is mean by platform independent language?

The meaning of platform-independent is that **the java compiled code(byte code) can run on all operating systems**.

# How is Java platform independent?

# The meaning of platform-independent is that the java compiled code(byte code) can run on all operating systems.

# A program is written in a language(.java), machine does not understand this source code

# Hence The compiler converts the source code into the Byte code

* The Byte code needs an interpreter to execute on a machine. This interpreter is the JVM and thus the Bytecode is executed by the JVM.

Access modifier in java

Defines the scope of variable/method/class/constructor.

1. public
2. private
3. protected
4. default

public – accessible everywhere

private – accessible inside the class

protected -- accessible in the same package and **subclasses**

default -- accessible in the same package, if we don’t specify Access modifier it is default

Non access modifier in java

Provides the information to the JVM about the behavior of the variable/method/class/constructor.

1. static
2. final
3. abstract
4. synchronized
5. transient
6. volatile
7. native

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include string, [Classes](https://www.javatpoint.com/object-and-class-in-java), [Interfaces](https://www.javatpoint.com/interface-in-java), and [Arrays](https://www.javatpoint.com/array-in-java)/List/Set.

How to find maximum and minimum value of primitive data types

DataType.MIN\_VALUE; DataType.MAX\_VALUE;

e.g Integer.MIN\_VALUE; Integer.MAX\_VALUE;

Byte.MIN\_VALUE; Byte.MAX\_VALUE;

Overflow – value larger than the maximum value, it will give minimum value.

underflow – value smaller than the minimum value, it will give maximum value.

Casting

Type casting is when you assign a value of one primitive data type to another type.

double myDouble = 9.78d;

int myInt = (int) myDouble;

Ternary Operator

variable = (condition) ? expression1 : expression2

The above statement states that if the condition returns **true, expression1** gets executed, else the **expression2** gets executed and the final result stored in a variable.

1. x = 20;

y = (x == 1) ? 61: 90;

ouput: y=90

1. x = 20;

y = (x == 20) ? 61: 90;

ouput: y=61

toString()

* toString() method is used to return a string representation of an object.
* If any object is printed in java by default java compiler invokes the toString() method
* We have to @override toString() method, but no need to call this method before printing the object If any object is printed by default java compiler call the toString() method

@Override

**public** String toString() {

**return** “string representation of object”;

}

Difference between abstract class and interface

|  |  |
| --- | --- |
| Abstract class | Interface |
| 1) Abstract class can **have abstract and non- abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 3) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 4) abstract class contains constructor | Interface does not contain constructor |
| 5)  Abstract class and interface both can't be instantiated (can not create object). | Abstract class and interface both can't be instantiated (can not create object). |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface** can be implemented using keyword "implements". |
| 8) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator
3. Comparable

* The class itself must implements the **java.lang.Comparable** interface to compare its instances.

e.g **public** **class** Student **implements** Comparable<Student>{..}

* use compareTo() method in the class
* in main class use Collections.sort(list); to sort the objects of the class according to the attributes of the class

example : create a student class with attribute rollNo, marks, name and compare the objects of the class according to the attributes rollNo, marks, name.

code

**public** **class** Student **implements** Comparable<Student>{

**int** rollNo;

**int** marks;

String name;

**public** Student(**int** rollNo, **int** marks, String name) {

**this**.rollNo = rollNo;

**this**.marks = marks;

**this**.name = name;

}

@Override

**public** **int** compareTo(Student o) {

**return** **this**.marks - o.marks;

}

@Override

**public** String toString() {

**return** "Student [rollNo=" + rollNo + ", marks=" + marks + ", name=" + name + "]";

}

}

**public** **class** Main {

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

List<Student> list = **new** ArrayList<Student>();

list.add(**new** Student(10,50,"bbb"));

list.add(**new** Student(5,60,"aaa"));

list.add(**new** Student(11,30,"vvv"));

Collections.*sort*(list);

ListIterator<Student> i = list.listIterator();

**while**(i.hasNext()) {

System.***out***.println(i.next());

}

}

}

Output

Student [rollNo=11, marks=30, name=vvv]

Student [rollNo=10, marks=50, name=bbb]

Student [rollNo=5, marks=60, name=aaa]

1. Comparator

* Create a new class and implement java.util.Comparator interface in this class use compare() method.

**e.g**

**class** NameComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.name.compareTo(o2.name);

}

}

* In main class use Collections.*sort*(list, **new** NameComparator());

example : create a student class with attribute rollNo, marks, name and compare the objects of the class according to the attributes rollNo, marks, name.

Code

**public** **class** Student {

**int** rollNo;

**int** marks;

String name;

**public** Student(**int** rollNo, **int** marks, String name) {

**this**.rollNo = rollNo;

**this**.marks = marks;

**this**.name = name;

}

@Override

**public** String toString() {

**return** "Student [rollNo=" + rollNo + ", marks=" + marks + ", name=" + name + "]";

}

}

**public** **class** Main {

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

List<Student> list = **new** ArrayList<Student>();

list.add(**new** Student(10,50,"bbb"));

list.add(**new** Student(5,60,"aaa"));

list.add(**new** Student(11,30,"vvv"));

Collections.*sort*(list, **new** NameComparator());

list.forEach(System.***out***::println);

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

Collections.*sort*(list, **new** rollNoComparator());

list.forEach(System.***out***::println);

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

Collections.*sort*(list, **new** marksComparator());

list.forEach(System.***out***::println);

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

}

}

**class** NameComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.name.compareTo(o2.name);

}

}

**class** rollNoComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.rollNo - o2.rollNo;

}

}

**class** marksComparator **implements** Comparator<Student>{

@Override

**public** **int** compare(Student o1, Student o2) {

**return** o1.marks - o2.marks;

}

}

Output

Student [rollNo=5, marks=60, name=aaa]

Student [rollNo=10, marks=50, name=bbb]

Student [rollNo=11, marks=30, name=vvv]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Student [rollNo=5, marks=60, name=aaa]

Student [rollNo=10, marks=50, name=bbb]

Student [rollNo=11, marks=30, name=vvv]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Student [rollNo=11, marks=30, name=vvv]

Student [rollNo=10, marks=50, name=bbb]

Student [rollNo=5, marks=60, name=aaa]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

OOPs (Object-Oriented Programming)

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. Class
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

[Object](https://www.javatpoint.com/object-and-class-in-java)

Object is an entity e.g pen, car, bike, House

object is an instance of the class

ClassName objectName = new ClassName();

Class

A class is a template or blueprint from which objects are created.

class ClassName {

fields;

methods;

}

[Inheritance](https://www.javatpoint.com/inheritance-in-java)

**Inheritance** is a mechanism in which one class acquires the property of another class.

**class** Child **extends** Parent

{

//methods and fields

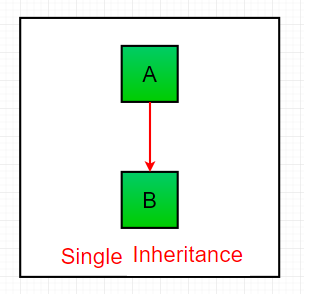
}

* Child class can reuse all the methods and fields of Parent class, also we can add new methods and fields in child class
* Inheritance represents the **IS-A relationship**. E.g car is a vehicle, dog is an animal
* extends keyword is used to inherit class

**Types of Inheritance in Java**

1. **Single Inheritance**

In single inheritance, subclasses inherit the features of one superclass.



//code

|  |
| --- |
| class one {  public void print\_geek()  {  System.out.println("Geeks");  }  }  class two extends one {  public void print\_for() { System.out.println("for"); }  }  public class Main {  public static void main(String[] args)  { two g = new two();  g.print\_geek();  g.print\_for();  g.print\_geek();  }  } |

**Output**

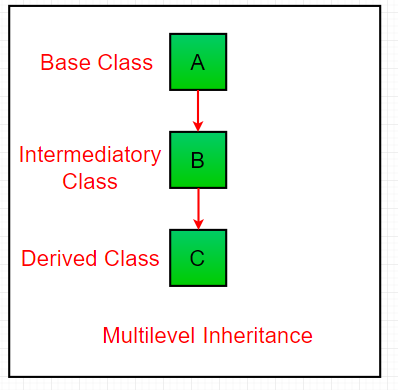
Geeks

For

Geeks

1. **Multilevel Inheritance**

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

  
code

class one {

public void print\_geek()

{

System.out.println("Geeks");

}

}

class two extends one {

public void print\_for() { System.out.println("for"); }

}

class three extends two {

public void print\_geek()

{

System.out.println("Geeks");

}

}

// Drived class

public class Main {

public static void main(String[] args)

{

three g = new three();

g.print\_geek();

g.print\_for();

g.print\_geek();

}

}

**Output**

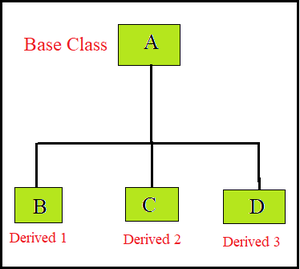
Geeks

for

Geeks

1. **Hierarchical Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.



Code

class A {

public void print\_A() { System.out.println("Class A"); }

}

class B extends A {

public void print\_B() { System.out.println("Class B"); }

}

class C extends A {

public void print\_C() { System.out.println("Class C"); }

}

class D extends A {

public void print\_D() { System.out.println("Class D"); }

}

// Driver Class

public class Test {

public static void main(String[] args)

{

B obj\_B = new B();

obj\_B.print\_A();

obj\_B.print\_B();

C obj\_C = new C();

obj\_C.print\_A();

obj\_C.print\_C();

D obj\_D = new D();

obj\_D.print\_A();

obj\_D.print\_D();

}

}

**Output**

Class A

Class B

Class A

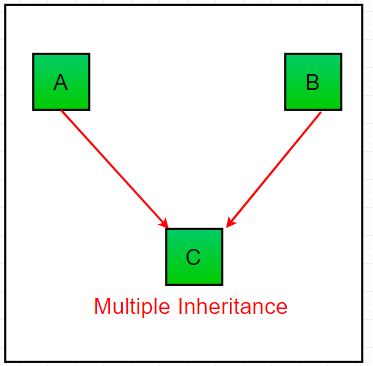
Class C

Class A

Class D

1. [**Multiple Inheritance**](https://www.geeksforgeeks.org/java-and-multiple-inheritance/)**(Through Interfaces)**

Note: Java does **not** support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).



code

interface one {

public void print\_geek();

}

interface two {

public void print\_for();

}

interface three extends one, two {

public void print\_geek();

}

class child implements three {

@Override

public void print\_geek()

{System.out.println("Geeks");}

@Override

public void print\_for()

{ System.out.println("for"); }

}

public class Main {

public static void main(String[] args)

{

child c = new child();

c.print\_geek();

c.print\_for();

c.print\_geek();

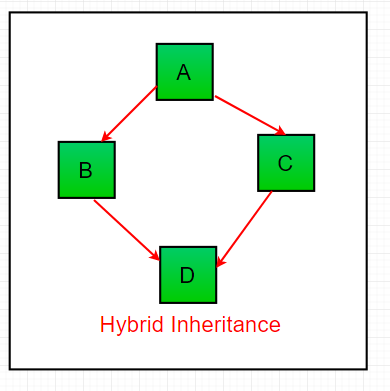
}

}

1. **Hybrid Inheritance(Through Interfaces):**

It is a mix of two or more of the above types of inheritance.

In java, we can achieve hybrid inheritance only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/).



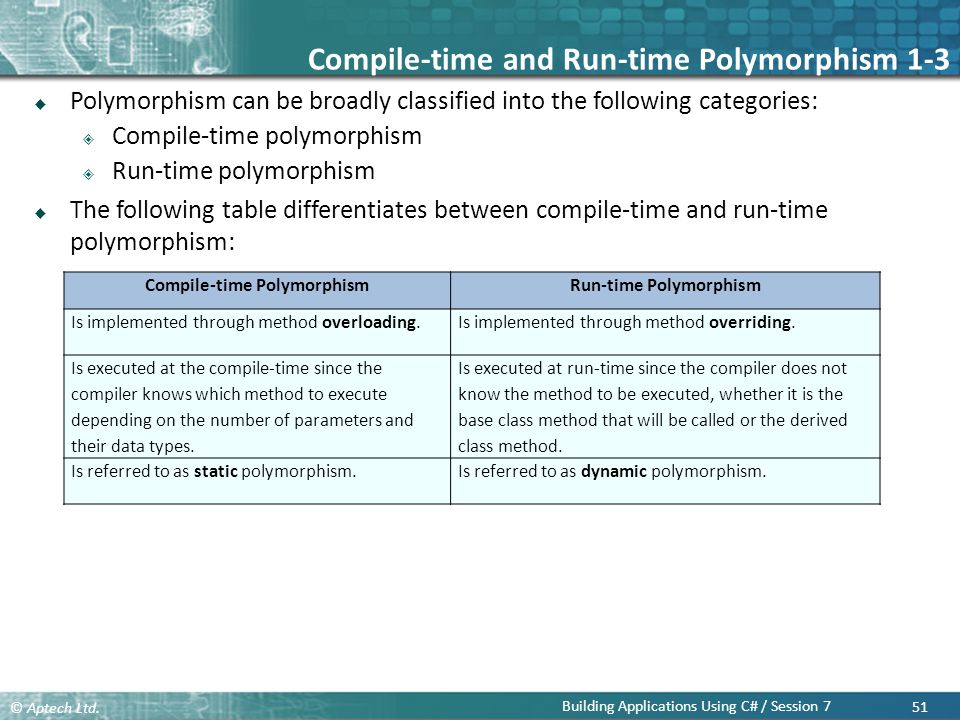
Polymorphism

* polymorphism means many forms.
* polymorphism in java allows us to perform the same action in many different ways.

**types of polymorphism:**

* Compile time Polymorphism
* Runtime Polymorphism

Diff between Compile time Polymorphism and Runtime Polymorphism



1. Compile time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But **Java doesn’t support the Operator Overloading**.

|  |
| --- |
| class MultiplyFun {        // Method with 2 parameter      static int Multiply(int a, int b)      {          return a \* b;      }        // Method with the same name but 2 double parameter      static double Multiply(double a, double b)      {          return a \* b;      }  }    class Main {      public static void main(String[] args)      {            System.out.println(MultiplyFun.Multiply(2, 4));            System.out.println(MultiplyFun.Multiply(5.5, 6.3));      }  } |

**Output:**

8

34.65

1. Runtime Polymorphism

* It is a process in which a function call to the overridden method is resolved at Runtime.
* This type of polymorphism is achieved by Method Overriding.

### **Upcasting**

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:

**class** A{}

**class** B **extends** A{}

A a=**new** B();//upcasting

code

|  |
| --- |
| class Parent {        void Print()      {          System.out.println("parent class");      }  }    class subclass1 extends Parent {        void Print()      {          System.out.println("subclass1");      }  }    class subclass2 extends Parent {        void Print()      {          System.out.println("subclass2");      }  }    class TestPolymorphism3 {      public static void main(String[] args)      {          Parent a;            a = new subclass1();          a.Print();            a = new subclass2();          a.Print();      }  } |

**Output:**

subclass1

subclass2

Stream()

the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

Different Operations On Streams-

1. **Intermediate Operations: they return stream e.g filter(), map(), sorted()**
2. **Terminal Operations: only one value they return e.g reduce(), collect(), forEach()**

**Intermediate Operations**

**map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

e.g square of each element of the stream, multiply/divide by some value to each element of the string

List<Integer> list = List.*of*(1,2,3,6,4,9);

List<Integer> list2 = list.stream().map(e -> e\*e).collect(Collectors.*toList*());

**filter:** The filter method is used to filter the elements of some particular type

e.g odd numbers, even numbers, strings start with some alphabets

List<Integer> list = List.*of*(1,2,3,6,4,9);

List<Integer> list3 = list.stream().filter(e -> e%2==0).collect(Collectors.*toList*());

**sorted:** The sorted method is used to sort the stream.

e.g

List<Integer> list = List.*of*(1,2,3,6,4,9);

List<Integer> list1 = list.stream().sorted().collect(Collectors.*toList*());

**Terminal Operations**

**collect:** The collect method is used to convert stream to the list/array/set/map format

List<Integer> list3 = list.stream().filter(e -> e%2==0).collect(Collectors.*toList*());

**forEach:** The forEach method is used to iterate through every element of the stream.

List<Integer> list3 = list.stream().filter(e -> e%2==0).forEach(System.out::println);

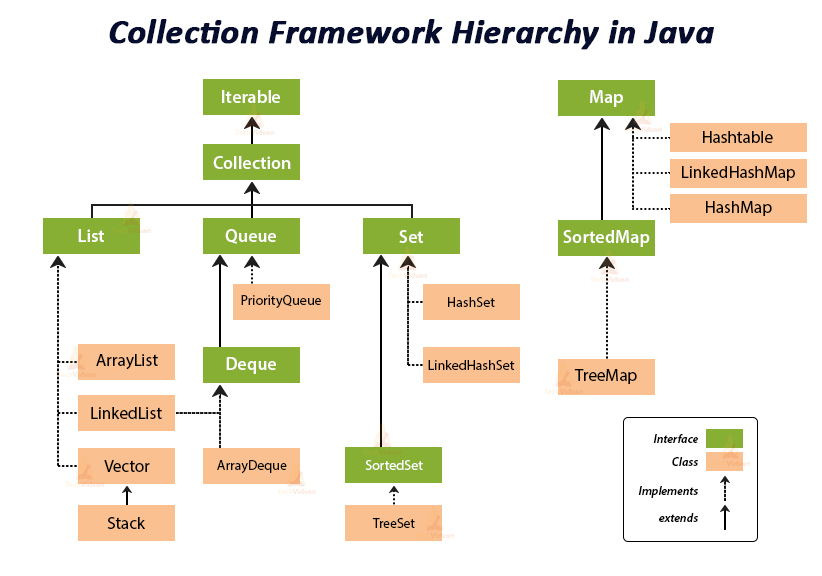
**reduce:** The reduce method is used to reduce the elements of a stream to a single value.

e.g sum of all numbers, large number,small number

List<Integer> list = List.*of*(1,2,3,6,4,9);

Integer number = list.stream().reduce(0,(e1,e2)-> e1 + e2);

Collections



## Methods of the Collection Interface

This interface contains various methods which can be directly used by all the collections which implement this interface(List/Set/Queue).

|  |  |
| --- | --- |
| Method | Description |
| [**add(Object)**](https://www.geeksforgeeks.org/collection-add-method-in-java-with-examples/) | This method is used to add an object to the collection. |
| [**addAll(Collection c)**](https://www.geeksforgeeks.org/collections-addall-method-in-java-with-examples/) | This method adds all the elements in the given collection to this collection. |
| [**clear()**](https://www.geeksforgeeks.org/collection-clear-method-in-java-with-examples/) | This method removes all of the elements from this collection. |
| [**contains(Object o)**](https://www.geeksforgeeks.org/collection-contains-method-in-java-with-examples/) | This method returns true if the collection contains the specified element. |
| **containsAll(Collection c)** | This method returns true if the collection contains all of the elements in the given collection. |
| **equals(Object o)** | This method compares the specified object with this collection for equality. |
| **hashCode()** | This method is used to return the hash code value for this collection. |
| [**isEmpty()**](https://www.geeksforgeeks.org/collection-isempty-method-in-java-with-examples/) | This method returns true if this collection contains no elements |
| **iterator()** | This method returns an iterator over the elements in this collection. |
| [**max()**](https://www.geeksforgeeks.org/collections-max-method-in-java-with-examples/) | This method is used to return the maximum value present in the collection. |
| **remove(Object o)** | This method is used to remove the given object from the collection. If there are duplicate values, then this method removes the first occurrence of the object |
| **removeAll(Collection c)** | This method is used to remove all the objects mentioned in the given collection from the collection. |
| **removeIf(Predicate filter)** | this method is used to remove all the elements of this collection that satisfy the given [predicate](https://www.geeksforgeeks.org/mathematic-logic-predicates-quantifiers/). |
| **size()** | This method is used to return the number of elements in the collection. |
| **stream()** | This method is used to return a sequential Stream with this collection as its source. |
| **toArray()** | This method is used to return an array containing all of the elements in this collection. |
| **retainAll(Collection c)** | This method is used to retain only the elements in this collection that are contained in the specified collection. |
| **spliterator()** | This method is used to create a [Spliterator](https://www.geeksforgeeks.org/java-program-to-convert-iterator-to-spliterator/) over the elements in this collection. |

**List Interface**

* It is an ordered collection of objects in which duplicate values can be stored.
* Since List preserves the insertion order, it allows positional access and insertion of elements.
* List<obj\_type> list = new ArrayList();
* List<obj\_type> list = List.of(obj1, obj2, obj3);
* **Classes that impliments list interface**

1. Vector
2. Stack
3. ArrayList
4. LinkedList

* **Methods list supports**

|  |  |
| --- | --- |
| **add(Object)** | This method is used to add an element at the end of the List. |
| **add(int index, Object)** | This method is used to add an element at a specific index in the List |
| **remove(Object)** | This method is used to simply remove an object from the List. If there are multiple such objects, then the first occurrence of the object is removed. |
| **remove(int index)** | Since a List is indexed, this method takes an integer value which simply removes the element present at that specific index in the List. After removing the element, all the elements are moved to the left to fill the space and the indices of the objects are updated. |
| [**get(int index)**](https://www.geeksforgeeks.org/list-get-method-in-java-with-examples/) | This method returns elements at the specified index. |
| [**set(int index, element)**](https://www.geeksforgeeks.org/arraylist-set-method-in-java-with-examples/) | This method replaces elements at a given index with the new element. This function returns the element which was just replaced by a new element. |

**Vector**

* A vector provides us with dynamic arrays in Java.
* This is identical to ArrayList in terms of implementation.
* However, the primary difference between a vector and an ArrayList is that a Vector is synchronized and an ArrayList is non-synchronized.

**Code**

**import** java.util.List;

**import** java.util.Vector;

**public** **class** VectorClass {

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

// 3 ways to take list

// 1st no data type

List vector1 = **new** Vector();

// 2nd mention data type

List<Integer> vector2 = **new** Vector();

List<Integer> vector = **new** Vector<Integer>();

//dont use List interface

Vector vector4 = **new** Vector();

// add elements in the list

vector1.add(00);

vector1.add(10);

vector1.add(20);

vector1.add(3, 30);

vector1.add(40);

vector1.add(5,50);

vector1.add(2, 88); // adding at thet place and shifting the previous elements

System.***out***.println(vector1);

// remote element by index

vector1.remove(4);

// list1.remove(50); //getting error

System.***out***.println(vector1);

// set elements by index

vector1.set(3, 99);

System.***out***.println(vector1);

}

}

**Output**

[0, 10, 88, 20, 30, 40, 50]

[0, 10, 88, 20, 40, 50]

[0, 10, 88, 99, 40, 50]

**Stack**

* The class is based on the basic principle of last-in-first-out.
* extends the vector class

**code**

**import** java.util.Iterator;

**import** java.util.List;

**import** java.util.Stack;

**public** **class** StackClass {

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

// 3 ways to take list

// 1st no data type

Stack stack1 = **new** Stack();

// 2nd mention data type

Stack<Integer> stack2 = **new** Stack();

Stack<Integer> stack3 = **new** Stack<Integer>();

// push

stack2.push(10);

stack2.push(20);

stack2.push(30);

stack2.push(40);

stack2.push(50);

// search method

System.***out***.println(stack2.search(20)); // calculate from last as last element is at the top of the stack

//empty method

System.***out***.println(stack2.empty());

//pop

stack2.pop();

System.***out***.println(stack2);

// remote element by index

stack2.remove(2);

// list1.remove(50); //getting error

System.***out***.println(stack2);

Iterator i1 = stack2.iterator();

**while**(i1.hasNext()) {

System.***out***.println(i1.next());

}

Output

4

false

[10, 20, 30, 40]

[10, 20, 40]

10

20

40

**ArrayList**

* [**ArrayList**](https://www.geeksforgeeks.org/arraylist-in-java/)class which is implemented in the collection framework provides us with dynamic arrays in Java.

**Code**

import java.util.ArrayList;

import java.util.List;

public class ArrayListClass {

public static void main(String[] args) {

// 3 ways to take list

// 1st no data type

ArrayList list1 = new ArrayList();

// 2nd mention data type

List<Integer> list2 = new ArrayList();

List<Integer> list3 = new ArrayList<Integer>();

//dont use List interface

List list4 = new ArrayList();

// add elements in the list

list1.add(00);

list1.add(10);

list1.add(20);

list1.add(3, 30);

list1.add(40);

list1.add(5,50);

list1.add(2, 88); // adding at thet place and shifting the previous elements

System.out.println(list1);

// remote element by index

list1.remove(4);

// list1.remove(50); //getting error

System.out.println(list1);

// set elements by index

list1.set(3, 99);

System.out.println(list1);

}

}

Output

[0, 10, 88, 20, 30, 40, 50]

[0, 10, 88, 20, 40, 50]

[0, 10, 88, 99, 40, 50]

[Vectors](https://www.geeksforgeeks.org/java-util-vector-class-java/) and  [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/)  both implement the List interface and both use **dynamically resizable arrays**.

Difference between vector and Array List

|  |  |
| --- | --- |
| **Vector** | **ArrayList** |
| Vector<T> v = new Vector<T>(); | ArrayList<T> al = new ArrayList<T>(); |
| Synchronized, thread safe, not used in multithreading | Unsynchronized, not thread safe, can be used in multithreading |
| Slower | Faster |
| Uses Iterator interface and enumeration to traverse through the elements | Uses Iterator interface to traverse through the elements |

LinkedList

* every element is a separate object with a data part and address part.
* The elements are linked using pointers and addresses. Each element is known as a node.
* Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.
* It also has a few disadvantages like the nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

Code

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.ArrayList;

**public** **class** LinkedListClass {

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

LinkedList<Integer> linkedList1 = **new** LinkedList();

linkedList1.add(00);

linkedList1.add(1, 10);

linkedList1.add(20);

//addAll(int index, Collection<E> c)

List<Integer> arrayList1 = **new** ArrayList();

arrayList1.add(30);

arrayList1.add(40);

linkedList1.addAll(3, arrayList1);

//addAll(Collection<E> c)

linkedList1.addAll(List.*of*(50,60,70));

linkedList1.addAll(**new** ArrayList(List.*of*(80,90)));

//addFirst(E e)

linkedList1.addFirst(0000);

//addLast(E e)

linkedList1.addLast(9999);

System.***out***.println(linkedList1);

System.***out***.println(linkedList1.get(7));

System.***out***.println(linkedList1.getFirst());

System.***out***.println(linkedList1.getLast());

//remove(Object o)

//linkedList1.remove(60); // give error

//remove(int index)

linkedList1.remove(6);

System.***out***.println(linkedList1);

}}

Output

[0, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 9999]

60

0

9999

[0, 0, 10, 20, 30, 40, 60, 70, 80, 90, 9999]

**What is the Need for a Separate Collection Framework in java?**

* Before the Collection Framework was introduced, the standard methods for grouping Java objects were [**Arrays**](https://www.geeksforgeeks.org/introduction-to-arrays/) or [**Vectors**](https://www.geeksforgeeks.org/java-util-vector-class-java/), or **[Hashtables](https://www.geeksforgeeks.org/hashtable-in-java/)**.
* All of these collections had no common interface. Therefore, though the main aim of all the collections is the same, the implementation of all these collections was defined independently and had no correlation among them.
* Therefore, it is very difficult for the users to remember all the different [**methods**](https://www.geeksforgeeks.org/methods-in-java/), syntax, and [**constructors**](https://www.geeksforgeeks.org/constructors-in-java/) present in every collection class

Let’s understand this with an example

// Java program to demonstrate

// why collection framework was needed

import java.io.\*;

import java.util.\*;

class CollectionDemo {

public static void main(String[] args)

{

// Creating instances of the array,

// vector and hashtable

int arr[] = new int[] { 1, 2, 3, 4 };

Vector<Integer> v = new Vector();

Hashtable<Integer, String> h = new Hashtable();

// Adding the elements into the

// vector

v.addElement(1);

v.addElement(2);

// Adding the element into the

// hashtable

h.put(1, "geeks");

h.put(2, "4geeks");

// Array instance creation requires [],

// while Vector and hastable require ()

// Vector element insertion requires addElement(),

// but hashtable element insertion requires put()

// Accessing the first element of the

// array, vector and hashtable

System.out.println(arr[0]);

System.out.println(v.elementAt(0));

System.out.println(h.get(1));

// Array elements are accessed using [],

// vector elements using elementAt()

// and hashtable elements using get()

}

}

**Output:**

1

1

geeks

**Set interface**

* [interface](https://www.geeksforgeeks.org/collections-in-java-2/) is an unordered collection of objects in which duplicate values cannot be stored.
* It is an interface that implements the mathematical set.(supports union, intersection,difference)
* Set<Obj\_type> set = new HashSet<Obj\_type> ();
* Set<Obj\_type > set = Set.of(obj1, obj2, obj3);

**Code**

**Question:**

**Set A = (1,2,3,4,5,6);**

**Set B= (4,5,6,7,8,9);**

**A union B = (1,2,3,4,5,6,7,8,9)**

**A intersection B = (4,5,6)**

**What is static keyword in java?**

* **The static keyword in java is used to share the same variable or method of a given class**
* **The static keyword belongs to the class than an instance of the class**
* For example there are thousands of employees working in a company

So the same company name is shared between thousands of employee

So we can declare company name as a static variable so the every can use it

* Basically static keyword is used for memory management
* Every object of the class shares the same static variable and method
* If any change made to the static variable, all the instances will see the effect of the change
* Because when you change the value of the static variable for one instance the actual value of the variable will gets change

**Can we override static method?**

* We can not override static method
* If you declare the same method in child class it will hide the parent class method, this approach is useful in method hiding

**What is final keyword in java?**

* Final keyword in java is used to make a variable or method or class unchangeable
* You can assign value to the final variable only at once, e.g PI value is 3.14159
* We can not override a final method
* We can not inherit the final the class

**String immutability concept?**

* String is immutable class in java
* Immutable means unchangeable
* When u create a object of string after the we can not change the value of the object
* But You can change the reference to the object
* E.g if there are 10 persons staying in pune city

And if person1 shifted to Mumbai city

Then in java jvm will create new string object in heap area and assign Mumbai value to it

And change the reference of the person1 to the Mumbai

The pune literal will not get change, due to this change of one person will not affect for the other persons who are using pune literal

**Why string is immutable in java?**

* String is immutable in java because string objects are cached in string pool. As cached string literals are shared between multiple clients there is always a risk. Where one client’s action would affect all another client

To avoid this string is immutable in java

* Due to string immutability string is thread safe. It is useful in multithreading. A single string instance can be shared across different threads. This avoid uses of synchronization for thread safety
* If new object is created and if it is already present in string pool JVM will not create a new one the new object reference will point to the old literal

**How to create an immutable class in java?**

1. **String is immutable class in java**
2. Declare the class as final so it can’t be extended
3. Make all fields private so the direct access is not allowed
4. Don’t provide setter methods for variable
5. Make all the mutable fields final so that its value can be assigned only at once
6. Initialize all the fields via a constructor performing deep copy
7. Perform cloning of the objects in getter methods to return the copy of objects that the actual object reference

**String Literal vs string object**

**String Literal :** String s1 = “savita”;

**string object:** String s1 = new String(“savita”);

**String Literal :** when we create a string operator using String literal ”\_\_\_”, it will create a new object in string pool and put it into string pool for future re-use

And if literal is already present in string pool it may return an existing object from the string pool

**string object:** when we create a string operator using new keyword, it always creates a new object in heap memory

* String s1 = new String(“savita”);

String internedString = S1.intern();

by using intern function we can manually intern a string into the string pool also

note 1 - unreferenced Strings will be removed from the string pool thereby releasing memory

note 2- JVM can achieve memory optimization by using string pool

when we create a string operator using String literal ”\_\_\_”, And if literal is already present in string pool it may return an existing object from the string pool.

We are storing only one copy of each literal in string pool, this process is called interning

Note 3-garbage collection of string pool

Before java7 string pool is in permanent generation space. So string literals were never garbage collected (which is also led to OutOfMemory issue)

After java7 string pool is placed inheap space, which is garbage collected by the jvm it also reduces the chances of OutOfMemory issue

Garbage collection in java

* Java garbage collection is the process by which Java programs perform automatic memory management.
* Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short.
* When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program.
* Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.0

### **How Java Garbage Collection Works**

* Java garbage collection is an automatic process.
* The programmer does not need to explicitly mark objects to be deleted.
* The garbage collection implementation lives in the JVM. Each JVM can implement garbage collection however it pleases; the only requirement is that it meets the JVM specification. Although there are many JVMs, Oracle’s HotSpot is by far the most common. It offers a robust and mature set of garbage collection options.

**String vs StringBuffer vs StringBuilder**

String Buffer vs String builder are the mutable objects

Which are used for string manipulation

Which provides methods such as substring(), insert(), append(), delete() for string manipulation

|  |  |  |
| --- | --- | --- |
| **String** | **StringBuffer** | **StringBuilder** |
| Stored in string pool | In heap area | In heap area |
| String is immutable | mutable | mutable |
| It is thread safe  Synchronized | It is thread safe  Synchronized  Used in single threading | Not thread safe  Not synchronized  Used in multi threading |
| String str = “savita”; | StringBuffer sbf = new StringBuffer(“savita”); | StringBuilder sbd = new StringBuilder (“savita”); |
|  |  |  |

**Conversion from ont type to another string/stringBuffer/StringBuilder**

**String to StringBuffer/ StringBuilder**

String str = “savita”;

StringBuffer sbf = new StringBuffer(str);

StringBuilder sbd = new StringBuilder (str);

**StringBuffer/ StringBuilder to String**

StringBuffer sbf = new StringBuffer(“savita”);

String str =Sbf.toString();

StringBuilder sbd = new StringBuilder (str);

String str =Sbd.toString();

**StringBuffer to StringBuilder and vice-versa**

StringBuffer sbf = new StringBuffer(“savita”);

String str =Sbf.toString();

StringBuilder sbd = new StringBuilder (str);

Optional class in java

* It is used to avoid null pointer exception
* Optional is a container object may or may not contain a non-null value.
* Optional object is used to represent null with absent value

Methods used in optional class

1. ofNullable()

* in case we expect some null values, we can use the ofNullable() method

String name = null;

Optional<String> opt = Optional.*ofNullable*(name);

**if**(opt.isPresent())

System.***out***.print(name);

**else**

System.***out***.println("value not present");

* By doing this, if we pass in a null reference, it doesn't throw an exception but rather returns an empty Optional object

## isPresent() **and**isEmpty()

When we have an Optional object returned from a method or created by us, we can check if there is a value in it or not with the isPresent() and isEmpty() method

isPresent() method returns true if the wrapped value is not null.

isEmpty() method return false if the wrapped value is not null.

1. ifPresent() which executes a block of code if the value is present.
2. **orElse()** which returns a default value if value not present

Functional interface

* functional interface have only one abstract method
* functional interface can have multiple default methods
* functional interface support the lambda expression
* it can be said lambda expression is the instance of functional interface.

4 main functional interfaces are introduced which could be used in different scenarios.

1. Consumer
   1. BiConsumer
2. Predicate
   1. BiPredicate
3. Function
   1. BiFunction,
   2. UnaryOperator,
   3. BinaryOperator
4. Supplier

**Consumer**

* The consumer interface accepts one argument but there is no return value.
* The name of function inside this interface is accept.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Consumer<T>{**

void accept([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t);

}

Consumer< String > display = s -> System.out.println(s.ToUpperCase);

display.accept(“savita”);

Output: SAVITA

**BiConsumer**

The extension of the Consumer which is BiConsumer accepts two arguments and return nothing.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Consumer<T>{**

void accept([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t, U u);

}

Consumer<String, String> display = (x,y) -> System.out.println(x.ToUpperCase+” ”+y.ToUpperCase);

display.accept(“savita”, “shinde”);

Output: SAVITA SHINDE

Predicate

Predicate will accept one argument, do some processing and then return boolean.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Predicate<T>{**

boolean test([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t);

}

**Predicate** <Integer> p = i ->{return i%2 == 0;};

System.out.println(p.test(10));

Output: true

**BiPredicate**

Instead of one argument BiPredicate will accept two arguments and return boolean.

Function

This interface accepts one argument and return a value after the required processing. It is defined as below. The required processing logic will be executed on invocation of the apply method.

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Function<T>{**

R apply([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html" \o "type parameter in Consumer) t);

}

For example, the above Functional interface will be executed while the following code would be executed.

static Function<String, String> fun = y -> y.ToUpperCase;

System.out.println(fun.apply(“savita”));

Output: SAVITA

Method reference in java

* Java provides a new feature called method reference in Java 8.
* when you are using lambda expression to just referring an existing method, you can replace your lambda expression with method reference.

By using lambda fuction

list.forEach(s -> System.out.println(s));

by using method reference

list.forEach(System.out::println);

* pass a method as a variable,inside the method

// This square function is a variable getSquare.

Function<Integer, Integer> getSquare = i -> i \* i;

SomeFunction(a, b, getSquare);

e.g

public int Add(List<Integer> l, Function<Integer, Integer> ops)

{

-------

-------

}

class Demo {

public static Integer doHalf(Integer x) {

return x / 2;

}

}

List<Integer> list = new ArrayList<>();

// Add some element to list

...

**// Using an anonymous class**

Add(list, **new Function<Integer, Integer>() {**

**public Integer apply(Integer i) {**

**// The method**

**return Demo.doHalf(i);**

**}**

**});**

**// Using a lambda expression**

Add(list, i -> Demo.doHalf(i));

**// Using a method reference**

Add(list, Demo::doHalf);

* **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.

1.Static Method Reference

*(args) -> Class.staticMethod(args)  // by using lambda*

*Class::staticMethod //by using method reference*

*e.g*

**public** **class** Person {

String name;

**int** age;

**public** Person(String name, **int** age) {

**this**.name = name;

**this**.age = age;}

**public** String getName() {

**return** name;}

**public** Integer getAge() {

**return** age;}

@Override

**public** String toString() {

**return** "Person [name=" + name + ", age=" + age + "]";}

}

**public** **class** Main {

**public** **static** **int** compareByName(Person a, Person b){

**return** a.getName().compareTo(b.getName());}

**public** **static** Integer compareByAge(Person a, Person b) {

**return** a.getAge().compareTo(b.getAge());}

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

List<Person> list1 = **new** ArrayList<Person>();

list1.add(**new** Person("savita", 23));

list1.add(**new** Person("engineer", 25));

list1.add(**new** Person("Business Women", 26));

Collections.*sort*(list1, Main::*compareByName*);

list1.forEach(System.***out***::println);

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

Collections.*sort*(list1, Main::*compareByAge*);

list1.forEach(System.***out***::println);

}}

Output

Person [name=Business Women, age=26]

Person [name=engineer, age=25]

Person [name=savita, age=23]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Person [name=savita, age=23]

Person [name=engineer, age=25]

Person [name=Business Women, age=26]

2.Instance Method Reference of a particular object.

*// If a lambda expression just call a default method of an object*

*(args) -> obj.instanceMethod(args)*

*// Shorthand if a lambda expression just call a default method of an object*

*obj::instanceMethod*

*e.g*

**public** **class** Main {

**public** **int** compareByName(Person a, Person b){

**return** a.getName().compareTo(b.getName());}

**public** Integer compareByAge(Person a, Person b) {

**return** a.getAge().compareTo(b.getAge());}

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

List<Person> list1 = **new** ArrayList<Person>();

list1.add(**new** Person("savita", 23));

list1.add(**new** Person("engineer", 25));

list1.add(**new** Person("Business Women", 26));

Main obj = **new** Main();

Collections.*sort*(list1, obj::compareByName);

list1.forEach(System.***out***::println);

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

Collections.*sort*(list1,**new** Main()::compareByAge);

list1.forEach(System.***out***::println);

}}

Output

Person [name=Business Women, age=26]

Person [name=engineer, age=25]

Person [name=savita, age=23]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Person [name=savita, age=23]

Person [name=engineer, age=25]

Person [name=Business Women, age=26]

3.Instance Method Reference of an arbitrary object of a particular type.

*// If a lambda expression just call an instance method of a  ObjectType*

*(obj, args) -> obj.instanceMethod(args)*

*// if a lambda expression just call an instance method of a ObjectType*

*ObjectType::instanceMethod*

*e.g*

**public** **class** Demo {

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

ArrayList<String> list = **new** ArrayList<>(List.*of*("sau","Director","Business Women"));

Collections.*sort*(list, String::compareTo);

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

ArrayList<Integer> list1 = **new** ArrayList<>(List.*of*(30, 10, 40, 5, 70, 50));

Collections.*sort*(list1, Integer::compareTo);

System.***out***.println(list1);

}}

4.Constructor Reference

*// If a lambda expression just create an object   
(args) -> new ClassName(args)*

*// by using method reference*

*ClassName::new*

*e.g*

**interface** Messageable{

    Message getMessage(String msg);

}

**class** Message{

    Message(String msg){

        System.out.print(msg);

    }  }

**public** **class** ConstructorReference {

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

        Messageable hello = Message::**new**;

        hello.getMessage("Hello");

   }  }

Output

helloello

He

Java new features

|  |  |  |
| --- | --- | --- |
| J2SE 5.0 | Sep 2004 | 1. Enhanced for loop 2. Generics 3. Enums 4. Autoboxing |
| Java SE8(LTS) | Mar 2014 | Functional programming  Lambdas  Streams  Static methods in interface |
| Java SE9 | Sep 2014 | Modularization (java platform module system) |
| Java SE10 | Mar 2018 | Var (Local variable type interface) |
| Java SE14 | Mar 2020 | Switch expression(preview in 12 and 13) |
| Java SE15 | Sep 2020 | Text blocks(preview in 13) |
| Java SE16 | Mar 2021 | Record classes(preview in 14 and 15) |
| Java SE17(LTS) | Sep 2021 | It introduces an API by which programs can interpret code and data outside the java runtime  2.5 vector API (second incubator): It introduces an API to express vector computations that reliably compile at runtime to optimal vector instructions |

LTS(long time support release)

In all java versions -> API improvements, performance and garbage collection improvements

Java Enums

* The Enums in java is a data type which contains a fixed set of constants
* All constants are in capital letter always
* e.g It can be used for days of the week

SUNDAY, MONDAY, TUESDAY, WEDNESDAY

e.g For directions

EAST, WEST, NORTH, SOUTH

* Advantages of enum

1. Enum improves time safely
2. Enum can be easily used in switch
3. Enum can be traversed
4. Enum can have fields constructor and methods
5. Enum may impliments many interfaces but can not extend any class because it internally extends enum class

Code

**public** **class** EnumCode {

**enum** Days{

***SUNDAY***, ***MONDAY***, ***TUESDAY***, ***WEDNESDAY***, ***THURSDAY***, ***FRIDAY***, ***SATERDAY***};

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

System.***out***.println(Days.*valueOf*("SUNDAY")); // give SUNDAY

System.***out***.println(Days.*valueOf*("WEDNESDAY").ordinal());//give index

**for** (Days d : Days.*values*()) {

System.***out***.print(d+" ");

}}}

Output

SUNDAY

3

SUNDAY MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY SATERDAY

Modularization

* Modularity explores the creation of a program by using different modules that a single architecture
* It means like microservices in an MVC architecture
* It is introduced in java 9
* Purpose of modularization is -> modularize JDK, modularize application

e.g Modularize JDK

(cmd-> java--list-modules)

List of modules

Java.base

Java.logging

Java.sql

Java.xml

Jdk.compiler

Jdk.jartool

Jdk.Jshell

Module descripter

1. **Module-info.java ->**

Defines metadata about the module

1. exports

in a java project(/module) in module-info.java file write

exports package\_name\_which\_is \_inside\_that\_module;

**exports** com.sau.sorting.util;

so that other modules can use this packages

by doing this you are allowing other modules to use your packages

1. requires

in your module in module-info.java file write

requires module\_name;

**requires** java.logging;

that means you need that module/package/class from somewhere else to do your work

1. requires transitive module\_name

that means I need that module to do my work

and my users also need access to that module

1. opens package.b to module.a

above statements allows module.a access to perform reflection on public types in package.b

before java 9 reflection can be used to find details about types (private, public and protected)

from java 9 you can decide which package to expose

Advantages

* compile time checks for availability of modules
* better encapsulation
* make only a subset of classes from a module available to other modules m
* smaller java runtime
* use only the modules of java that you need

.copyOf()

List/Set/Map.copyOf() it takes collection as an argument

And creates unchangeable copy of List/Set/Map, containing all the elements of the original collection

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

List<Integer> list1 =List.*of*(10, 50, 30, 20);

List<Integer> list2 = List.*copyOf*(list1);

list1.add(90); //give error //can not add elements in the list now

list2.add(70); //give error //can not add elements in the list now

}

String.valueOf(e)

* converts different types of values into the string
* by using String.valueOf(anyTypeOfValue) function you can convert int,char,double,List,Array,object,Boolean to string
* e.g

int i = 40;

String s1 = String.valueOf(i);

How to take int ,string, double from scanner at the same time

* when we are taking string after the integer/Double from the scanner(user)
* add sc.nextLine(); in between the int and String input
* because nextInt() method doesn’t read the newline character(enter of keyboard is newline character) of your input

so when you issue the command nextLine() the scanner finds the newline character and gives you that newLine

code

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

Scanner sc = **new** Scanner(System.***in***);

**int** i = sc.nextInt();

sc.nextLine(); // add sc.nextLine() in between//when taking String after integer or double

String s = sc.nextLine();

Double d = sc.nextDouble();

sc.nextLine();// add sc.nextLine() in between//when taking String after integer or double

String s1 = sc.nextLine();

System.***out***.println(i+"\n"+s+"\n"+d+"\n"+s1);

}

Output

5

fgt ghu ijh tyu huji

9.8

ghty ghyt gjhu jfkri khi

5

fgt ghu ijh tyu huji

9.8

ghty ghyt gjhu jfkri khi