**What is Java Development Kit (JDK)?**

Software development environment that offers a collection of tools and libraries necessary for developing Java applications.

JDK converts source code into a format that the Java Runtime Environment (JRE) can execute.

Programmer needs to install JDK to build application.

**What is Java Runtime Environment (JRE)?**

A software which is designed to run other software.

It contains the class libraries, loader class, and JVM.

To run Java program, JRE is required.

**What is Java Virtual Machine (JVM)?**

An engine that provides a runtime environment to drive the Java Code or applications.

It converts Java bytecode into machine language.

Part of JRE.

**Primitive Data Type in Java**

Specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type Size Description

byte 1 byte Stores whole numbers from -128 to 127

short 2 bytes Stores whole numbers from -32,768 to 32,767

int 4 bytes Stores whole numbers from -2,147,483,648 to 2,147,483,647

long 8 bytes Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float 4 bytes Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits

double 8 bytes Stores fractional numbers. Sufficient for storing 15 decimal digits

boolean 1 bit Stores true or false values

char 2 bytes Stores a single character/letter or ASCII values

**Typecasting**

Assigning a value of one primitive data type to another type.

Widening Casting (automatically) - converting a smaller type to a larger type size

byte -> short -> char -> int -> long -> float -> double

int myInt = 9;

double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9

System.out.println(myDouble); // Outputs 9.0

Narrowing Casting (manually) - converting a larger type to a smaller size type

double -> float -> long -> int -> char -> short -> byte

double myDouble = 9.78;

int myInt = (int) myDouble; // Manual casting: double to int

System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt); // Outputs 9

**Parsing**

Int number = Integer.parseInt(string);

**Wrapper Class in Java**

Class whose object wraps or contains primitive data types.

// int data type

int b = 10;

//wrapping around Integer object

Integer intobj = new Integer(b);

**What is Class?**

Class is a blueprint that defines the variables and the methods. Class in Java determines how an object will behave and what the object will contain.

**What is Method?**

A method in object-oriented programming is a procedure associated with a class. A method defines the behaviour of the objects that are created from the class.

**Static Method**

Can’t access instance method and instance variables directly.

Used for operation that does not require any data from the instance of the class.

Keyword this cannot be used.

Does not require instance to be created.

|  |
| --- |
| class Calculator {  public static void printSum (int a, int b) {  System.*out*.println(a+b);  } } |
| public static void main(String args[]){  Calculator.*printSum*(5,10); //Calculator is class here } |

**Instance Method**

Belongs to instance of a class.

To use, class needs to be instantiated.

Can access static methods, instance method, static variables and instance variable directly.

**What is Object?**

Object is an instance of a class.

Java always has reference to object in memory.

No way accessing the object directly.

**Static Variable & Block**

Means there is only one copy of the variable in memory shared by all instances of the class.

Change made in one object will be shared by others object.

Used for static initializations of a class. This code inside static block is executed only once: the first time the class is loaded into memory.

Variable and block are executed in the order they appear.

class Test

{

    // static variable

    static int a = m1();

    // static block

    static {

        System.out.println("Inside static block");

    }

    // static method

    static int m1() {

        System.out.println("from m1");

        return 20;

    }

    // static method(main !!)

    public static void main(String[] args)

    {

       System.out.println("Value of a : "+a);

       System.out.println("from main");

    }

}

Output:

from m1

Inside static block

Value of a : 20

from main

**Final in Java**

Java final variable: If you make any variable as final, you cannot change the value of final variable (It will be constant).

Java final method: If you make any method as final, you cannot override it.

Java final class: If you make any class as final, you cannot extend it.

Static blank final variable: A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

class A{

static final int data;//static blank final variable

static{ data=50;}

public static void main(String args[]){

System.out.println(A.data);

}

}

**Get() & Set()**

Set method is used to access the private variables of a class from another method.

Get method is used to retrieve the value of the private variable in a class from another method.

|  |
| --- |
| public class Car {  private String model;  public void setModel(String model){  this.model = model; *// this refers to variable declared in the class* }   public String getModel(){  return this.model; *//returns the set variable* } } |
| public class Main {   public static void main(String[] args) {  *// write your code here* Car porsche */\* Object of the class \*/* = new Car(); *//Initialization* Car holden = new Car();   porsche.setModel("Carrera");  System.*out*.println(porsche.getModel());  } |

**Constructor**

Special method that is used to initialize objects.

Is called when an object of a class is created.

It can be used to set initial values for object attributes:

|  |
| --- |
| public class BankAccount {  private String accountNO;  private double balance;  private String name;  private String mailID;  private int contactNO;   *//Constructors* public BankAccount(String accountNO, double balance, String name, String mailID, int contactNO) {  this.accountNO = accountNO;  this.balance = balance;  this.name = name;  this.mailID = mailID;  this.contactNO = contactNO;  } }  public BankAccount(){ *//Empty Constructor, which calls the previous one. Chain of constructors* this("123456" , 0.0 , "No Name" , "nomail@nomail.com" , 0000000); *//Should be the first line always* } |
| public class Main {   public static void main(String[] args) { BankAccount bankAccount1 = new BankAccount();  BankAccount bankAccount2 = new BankAccount("123456", 0.0, "Santanu", "santanu@dhar.com", 999888);  } |

**Inheritance**

Mechanism in which one object acquires all the properties and behaviours of a parent object.

Represents the IS-A relationship which is also known as a parent-child relationship.

Used for method overriding to achieve runtime polymorphism.

|  |
| --- |
| public class Vehicle {  private String name;  private String size;  private int velocity;  private int direction;   public Vehicle(String name, String size){  this.name = name;  this.size = size;  this.velocity = 0;  this.direction = 0;  }  public void stop(){  System.*out*.println("Vehicle stopped");  }  } |
| public class Car extends Vehicle{   private int wheels;  private int doors;  private int gears;  private boolean isManual;  private int currentGear;   public Car(String name, String size, int wheels, int doors, int gears, boolean isManual) {  super(name, size); //Initialize base class variable  this.wheels = wheels;  this.doors = doors;  this.gears = gears;  this.isManual = isManual;  this.currentGear = 1;  }  //Runtime polymorphism @Override public void stop() {  System.*out*.println("Car stopped"); }  } |

**Composition**

implement a has-a relationship.

Achieved by using an instance variable that refers to other objects.

When there is a composition between two entities, the composed object cannot exist without the other entity.

|  |
| --- |
| public class Resolution {  private int width;  private int height;   public Resolution(int width, int height) {  this.width = width;  this.height = height;  }   public int getWidth() {  return width;  }   public int getHeight() {  return height;  } } |
| public class Monitor {  private String model;  private String manufacturer;  private int size;  private Resolution nativeResolution;   public Monitor(String model, String manufacturer, int size, Resolution nativeResolution) {  this.model = model;  this.manufacturer = manufacturer;  this.size = size;  this.nativeResolution = nativeResolution;  }   public void drawPixelAt(int x, int y, String color){  System.*out*.println("Drawing pixel at " + x + "," + y + " in color " + color);  }   public String getModel() {  return model;  }   public String getManufacturer() {  return manufacturer;  }   public int getSize() {  return size;  }   public Resolution getNativeResolution() {  return nativeResolution;  } } |
| public class Main {   public static void main(String[] args) {  Dimensions dimensions = new Dimensions(20, 20, 5);  Case theCase = new Case("220B", "Dell", "240", dimensions);  Monitor monitor = new Monitor("27inch Beast", "Acer", 27, new Resolution(2540, 1440));  Motherboard motherboard = new Motherboard("BJ-200", "Asus", 4, 6, "v2.44");   PC thePC = new PC(theCase,monitor, motherboard);  thePC.getMonitor().drawPixelAt(1200,1200,"red");  thePC.getMotherboard().loadProgram("Windows 1.0");  thePC.getTheCase().pressPowerButton();  } } |

**Encapsulation**

Variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java

Declare the variables of a class as private.

Provide public setter and getter methods to modify and view the variables values.

**Polymorphism**

Occurs when we have many classes that are related to each other by inheritance.

Inheritance allows to inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Run time – Overriding

Compile time – Overloading

class Animal {  
 public void animalSound() {  
 System.*out*.println("The animal makes a sound");  
 }  
}  
  
class Pig extends Animal {

@Override  
 public void animalSound() {  
 System.*out*.println("The pig says: wee wee");  
 }  
}  
  
class Dog extends Animal {

@Override  
 public void animalSound() {  
 System.*out*.println("The dog says: bow wow");  
 }  
}

**this()**

Used to call the current class member. Commonly used in constructor and setter.

**Super()**

Super is used to call the parent class member. Commonly used in method overriding.

**Overloading**

Providing two or more methods in same class with same name but different parameters.

Return type may vary thus allow to reuse same method name.

Static and Instance method can be overloaded.

**Overriding**

Defining a method in child class that is already defined in parent class with same signature (name, parameter).

Must return same type.

Runtime polymorphism, because method to be called is decided during runtime by JVM. **Dynamic Method Dispatch**.

@Override is put immediately over above the method definition. Compiler reads this and tells if overriding rules are not followed.

Static method, constructor, private methods can’t be override. Only instance method.

**How to read user input in Java and Ternary Operator**

import java.util.Scanner;  
  
class Main {  
 public static void main(String[] args) {  
  
 *// take input from users* Scanner input = new Scanner(System.*in*);  
 System.*out*.println("Enter your marks: ");  
 double marks = input.nextDouble();  
  
 *// ternary operator checks if  
 // marks is greater than 40* String result = (marks > 40) ? "pass" : "fail";  
  
 System.*out*.println("You " + result + " the exam.");  
 input.close();  
 }  
}

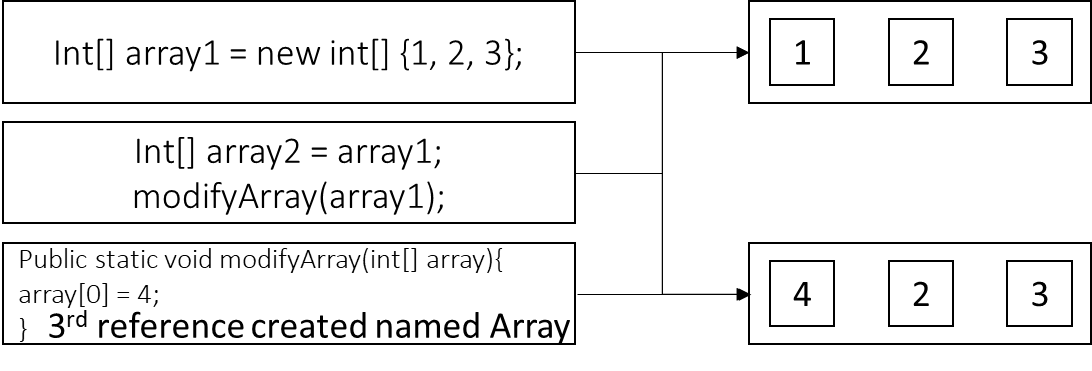
**Array in Java**

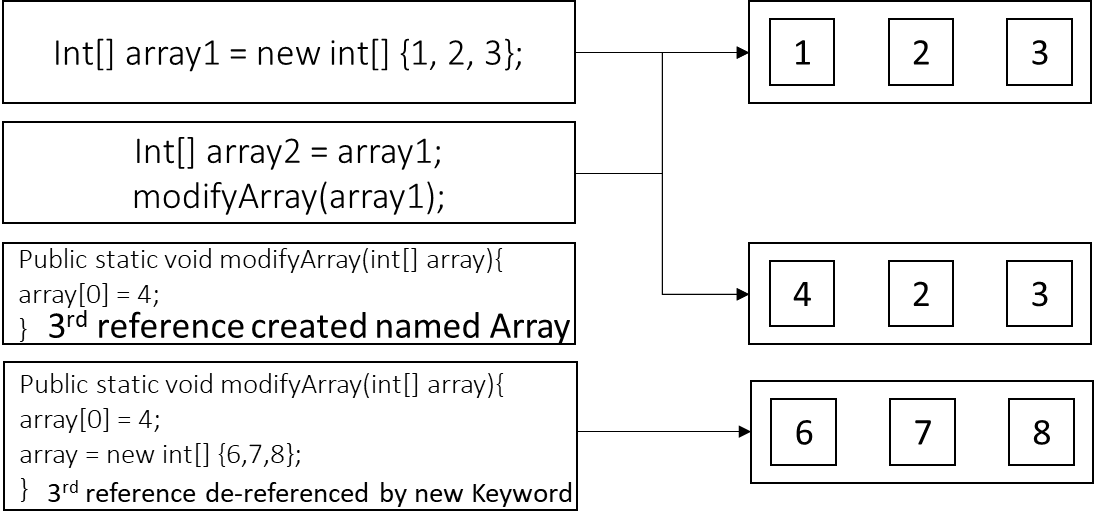
Data structure that allows to store multiple values of the same datatype into a single variable.

Int[] a = new int[array\_lenght];

Arrays.sort(a); //Sorting

**Reference of Array**





**ArrayList**

The ArrayList class is a resizable array, which can be found in the java.util package.

Elements can be added and removed from an ArrayList whenever you want.

Does not work with primitive data type.

import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

cars.add("Volvo"); //Add an item

cars.add(1,”Alto”); //Add item at index 1 and pushes the list by 1 index

cars.get(0); //Get an item in a position

cars.set(0, "Opel"); //Replace an item in a position

cars.remove(0); //Remove an item from a position

cars.size(); //Get the size of the array

cars.contains(“Volvo); //Check whether an item is present in array

cars.indexOf(“Opel”); //Returns the first occurrence of the item in the array

**Autoboxing and Unboxing in Java**

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing.

Opposite operation is known as unboxing.

ArrayList<Integer> arrayListInt = new ArrayList<Integer>();  
for (int i=0; i<10; i++){  
 arrayListInt.add(Integer.*valueOf*(i)); *//Autoboxing  
 //Alternate Method* arrayListInt.add(i); *//Runtime java converts to Interger.valueOf(i)*}  
  
for (int i=0; i<10; i++){  
 System.*out*.println(arrayListInt.get(i).intValue()); *//Unboxing  
 //Alternate Method* System.*out*.println(arrayListInt.get(i)); *//Runtime java converts to get(i).intValue()*}

**Iterator**

Object that can be used to loop through collections.

import java.util.ArrayList;  
import java.util.Iterator;  
  
public class Main {  
 public static void main(String[] args) {  
  
 *// Make a collection* ArrayList<String> cars = new ArrayList<String>();  
 cars.add("Volvo");  
 cars.add("BMW");  
 cars.add("Ford");  
 cars.add("Mazda");  
  
 *// Get the iterator* Iterator<String> it = cars.iterator();  
  
 *// Print the all item* while(it.hasNext())  
 System.*out*.println(it.next());  
 }  
}

**ListIterator**

ArrayList<Integer> list = new ArrayList<Integer>();   
 list.add(1);  
 list.add(2);  
 list.add(3);  
 list.add(4);  
 list.add(5);  
  
 ListIterator i = list.listIterator();  
  
 while (i.hasNext())  
 System.out.print(i.next() + " ");  
  
 while (i.hasPrevious())  
 System.out.print(i.previous() + " ");

**Differences between Iterator and ListIterator:**

|  |  |
| --- | --- |
| **ITERATOR** | **LISTITERATOR** |
| Can traverse elements present in Collection only in the forward direction. | Can traverse elements present in Collection both in forward and backward directions. |
| Helps to traverse Map, List and Set. | Can only traverse List and not the other two. |
| Indexes cannot be obtained by using Iterator. | It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List. |
| Cannot modify or replace elements present in Collection | We can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time. |
| Certain methods of Iterator are next(), remove() and hasNext(). | Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e). |

**LinkedList**

Stores its items in "containers."

The list has a link to the first container and each container has a link to the next container.

To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

LinkedList<String> placesToVisit = new LinkedList<String>();

*addInOrder*(placesToVisit,"Sydney");  
*addInOrder*(placesToVisit,"Melbourne");  
*addInOrder*(placesToVisit,"Brisbane");  
*addInOrder*(placesToVisit,"Canberra");

private static boolean addInOrder(LinkedList<String> linkedList, String newCity){  
 ListIterator<String> stringListIterator = (ListIterator<String>) linkedList.iterator();  
  
 while(stringListIterator.hasNext()){  
 int comparison = stringListIterator.next().compareTo(newCity);  
 if(comparison==0){  
 System.*out*.println(newCity + " is alrady included in destination");  
 return false;  
 } else if (comparison>0){  
 stringListIterator.previous();  
 stringListIterator.add(newCity);  
 return true;  
 } else if(comparison < 0){  
  
 }  
 }  
 stringListIterator.add(newCity);  
 return true;  
}

**Difference Between ArrayList and LinkedList**

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

**Abstraction and Abstraction Class**

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

An abstract class must be declared with an abstract keyword.

It can have abstract and non-abstract methods.

It cannot be instantiated.

It can have constructors and static methods also.

It can have final methods which will force the subclass not to change the body of the method.

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

|  |
| --- |
| abstract class Bank{  abstract int getRateOfInterest();  } |
| class SBI extends Bank{  int getRateOfInterest(){return 7;}  }  class PNB extends Bank{  int getRateOfInterest(){return 8;}  } |
| class TestBank{  public static void main(String args[]){  Bank b;  b=new SBI();  System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  b=new PNB();  System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");  }} |

**Interface**

Completely "abstract class" that is used to group related methods with empty bodies.

Like abstract classes, interfaces cannot be used to create objects/ instantiated. No constructor.

Interface methods do not have a body - the body is provided by the "implement" class.

On implementation of an interface, you must override all of its methods.

Interface methods are by default abstract and public.

Used for:

To achieve security - hide certain details and only show the important details of an object (interface).

Multiple inheritance - class can implement multiple interfaces.

|  |  |  |
| --- | --- | --- |
| interface FirstInterface {  public void myMethod();  }  interface SecondInterface {  public void myOtherMethod();  } | class DemoClass implements FirstInterface, SecondInterface {  public void myMethod() {  System.out.println("Some text..");  }  public void myOtherMethod() {  System.out.println("Some other text...");  }  } | class Main {  public static void main(String[] args) {  DemoClass myObj = new DemoClass();  myObj.myMethod();  myObj.myOtherMethod();  }  } |

ITelephone santanuPhone = new DeskPhone(999888777);  
santanuPhone = new MobilePhone(1122333);

ITelephone is Inheritance Class.

DeskPhone, MobilePhone are regular Class.

Advantage: Using Interface, same Interface variable can be used to instantiate various class objects who implement the interface.

**Abstract Class vs Interface**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 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. |
| Abstract class doesn't support multiple inheritance. | Interface supports multiple inheritance. |
| Abstract class can have final, non-final, static and non-static variables. | Interface has only static and final variables. |
| Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| An abstract class can extend another Java class and implement multiple Java interfaces. | An interface can extend another Java interface only. |
| A Java abstract class can have class members like private, protected, etc. | Members of a Java interface are public by default. |

**Inner Class or Nested Class**

Java inner class or nested class is a class which is declared inside the class or interface.

Logically group classes and interfaces in one place so that it can be more readable and maintainable.

Can access all the members of outer class including private data members and methods.

|  |
| --- |
| public class GearBox {  private ArrayList<Gear> gears;  private int maxGears;  private int currentGear = 0;   public GearBox(int maxGears) {  this.maxGears = maxGears;  this.gears = new ArrayList<>(); *// Gear neutral = new Gear(0, 0.0); // this.gears.add(neutral);* }  *// public void addgear(int gearNumber, double ratio){ // this.gears.add(new Gear(gearNumber, ratio)); // }* public class Gear{  private int gearNumber;  private double ratio;   public Gear(int gearNumber, double ratio) {  this.gearNumber = gearNumber;  this.ratio = ratio;  }   public double driveSpeed(int revs){  return revs \* (this.ratio);  }  } } |
| public class Main {   public static void main(String[] args) {  *// write your code here* GearBox mcLaren = new GearBox(6);  GearBox.Gear first = mcLaren.new Gear(1, 12.3); //Calling the inner class  System.*out*.println(first.driveSpeed(1000));  } } |

**Generics in Java**

Enable programmers to specify, with a single class declaration, a set of related types or with a single method declaration, a set of related methods.

Advantage:

Type-safety: We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

Type casting is not required.

It is checked at compile time so problem will not occur at runtime.

|  |
| --- |
| public class GenericClass<T> {   private T variable;   public GenericClass(T var){  this.variable = var;  }   public T getVariable(){  return this.variable;  } } |
| public class Main {   public static void main(String[] args) {  *// write your code here* GenericClass var = new GenericClass("Santanu");  GenericClass var1 = new GenericClass(12);  GenericClass var2 = new GenericClass(1.24);   System.*out*.println(var.getVariable());  System.*out*.println(var1.getVariable());  System.*out*.println(var2.getVariable());  } } |