**OOPS in Java**

Object-Oriented Programming is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

Class

Class is a named group of properties and functions/methods.

A class in Java is a set of objects which shares common characteristics/ behaviour and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created.

**Properties of Java Classes**

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:

* Data member
* Method
* Constructor
* Nested Class
* Interface

**class** Student{

**int** rollNo;

String name;

**int** marks;

};

Student stu1 = **new** Student();

**new keyword** in above will allocate dynamic memory allocation.

**Constructor**

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

public class Main {

int x; // Create a class attribute

public Main() {

x = 5; // Set the initial value for the class attribute x

}

public static void main(String[] args) {

Main myObj = new Main();

System.out.println(myObj.x); // Print the value of x

}

}

**This keyword**

The this keyword refers to the current object in a method or constructor.

this can also be used to:

* Invoke current class constructor
* Invoke current class method
* Return the current class object
* Pass an argument in the method call
* Pass an argument in the constructor call

**package** ObjectOrientedProgramming;

**public** **class** ThisKeyword {

**int** rollNo;

String name;

**int** marks;

// Constructor

ThisKeyword(**int** rollNo, String name, **int** marks) {

**this**.rollNo = rollNo;

**this**.name = name;

**this**.marks = marks;

}

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

ThisKeyword key1 = **new** ThisKeyword(10, "Prithviraj", 99);

ThisKeyword key2 = **new** ThisKeyword(11, "Amogh", 90);

System.***out***.println(key1.rollNo);

System.***out***.println(key1.name);

System.***out***.println(key1.marks);

System.***out***.println(key2.rollNo);

System.***out***.println(key2.name);

System.***out***.println(key2.marks);

}

}

**Final Keyword:**

Final keyword is used for strict variables means that variables values cannot be changed or updated or It is just immutable.

Final variable should be initialized cumpolsurly

**final** **int** num = 10;

num = 20; //getting error because of final keyword

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

**Garbage collection**

Garbage collection in Java is the process by which Java programs perform automatic memory management.

[**Static**](https://www.geeksforgeeks.org/static-keyword-java/)

A single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

* Static is not only object specific.
* Static is whole class specific.
* Static method can call only static methods not the non static methods.
* For static we don’t require **this** keyword.
* Outer class cannot be static.

**int** num;

**static** String *ceo*;

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

// **TODO** Auto-generated method stub

StaticVariable emp1 = **new** StaticVariable();

StaticVariable emp2 = **new** StaticVariable();

emp1.num = 10;

emp1.*ceo* = "Prithviraj";

emp2.num = 11;

System.***out***.println(emp1.num + " " + emp1.*ceo*); //10 Prithviraj

System.***out***.println(emp2.num + " " + emp2.*ceo*); //11 Prithviraj

**Singleton**

Singleton class is a class that can have only one object (an instance of the class) at a time.

public class Singleton {

private static Singleton instance;

private Singleton() {

}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

public class Main {

public static void main(String[] args) {

// Get the instance of the Singleton class

Singleton singletonInstance = Singleton.getInstance();

// Use the singletonInstance object as needed

}

}

OOPS

**Constructor**

Java constructors or constructors in Java is a terminology used to construct something in our programs. A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

**package** com.JavaApplication;

**public** **class** Box {

**double** l;

**double** w;

**double** h;

Box(){

**this**.h = -1;

**this**.w = -1;

**this**.l = -1;

}

Box(**double** side){

**this**.h = side;

**this**.w = side;

**this**.l = side;

}

Box(Box old){

**this**.l = old.l;

**this**.w = old.w;

**this**.h = old.h;

}

}

**package** com.JavaApplication;

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

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

Box box = **new** Box();

System.***out***.println(box.l + " " + box.w + " " + box.h);

}

}

**Box:**

This is the name of a class. In Java, a class is a blueprint or template for creating objects. The Box class could potentially define properties (attributes) and behaviors (methods) that describe a box.

**box:**

This is a variable name. Variables are used to store data or references to objects. In this context, box is a variable that will hold a reference to an object of type Box.

**=:**

This is the assignment operator. It assigns a value to a variable. In this case, it assigns a new instance of the Box object to the box variable.

**new:**

This keyword is used to create a new instance (object) of a class. It allocates memory for the new object and returns a reference to it.

**Box():**

This is a constructor call. A constructor is a special method in a class that has the same name as the class and is used to initialize the newly created object. When new Box() is called, it invokes the constructor of the Box class to create a new Box object.

So, when you write Box box = new Box();, you are declaring a variable box of type Box and assigning to it a new instance of the Box object. This means you are creating a new box (object) based on the Box class blueprint and storing a reference to it in the box variable.

**super() in Java**

The super keyword in Java is a reference variable that is used to refer parent class objects.

**package** com.JavaApplication;

**public** **class** BoxWeight **extends** Box {

**double** weight = 1;

BoxWeight(){

**this**.weight = weight;

}

BoxWeight(**double** l, **double** w, **double** h, **double** weight){

**super**(l,h,w); //calls the parent class constructor used to initialize values present in parent class

**this**.weight = weight;

}

}

**Inheritance**

It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class.

**public** **class** BoxWeight **extends** Box {

**double** weight = 1;

BoxWeight(){

**this**.weight = weight;

}

}

**Polymorphism**

It means many ways to represent it.

* Compile-time Polymorphism
* Runtime Polymorphism

**package** com.JavaApplication;

**public** **class** MethodOverLoading {

**int** sum(**int** a, **int** b) {

**return** a+b;

}

**int** sum(**int** a, **int** b, **int** c) {

**return** a+b+c;

}

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

// **TODO** Auto-generated method stub

MethodOverLoading num = **new** MethodOverLoading();

System.***out***.println(num.sum(10, 10));

System.***out***.println(num.sum(10, 10,10));

}

}

Encapsulation

Wrapping up the implementation of data members and methods in a class

**private** **double** l;

**double** w;

**double** h;

Box(){

**this**.h = -1;

**this**.w = -1;

**this**.l = -1;

}

**public** **double** getL() {

**return** l;

}

Box box = **new** Box();

System.***out***.println(box.getL());

**Abstraction**

**Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

// Abstract class

abstract class Animal {

abstract void makeSound();

void sleep() {

System.out.println("The animal sleeps");

}

}

class Dog extends Animal {

@Override

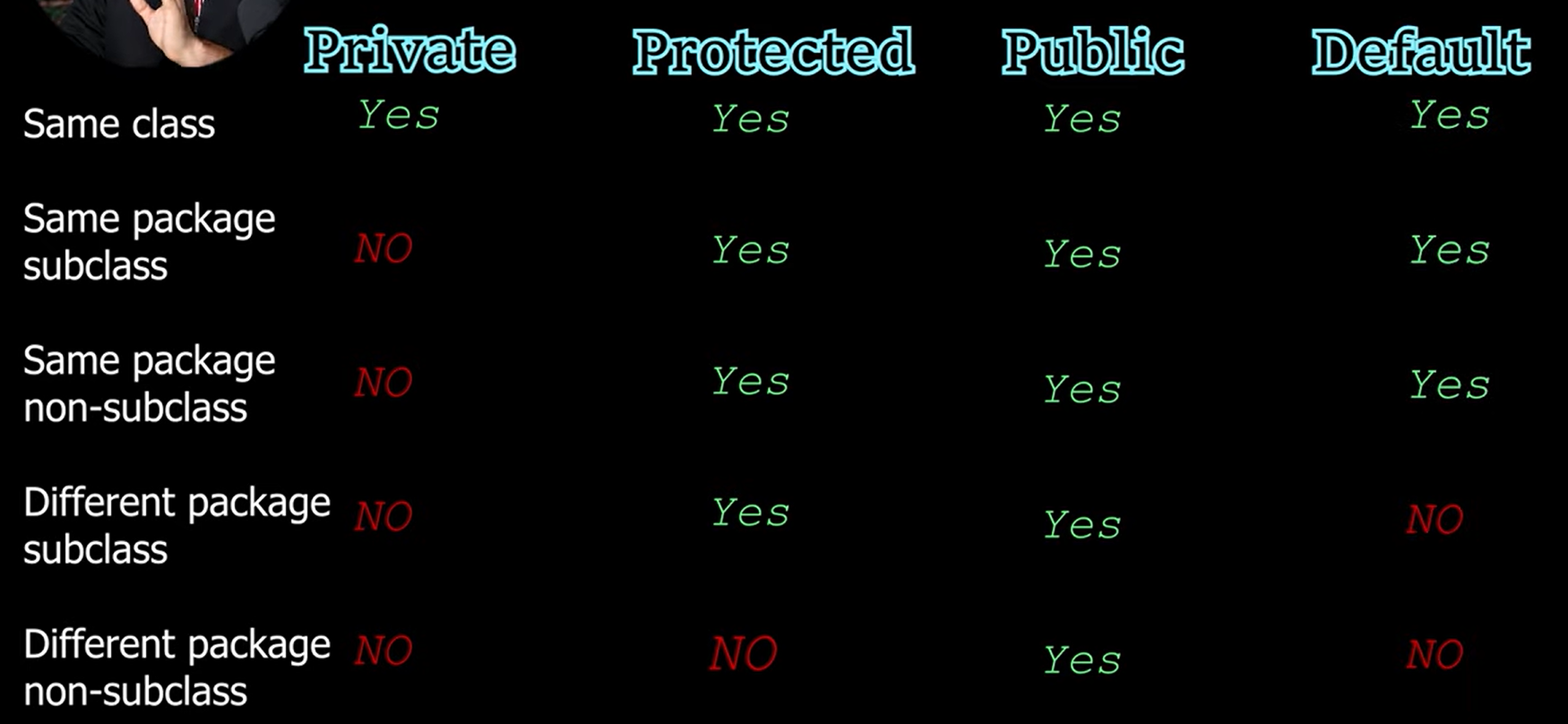
void makeSound() {

System.out.println("Dog barks");

}

}

**Access Modifiers**



**Packages**

Built in package

Lang -> Provides fundamental classes and interfaces that are automatically imported into every Java program. Includes classes like Object, String, System, Math, etc.

IO -> Provides classes for input and output operations, including file handling. Includes classes like File, FileInputStream, FileOutputStream, BufferedReader, BufferedWriter, etc.

Util -> Contains utility classes and data structures such as ArrayList, HashMap, HashSet, Date, Calendar, Scanner, etc.

java.net: Contains classes for networking operations. Includes classes like URL, URLConnection, Socket, ServerSocket, etc.

**Abstraction**

**Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

**package** com.JavaApplication;

**abstract** **public** **class** AbstractionCLass {

**abstract** **void** main();

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

AbstractionCLass obj = **new** anotherAbstrac();

obj.main();

}

}

**abstract** **class** abstractClassExt **extends** AbstractionCLass{

// void main() {

// System.out.println("This is main abstraction class");

// }

**void** display() {

System.***out***.println("This is display method");

}

}

**class** anotherAbstrac **extends** abstractClassExt{

**void** main() {

System.***out***.println("This is main abstraction class");

}

}

**Interface**

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

**package** com.JavaApplication;

**interface** A{

**int** ***num*** = 90; //we can only initialize variable cannto just declare because in iterface it is final and static

**void** main();

**void** display();

}

**public** **class** Interface **implements** A {

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

Interface obj = **new** Interface();

obj.display();

obj.main();

System.***out***.println(obj.***num***);

}

@Override

**public** **void** main() {

System.***out***.println("This is main method");

}

@Override

**public** **void** display() {

System.***out***.println("This is display method");

}

}

Extends interface

**interface** A {

**void** method1();

**void** method2();

}

// B now includes method1 and method2

**interface** B **extends** A {

**void** method3();

}

// the class must implement all method of A and B.

**class** gfg **implements** B {

**public** **void** method1()

{

System.***out***.println("Method 1");

}

**public** **void** method2()

{

System.***out***.println("Method 2");

}

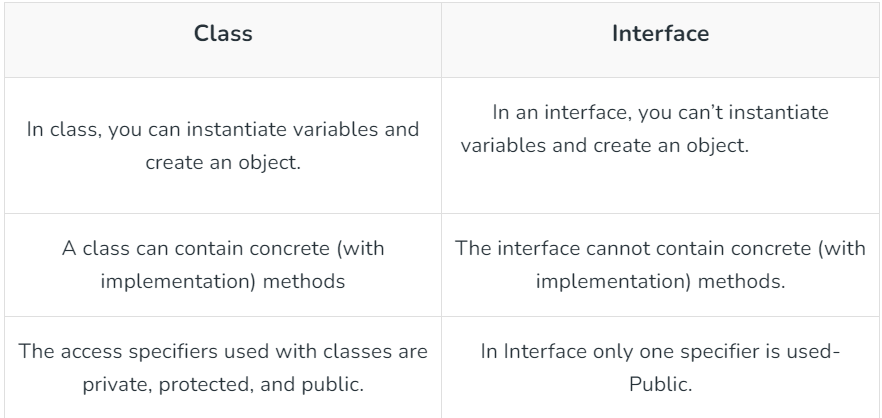
**public** **void** method3()

{

System.***out***.println("Method 3");

}

}



**Annotations**

Annotations are used to provide supplemental information about a program.

* Annotations start with ‘**@**’.
* Annotations do not change the action of a compiled program.
* Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
* Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
* Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.

**package** com.JavaApplication;

**class** A{

**void** showingTheAnnotation() {

System.***out***.println("This is A class");

}

**void** showingThennotation() {

// **TODO** Auto-generated method stub

}

}

**class** B **extends** A{

@Override

**void** showingThennotation() {

System.***out***.println("This is B class");

}

}

**public** **class** Anotaion {

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

// **TODO** Auto-generated method stub

B obj = **new** B();

obj.showingTheAnnotation();

}

}

**Generics**

Java Generics allows us to create a single class, [interface](https://www.programiz.com/java-programming/interfaces), and [method](https://www.programiz.com/java-programming/methods) that can be used with different types of data (objects).

class Box<T> {

private T item;

public Box(T item) {

this.item = item;

}

public T getItem() {

return item;

}

}

public class Main {

public static void main(String[] args) {

// Creating a box for Integer

Box<Integer> integerBox = new Box<>(10);

System.out.println("Integer value in the box: " + integerBox.getItem());

Box<String> stringBox = new Box<>("Hello, World!");

System.out.println("String value in the box: " + stringBox.getItem());

}

}

**Comparable interface:** This interface allows objects to be compared with one another based on their natural ordering. Classes that implement Comparable must override the compareTo() method.

**package** ObjectOrientedProgramming;

**class** Person **implements** Comparable<Person> {

**private** String name;

**private** Integer rollNo;

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

**this**.name = name;

**this**.rollNo = rollNo;

}

**public** Integer getName() {

**return** rollNo;

}

@Override

**public** **int** compareTo(Person otherPerson) {

**return** **this**.rollNo.compareTo(otherPerson.getName());

}

}

**public** **class** ComparableObject {

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

// **TODO** Auto-generated method stub

Person person2 = **new** Person("Bobybob",10);

Person person1 = **new** Person("Alice",11);

**if** (person1.compareTo(person2) < 0) {

System.***out***.println(person1.getName() + " comes before " + person2.getName());

} **else** **if** (person1.compareTo(person2) > 0) {

System.***out***.println(person1.getName() + " comes after " + person2.getName());

} **else** {

System.***out***.println(person1.getName() + " is equal to " + person2.getName());

}

}

}

**Exception Handling** in Java is one of the effective means to handle runtime errors so that the regular flow of the application can be preserved.

**try** {

**int** a = 4;

**int** b = 0;

**int** c = a/b;

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

} **catch**(Exception e) {

System.***out***.println(e.getMessage());

}

**package** ObjectOrientedProgramming;

**public** **class** MyException **extends** Exception {

**public** MyException(String name) {

// **TODO** Auto-generated constructor stub

**super**(name);

}

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

// **TODO** Auto-generated method stub

**try** {

String name = "appu";

**if**(name.equals("appu")) {

**throw** **new** MyException("name is appu");

}

} **catch** (Exception e) {

System.***out***.println(e.getMessage());

} **finally** {

System.***out***.println("FInally block is called");

}

}

}