Unit 2

Java Classes/Objects

- A class is a blueprint from which individual objects are created (or, we can say a class is a data type of an object type).
- In Java, everything is related to classes and objects.
- Each class has its methods and attributes that can be accessed and manipulated through the objects.
- Example: if you want to create a class for *students*. In that case, "*Student*" will be a class, and student records (like *student1*, *student2*, etc) will be objects.

Properties of Java Classes

- A class does not take any byte of memory.
- A class is just like a real-world entity, but it is not a real-world entity. It's a blueprint where we specify the functionalities.
- Class is a group of variables of different data types and a group of methods.
- A Class in Java can contain:
  - Data member
  - Method
  - Constructor
  - Nested Class
  - Interface
- Classes follow all of the rules of OOPs such as inheritance, encapsulation, abstraction, etc.

Types of Class Variables

- A class can contain any of the following variable types.
  - Local variables − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  - Instance variables − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
  - Class variables − Class variables are variables declared within a class, outside any method, with the static keyword.

Components of Java Classes

- Class declarations can include these components, in order:
  - Modifiers: A class can be public or has default access.
  - Class keyword: Class keyword is used to create a class.
  - Class name: The name should begin with an initial letter (capitalized by convention).
  - Superclass (if any): In Java, a superclass is a class from which another class is derived. The class that inherits from the superclass is called the subclass.
  - Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
  - Body: The class body is surrounded by braces, { }.

Syntax to create a Java class

- Use the below syntax to create (declare) class in Java:

```
access_modifier class class_name{
        data members;
        constructors;
        methods;
        ...;
}
```

- Example of a Java Class
  - In this example, we are creating a class "Dog". Where, the class attributes are breed, age, and color. The class methods are setBreed(), setAge(), setColor(), and printDetails().

```
// Creating a Java class
class Dog {
  // Declaring and initializing the attributes
  String breed;
  int age;
  String color;

  // methods to set breed, age, and color of the dog
public void setBreed(String breed) {
    this.breed = breed;
  }
  public void setAge(int age) {
    this.age = age;
  }
  public void setColor(String color) {
    this.color = color;
  }

  // method to print all three values
  public void printDetails() {
    System.out.println("Dog detials:");
    System.out.println(this.breed);
    System.out.println(this.age);
    System.out.println(this.color);
  }
}
```

Java Objects

- An object is a variable of the type class, it is a basic component of an object-oriented programming system.
- A class has the methods and data members (attributes), these methods and data members are accessed through an object.
- Thus, an object is an instance of a class.
- Example: If we consider the real world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior.

Creating (Declaring) a Java Object

- An object is created from a class. In Java, the **new** keyword is used to create new objects.
- There are three steps when creating an object from a class –
  - Declaration – A variable declaration with a variable name with an object type.
  - Instantiation – The 'new' keyword is used to create the object.
  - Initialization – The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

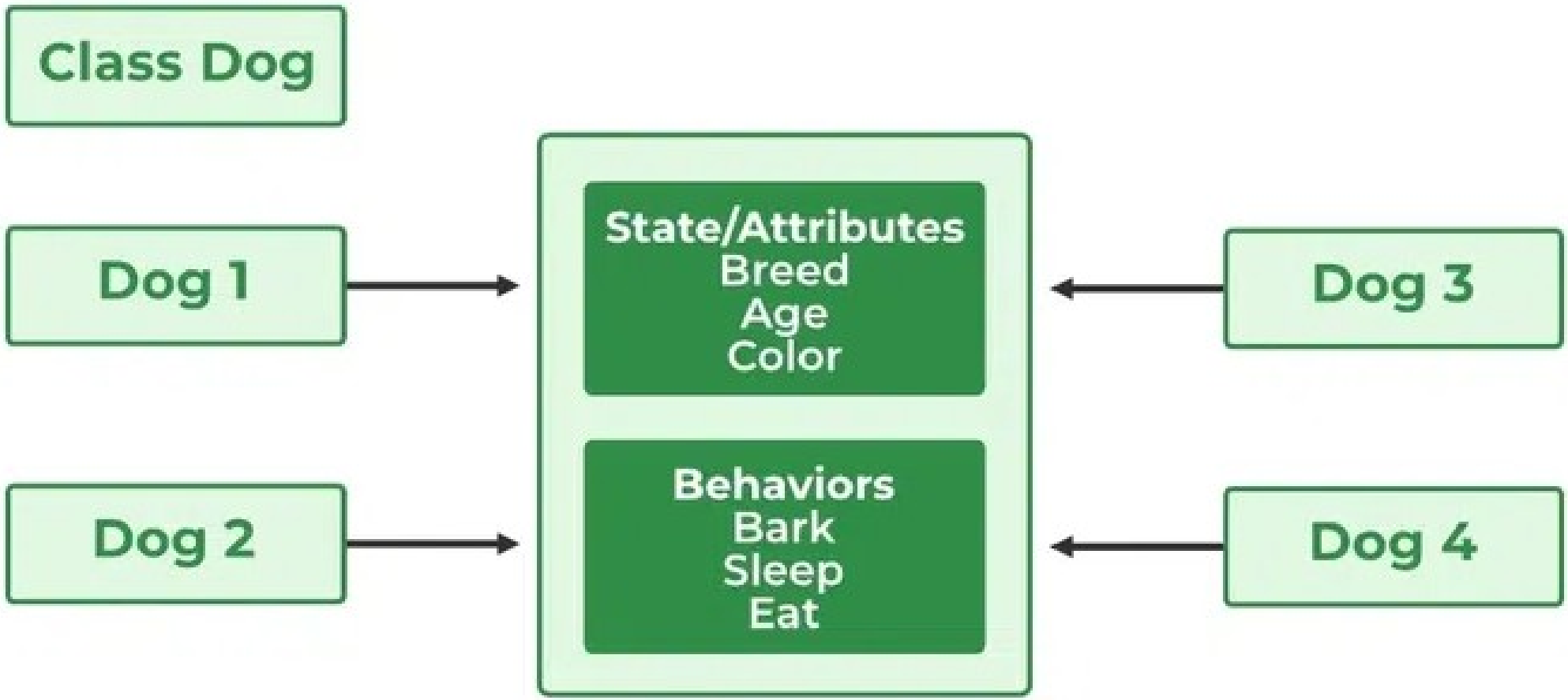
- Syntax to Create a Java Object:
  - Class_name object_name = new Class_name([parameters]);
  - Parameters are optional and can be used while you're using constructors in the class.
- Example to Create a Java Object
  - // Creating a Java class
  - class Dog {
  - // Declaring and initializing the attributes
  - String breed;
  - int age;
  - String color;

  - // methods to set breed, age, and color of the dog
  - public void setBreed(String breed) {
  - this.breed = breed;
  - }
  - public void setAge(int age) {
  - this.age = age;
  - }
  - public void setColor(String color) {
  - this.color = color;
  - }
  - // method to print all three values
  - public void printDetails() {
  - System.out.println("Dog detials:");
  - System.out.println(this.breed);
  - System.out.println(this.age);
  - System.out.println(this.color);
  - }
  - }

  - public class Main {
  - public static void main(String[] args) {
  - // Creating an object of the class Dog
  - Dog obj = new Dog();

  - // setting the attributes
  - obj.setBreed("Golden Retriever");
  - obj.setAge(2);
  - obj.setColor("Golden");

  - // Printing values
  - obj.printDetails();
  - }
  - }


Class Dog
Dog 1
Dog 2
State/Attributes
Breed
Age
Color
Behaviors
Bark
Sleep
Eat
Dog 3
Dog 4

Initializing a Java Object

- The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory.
- The new operator also invokes the class constructor.
- Example:
  - // Java Program to Demonstrate the
  - // use of a class with instance variable

  - // Class Declaration
  - public class Dog {
  -
  - // Instance Variables
  - String name;
  - String breed;
  - int age;
  - String color;

- // Constructor Declaration of Class
- public Dog(String name, String breed, int age,
- String color)
- {
- this.name = name;
- this.breed = breed;
- this.age = age;
- this.color = color;
- }

- // method 1
- public String getName() {
- return name;
- }

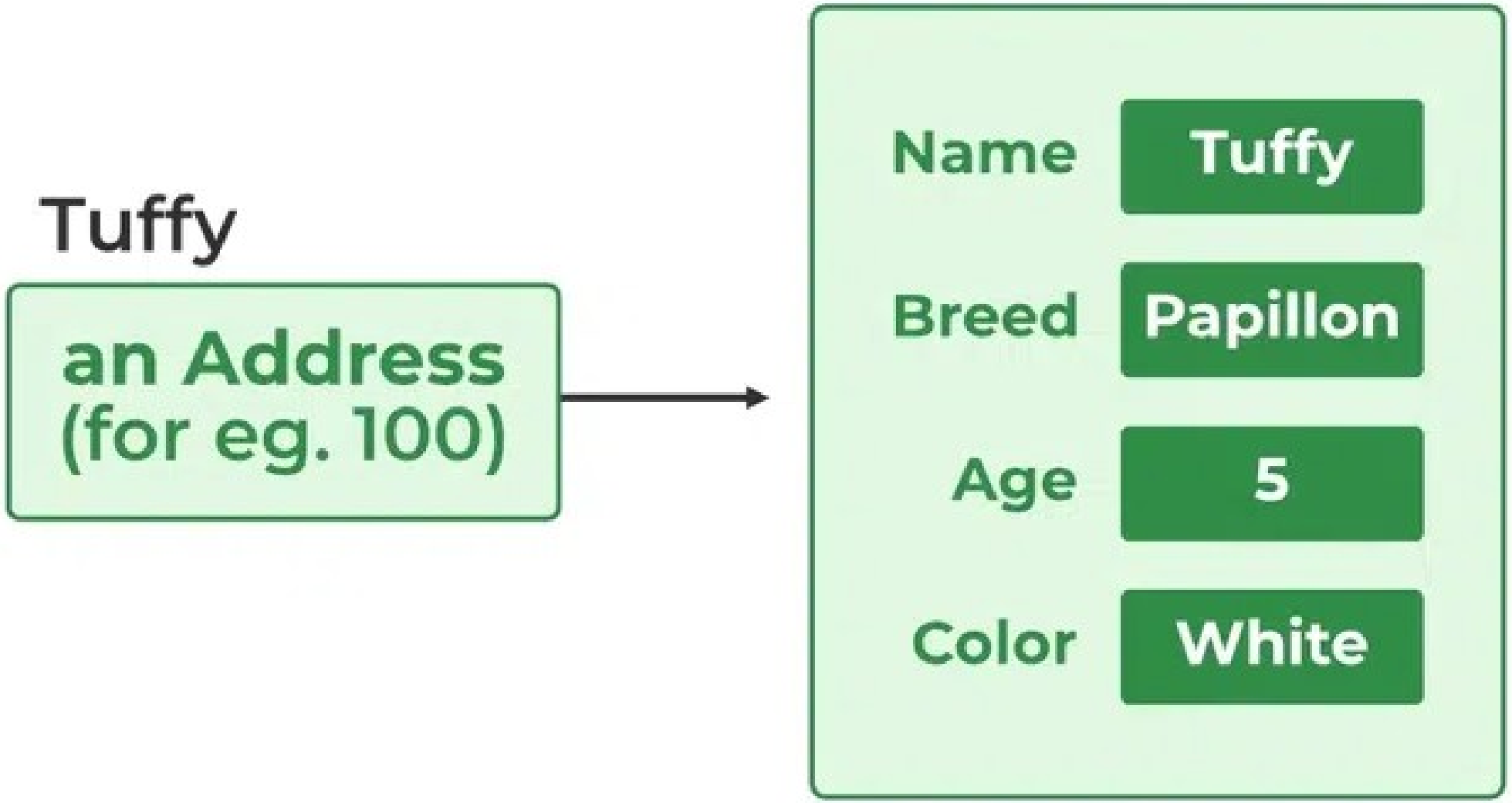
```
// method 2
    public String getBreed() {
      return breed;
    }

    // method 3
    public int getAge() {
      return age;
    }

    // method 4
    public String getColor() {
      return color;
    }
```

```
 @Override public String toString()
    {
        return ("Name is: " + this.getName()
                + "\nBreed, age, and color are: "
                + this.getBreed() + "," + this.getAge()
                + "," + this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy
            = new Dog("tuffy", "papillon", 5,
"white");
        System.out.println(tuffy.toString());
    }
}
```


Tuffy
an Address
(for eg. 100)
Name
Tuffy
Breed
Papillon
Age
5
Color
White

Initialize Object by using Method/Function

```
// Java Program to initialize Java Object
// by using method/function
public class Main{

    static String name;
    static float price;

    static void set(String n, float p) {
        name = n;
        price = p;
    }

 static void get()
    {
        System.out.println("Software name is: " + name);
        System.out.println("Software price is: " + price);
    }

    public static void main(String args[])
    {
        Main.set("Visual studio", 0.0f);
        Main.get();
    }
}
```

Ways to Create an Object of a Class

1. **Using new Keyword**
   - It is the most common and general way to create an object in Java.
   - **Example:** *Test t = new Test();*
2. **Using Class.forName(String className) Method**
   - There is a pre-defined class in java.lang package with name Class.
   - The forName(String className) method returns the Class object associated with the class with the given string name.
   - We have to give a fully qualified name for a class.
   - On calling the new Instance() method on this Class object returns a new instance of the class with the given string name.
   - Example:
     - // creating object of public class Test
     - // consider class Test present in com.p1 package
     - Test obj = (Test)Class.forName(“com.p1.Test”).newInstance();

3. Using clone() method

- The clone() method is present in the Object class.
- It creates and returns a copy of the object.
- Syntax:
  - // creating object of class Test
  - Test t1 = new Test();
  - // creating clone of above object
  - Test t2 = (Test)t1.clone();
- Example:
  - // Creation of Object
  - // Using clone() method

  - // Main class
  - // Implementing Cloneable interface
  - class Main implements Cloneable {

    - // Method 1
    - @Override
    - protected Object clone()
    - throws CloneNotSupportedException
    - {
    - // Super() keyword refers to parent class
    - return super.clone();
    - }
    - String name = “Hello World";
    -
  - // Method 2
    - // main driver method
    - public static void main(String[] args)
    - {
    - Main o1 = new Main();
    -
    - // Try block to check for exceptions
    - try {
    - Main o2 = (Main)o1.clone();
    - System.out.println(o2.name);
    - }
    - catch (CloneNotSupportedException e) {
    - e.printStackTrace();
    - }
    - }
  - }

Multiple Objects

- We can create multiple objects of one class:
- Example:
  - Create two objects of Main:
    - public class Main {
    - int x = 5;

    - public static void main(String[] args) {
    - Main myObj1 = new Main(); // Object 1
    - Main myObj2 = new Main(); // Object 2
    - System.out.println(myObj1.x);
    - System.out.println(myObj2.x);
    - }
    - }

Using Multiple Classes

- We can also create an object of a class and access it in another class.
- This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the main() method (code to be executed)).
- The name of the java file should match the class name.
- In this example, we have created two files in the same directory/folder:
  - Main.java
  - Second.java

- Main.java
  - public class Main {
  - int x = 5;
  - }

- Second.java
  - class Second {
  - public static void main(String[] args) {
  - Main myObj = new Main();
  - System.out.println(myObj.x);
  - }
  - }

- Compile:
  - C:\Users*Your Name*>javac Main.java
  - C:\Users*Your Name*>javac Second.java

- Run:
  - C:\Users*Your Name*>java Second

Anonymous Objects in Java

- Anonymous objects are objects that are instantiated but are not stored in a reference variable.
  - They are used for immediate method calls.
  - They will be destroyed after method calling.
  - They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).
  - In the example below, when a key button(referred to by the btn) is pressed, we are simply creating an anonymous object of EventHandler class for just calling the handle method.

- *btn.setOnAction(new EventHandler()*

```
{
    public void handle(ActionEvent event)
    {
        System.out.println("Hello World!");
    }
});
```

Difference Between Java Classes and Objects

Class	Object
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a **car**.	Objects of the class **car** can be BMW, Mercedes, Ferrari, etc.

Java Class Attributes

- Class attributes are variables within a class.
- Example:
  - Create a class called "Main" with two attributes: x and y:
    - public class Main {
    - int x = 5;
    - int y = 3;
    - }
- Accessing Attributes
  - You can access attributes by creating an object of the class, and by using the dot syntax (.)
  - Example: Create an object called "myObj" and print the value of x:
    - public class Main {
    - int x = 5;

    - public static void main(String[] args) {
    - Main myObj = new Main();
    - System.out.println(myObj.x);
    - }
    - }

Modify Attributes

- Example: Set the value of x to 40:
  - public class Main {
  - int x;
  - public static void main(String[] args) {
  - Main myObj = new Main();
  - myObj.x = 40;
  - System.out.println(myObj.x);
  - }
  - }
- Or override existing values: Change the value of x to 25:
  - public class Main {
  - int x = 10;

  - public static void main(String[] args) {
  - Main myObj = new Main();
  - myObj.x = 25; // x is now 25
  - System.out.println(myObj.x);
  - }
  - }

- If we don't want the ability to override existing values, declare the attribute as final:
- Example:
  - public class Main {
  - final int x = 10;

  - public static void main(String[] args) {
  - Main myObj = new Main();
  - myObj.x = 25; // will generate an error: cannot assign a value to a final variable
  - System.out.println(myObj.x);
  - }
  - }

Java Class Methods

- Methods are declared within a class, and that they are used to perform certain actions:
- Example: Create a method named myMethod() in Main:
  - public class Main {
  - static void myMethod() {
  - System.out.println("Hello World!");
  - }
  - }
- myMethod() prints a text (the action), when it is called. To call a method, write the method's name followed by two parentheses () and a semicolon;

- Example: Inside main, call myMethod():
  - public class Main {
  - static void myMethod() {
  - System.out.println("Hello World!");
  - }

  - public static void main(String[] args) {
  - myMethod();
  - }
  - }

  - // Outputs "Hello World!"

- Static vs. Public
  - We created a static method, which means that it can be accessed without creating an object of the class, unlike public, which can only be accessed by objects:
  - Example:
    - public class Main {
    - // Static method
    - static void myStaticMethod() {
    - System.out.println("Static methods can be called without creating objects");
    - }
    - // Public method
    - public void myPublicMethod() {
    - System.out.println("Public methods must be called by creating objects");
    - }
    -
    - // Main method
    - public static void main(String[] args) {
    - myStaticMethod(); // Call the static method
    - // myPublicMethod(); This would compile an error
    - Main myObj = new Main(); // Create an object of Main
    - myObj.myPublicMethod(); // Call the public method on the object
    - }
    - }

Access Methods With an Object

- Example:
  - // Create a Main class
  - public class Main {
  -
  - // Create a fullThrottle() method
  - public void fullThrottle() {
  - System.out.println("The car is going as fast as it can!");
  - }

  - // Create a speed() method and add a parameter
  - public void speed(int maxSpeed) {
  - System.out.println("Max speed is: " + maxSpeed);
  - }

- // Inside main, call the methods on the myCar object
- public static void main(String[] args) {
- Main myCar = new Main(); // Create a myCar object
- myCar.fullThrottle(); // Call the fullThrottle() method
- myCar.speed(200); // Call the speed() method
- }
- }

- // The car is going as fast as it can!
- // Max speed is: 200

- Explanation:
  1. We created a custom Main class with the class keyword.
  2. We created the fullThrottle() and speed() methods in the Main class.
  3. The fullThrottle() method and the speed() method will print out some text, when they are called.
  4. The speed() method accepts an int parameter called maxSpeed - we will use this in 8).
  5. In order to use the Main class and its methods, we need to create an object of the Main Class.
  6. Then, go to the main() method, which you know by now is a built-in Java method that runs your program (any code inside main is executed).
  7. By using the new keyword we created an object with the name myCar.
  8. Then, we call the fullThrottle() and speed() methods on the myCar object, and run the program using the name of the object (myCar), followed by a dot (.), followed by the name of the method (fullThrottle(); and speed(200);). Notice that we add an int parameter of 200 inside the speed() method.

Java Constructors

- 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:
- Example:
  - // Create a Main class
  - public class Main {
  - int x; // Create a class attribute

  - // Create a class constructor for the Main class
  - public Main() {
  - x = 5; // Set the initial value for the class attribute x
  - }

  - public static void main(String[] args) {
  - Main myObj = new Main(); // Create an object of class Main (This will call the constructor)
  - System.out.println(myObj.x); // Print the value of x
  - }
  - }

Constructor Parameters

- Constructors can also take parameters, which is used to initialize attributes.
- The following example adds an int y parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:
- Example:
  - public class Main {
  - int x;

  - public Main(int y) {
  - x = y;
  - }

  - public static void main(String[] args) {
  - Main myObj = new Main(5);
  - System.out.println(myObj.x);
  - }
  - }

- How Java Constructors are Different From Java Methods?
  - Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
  - Constructors do not return any type while method(s) have the return type or void if does not return any value.
  - Constructors are called only once at the time of Object creation while method(s) can be called any number of times.
  - Example:
    - class Geek
    - {
    - .....
    - // A Constructor
    - Geek() {
    - }
    - .....
    - }
    - // We can create an object of the above class
    - // using the below statement. This statement
    - // calls above constructor.
    - Geek obj = new Geek();

- The first line of a constructor is a call to super() or this(), (a call to a constructor of a super-class or an overloaded constructor), if you don’t type in the call to super in your constructor the compiler will provide you with a non-argument call to super at the first line of your code, the super constructor must be called to create an object.
- **Types of Constructors in Java:** Primarily there are three types of constructors in Java.
  - Default Constructor
  - Parameterized Constructor
  - Copy Constructor

- Default Constructor in Java
  - A constructor that has no parameters is known as default constructor.
  - A default constructor is invisible.
  - And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out.
  - It is being overloaded and called a parameterized constructor.
  - The default constructor changed into the parameterized constructor. But Parameterized constructor can’t change the default constructor.
  - The default constructor can be implicit or explicit.
    - Implicit Default Constructor: If no constructor is defined in a class, the Java compiler automatically provides a default constructor. This constructor doesn’t take any parameters and initializes the object with default values, such as 0 for numbers, null for objects.
    - Explicit Default Constructor: If we define a constructor that takes no parameters, it’s called an explicit default constructor. This constructor replaces the one the compiler would normally create automatically. Once you define any constructor (with or without parameters), the compiler no longer provides the default constructor for you.

- Example:
  - // Java Program to demonstrate
  - // Default Constructor
  - import java.io.*;

  - // Driver class
  - class GFG {

  - // Default Constructor
  - GFG() { System.out.println("Default constructor"); }

  - // Driver function
  - public static void main(String[] args)
  - {
  - GFG hello = new GFG();
  - }
  - }

- Parameterized Constructor in Java
  - A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.
  - Example:

```
// Java Program for Parameterized Constructor
import java.io.*;


class Test {
    // data members of the class.
    String name;
    int id;

    Test(String name, int id) {
        this.name = name;
        this.id = id;
    }
}


class GFG
{
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        Test test1= new Test("Avinash", 68);
        System.out.println("Name :" + test1.name
                           + " and Id :" + test1.id);
    }
}
```

- Copy Constructor in Java
  - Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.
  - Example:

```
// Java Program for Copy Constructor
import java.io.*;


class Test {
    // data members of the class.
    String name;
    int id;


    // Parameterized Constructor
    Test(String name, int id)
    {
        this.name = name;
        this.id = id;
    }


    // Copy Constructor
    Test(Test obj2)
    {
        this.name = obj2.name;
        this.id = obj2.id;
    }
}
```

```
class GFG {
    public static void main(String[] args)
    {
        // This would invoke the parameterized constructor.
        System.out.println("First Object");
        Test test1 = new Test("Avinash", 68);
        System.out.println("Name :" + test1.name
                           + " and Id :" + test1.id);


        System.out.println();


        // This would invoke the copy constructor.
        Test test2 = new Test(test1);
        System.out.println(
            "Copy Constructor used Second Object");
        System.out.println("Name :" + test2.name
                           + " and Id :" + test2.id);
    }
}
```

Garbage Collection in Java

- Garbage collection in Java is an automatic memory management process that helps Java programs run efficiently.
- Java programs compile to bytecode that can be run on a Java Virtual Machine (JVM).
- When Java programs run on the JVM, objects in 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.
- In Java, this process is automated. The garbage collector automatically finds and removes objects that are no longer needed, freeing up memory in the heap.
- It runs in the background as a daemon thread, helping to manage memory efficiently without requiring the programmer’s constant attention.

Working of Garbage Collection

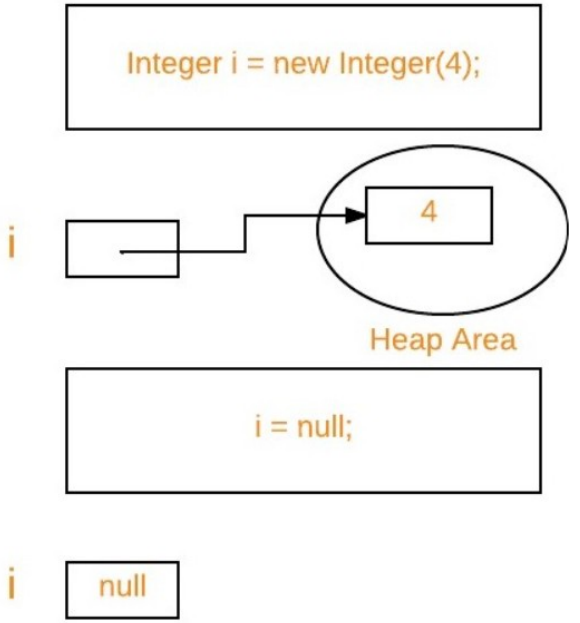
- Java garbage collection is an automatic process that manages memory in the heap.
- It identifies which objects are still in use (referenced) and which are not in use (unreferenced).
- Unreferenced objects can be deleted to free up memory.
- The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Types of Activities in Java Garbage Collection

- Two types of garbage collection activity usually happen in Java. These are:
  - Minor or incremental Garbage Collection: This occurs when unreachable objects in the Young Generation heap memory are removed.
  - Major or Full Garbage Collection: This happens when objects that survived minor garbage collection are removed from the Old Generation heap memory. It occurs less frequently than minor garbage collection.

Important Concepts on Garbage Collection

1. Unreachable Objects
   - An object is said to be unreachable if it doesn't contain any reference to it.
   - Note: Objects which are part of the island of isolation are also unreachable.
   - Example:
     - Integer i = new Integer(4);
     - // the new Integer object is reachable via the referen
     - i = null;
     - // the Integer object is no longer reachable.



Eligibility for Garbage Collection

- An object is said to be eligible for garbage collection if it is unreachable.
- After i = null, integer object 4 in the heap area is suitable for garbage collection in the above image.
- How to Make an Object Eligible for Garbage Collection?
  - Even though the programmer is not responsible for destroying useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
  - There are generally four ways to make an object eligible for garbage collection.

- Nullifying the reference variable: Set the reference to null.
- Re-assigning the reference variable: Assign a new object to the reference.
- An object created inside the method: Objects created within a method are eligible for garbage collection once the method completes.
- Island of Isolation: Objects that are isolated and not referenced by any reachable objects.
- **Ways to Request JVM to Run Garbage Collection**
  - Once an object is eligible for garbage collection, it may not be destroyed immediately. The garbage collector runs at the JVM's discretion, and you cannot predict when it will occur.
  - We can also request JVM to run Garbage Collector. There are two ways to do it :
    - Using System.gc(): This static method requests the JVM to perform garbage collection.
    - Using Runtime.getRuntime().gc(): This method also requests garbage collection through the Runtime class.

2. **Finalization**

- Before destroying an object, the garbage collector calls the finalize() method to perform cleanup activities.
- The method is defined in the Object class as follows:
  - *protected void finalize() throws Throwable*
- The finalize() method is called by Garbage Collector, not JVM.
- The default implementation of finalize() is empty, so overriding it is recommended for resource cleanup.
- The finalize() method is called only once per object.
- If an uncaught exception is thrown by the finalize() method, the exception is ignored, and the finalization of that object terminates.

Threads in Garbage Collection

1. **Controlling Garbage Collection Threads**
   - To control the number of threads used by the garbage collector, we can use multi-threaded collectors like the Parallel Garbage Collector.
2. **Limiting GC Pause Time**
   - To limit the maximum pause time during garbage collection, we can use the MaxGCPauseMillis option

Types of Garbage Collectors

1. **Serial Garbage Collector**
   - Serial Garbage Collector is the simplest type of garbage collector.
   - It uses a single thread to handle all garbage collection tasks.
   - When garbage collection is triggered, it pauses all running application threads.
   - Serial Garbage Collector is used for small applications or embedded systems where memory is limited.
2. **Parallel Garbage Collector**
   - Parallel Garbage Collector is the default garbage collector in Java 8.
   - It is also known as the throughput collector
   - It uses multiple threads to perform garbage collection tasks.
   - It is designed to maximize performance by making full use of available CPU resources.
   - Parallel Garbage Collector is suitable for applications that need high performance and can tolerate some pauses, such as batch processing systems.
   - Parallel Garbage Collector is better than the Serial Garbage Collector

3. Concurrent Mark-Sweep Collector

- Concurrent Mark-Sweep Collector is designed to minimize pause times by doing most of its work while the application is still running.
- It uses multiple threads to mark and clean up unreachable objects.
- As we know, Serial garbage collector and Parallel garbage collector freeze the running threads of the application while performing the garbage collection. But CMS Garbage collector will perform freezing of running threads i.e. application pause in two cases only:
  - While performing the garbage collection, If there is a change in heap memory in parallel.
  - While marking the referenced objects in the old generation space.
- Concurrent Mark-Sweep Collector is used for applications that require low latency, such as web servers and interactive applications.

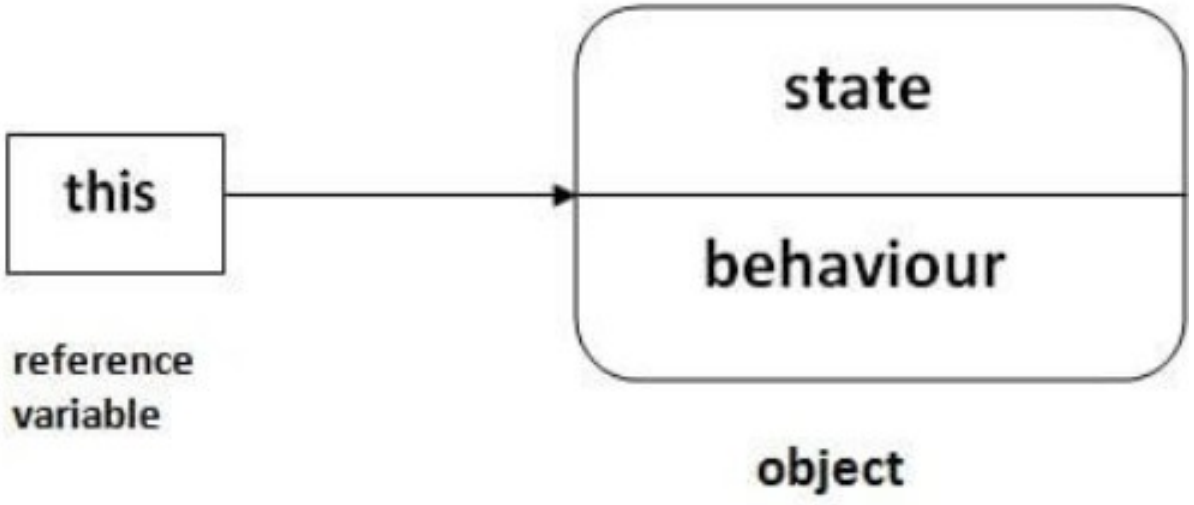
4. G1 Garbage Collector

- G1 Garbage collector was introduced in JDK 7 and became the default garbage collector in Java 9.
- Designed for applications with large heap sizes (greater than 4GB).
- Divides the heap into multiple equal-sized regions (1MB to 32MB).
- G1 Garbage collector replaced the CMS collector due to better performance efficiency.
- Marks regions with active objects and prioritizes collection in regions with the most unused objects, hence the name “Garbage-First” (G1).
- Compacts free heap space after garbage collection, improving memory efficiency.

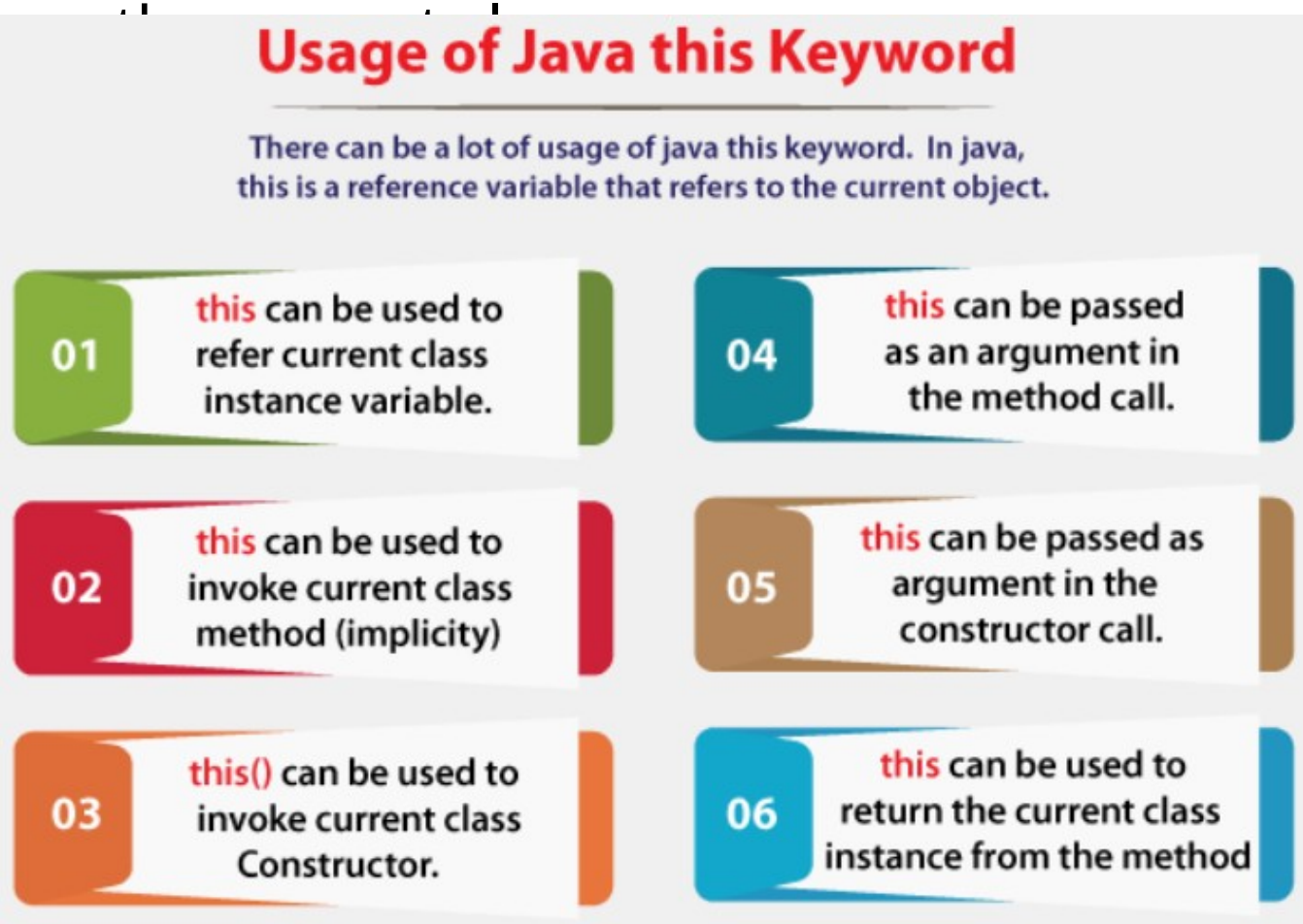
5. **Z Garbage Collector:**
   - ZGC is a low-latency garbage collector introduced in Java 11.
   - It is designed to handle large heaps with minimal pause times, typically in the range of milliseconds.
6. **Shenandoah Garbage Collector**
   - Shenandoah is another low-pause-time garbage collector introduced in Java 12.
   - It aims to reduce pause times by performing garbage collection concurrently with application threads.

this Keyword

- The this keyword refers to the current object in a method or constructor.
- The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name.



- this can also be used to:
  - Invoke current class constructor
  - Invoke current class method
  - Ret
    obj
  - Pass an argument in the method call
  - Pass an argument in the constructor call



- Understanding the problem without this keyword:

```
class Student{
        int rollno;
        String name;
        float fee;

Student(int rollno,String name,float fee){
        rollno=rollno;     name=name;
        fee=fee;
        }
        void display(){

System.out.println(rollno+" "+name+" "+fee
);
        }
}
```

```
class TestThis1{
        public static void main(String args[]){

Student s1=new Student(111,"ankit",5000f);

Student s2=new Student(112,"sumit",6000f);

        s1.display();
        s2.display();
        }
}
```

- **Output:**

```
0 null 0.0
0 null 0.0
```

- Solution of the above problem by this keyword

```
class Student{
	int rollno;
	String name;
	float fee;
	Student(int rollno,String name,float fee){
	this.rollno=rollno;
	this.name=name;
	this.fee=fee;
	}
	void display(){
	System.out.println(rollno+" "+name+" "+fee);
	}
}
```

```
class TestThis2{
	public static void main(String args[]){
	Student s1=new Student(111,"ankit",5000f);
	Student s2=new Student(112,"sumit",6000f);
	s1.display();
	s2.display();
	}
}
```

- Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

- Program where this keyword is not required

```
class Student{
	int rollno;
	String name;
	float fee;
	Student(int r,String n,float f){
	rollno=r;
	name=n;
	fee=f;
	}
	void display(){
	System.out.println(rollno+" "+name+" "+fee);
	}
}
```

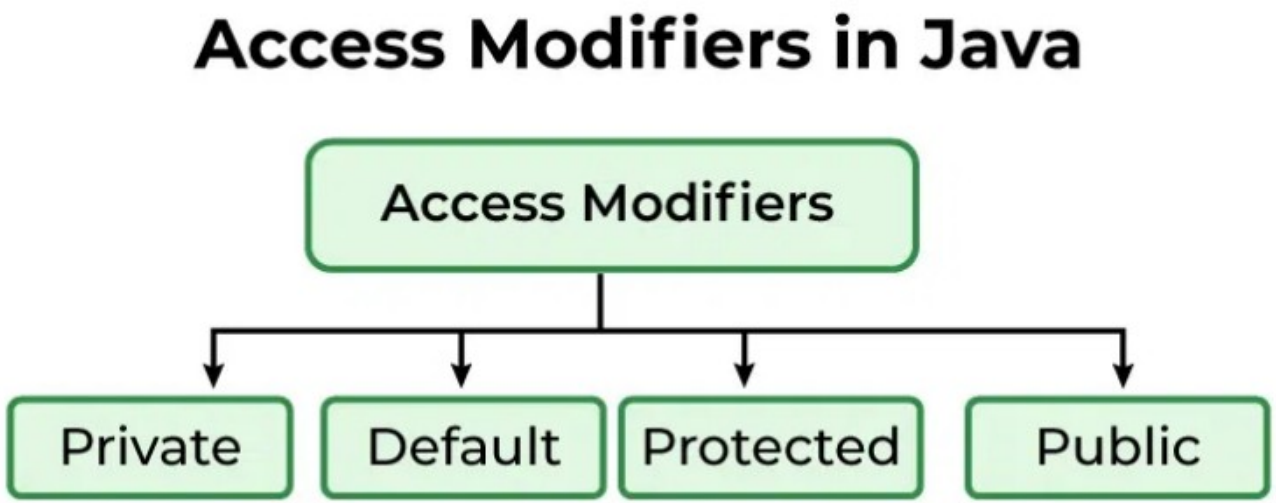
```
class TestThis3{
	public static void main(String args[]){
		Student s1=new Student(111,"ankit",5000f);
		Student s2=new Student(112,"sumit",6000f);
		s1.display();
		s2.display();
	}
}
```

- Output:

```
111 ankit 5000.0
112 sumit 6000.0
```

Access Modifiers in Java

- In Java, Access modifiers helps to restrict the scope of a class, constructor, variable, method, or data member.
- It provides security, accessibility, etc. to the user depending upon the access modifier used with the element.
- Types of Access Modifiers:
    - Default – No keyword required
    - Private
    - Protected
    - Public



- Default Access Modifier
  - When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default.
  - The default access modifiers are accessible only within the same package.

```
// error while using class from different
// package with default modifier
package p2;
import p1.*;    // importing package p1

// This class is having
// default access modifier
class GeekNew {
    public static void main(String args[]) {

        // Accessing class Geek from package p1
        Geek o = new Geek();

        o.display();
    }
}
```

- Default Access Modifier
  - When no access modifier is specified for a class, method, or data member, it is said to be having the default access modifier by default.
  - The default access modifiers are accessible only within the same package.

```
// error while using class from different
// package with default modifier
package p2;
import p1.*;    // importing package p1

// This class is having
// default access modifier
class GeekNew {
    public static void main(String args[]) {

        // Accessing class Geek from package p1
        Geek o = new Geek();

        o.display();
    }
}
```

- Private Access Modifier
  - The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.
  - Any other class of the same package will not be able to access these members.
  - Top-level classes or interfaces can not be declared as private because,
    - private means “only visible within the enclosing class“.
    - protected means “only visible within the enclosing class and any subclasses“.
  - Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes.

- Example:
  - // error while using class from different package with
  - // private access modifier
  - package p1;

  - // Class A
  - class A {
  - private void display() {
  - System.out.println(“Hello World");
  - }
  - }

  - // Class B
  - class B {
  - public static void main(String args[]) {
  - A obj = new A();
  -
  - // Trying to access private method
  - // of another class
  - obj.display();
  - }
  - }

- Protected Access Modifier
  - The protected access modifier is specified using the keyword protected.
  - The methods or data members declared as protected are accessible within the same package or subclasses in different packages.
  - Example:

```
// protected access modifier
package p1;

// Class A
public class A {
    protected void display() {
        System.out.println("GeeksforGeeks");
    }
}
```

  - So, it demonstrates that a protected method is accessible within the same package.

- Public Access Modifier
  - The public access modifier is specified using the keyword public.
    - The public access modifier has the widest scope among all other access modifiers.
    - Classes, methods, or data members that are declared as public are accessible from everywhere in the program. There is no restriction on the scope of public data members.

  - Example:

```
// public modifier
package p1;

public class A {

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

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Public Access Modifier

- Algorithm to Use Access Modifier in Java
  - Define a class: Create a class to represent the object you want to manage.
  - Define instance variables: Inside the class, define variables for the data you want to manage.
  - Set an access modifier:
    - Use private for variables only accessible within the class.
    - Use protected for variables accessible within the class and its subclasses.
    - Use public for variables accessible from anywhere.
  - Use getter and setter methods: To access or modify variables, use getter (accessor) and setter (mutator) methods, even for public variables, to maintain encapsulation.