4.1 Defining Class

The Concept of Java Classes and Objects

Classes and objects are the two most essential Java concepts that every programmer must learn. Classes and objects are closely related and work together. An object has behaviors and states, and is an instance of class. For instance, a cat is an object—it's color and size are states, and its meowing and clawing furniture are behaviors. A class models the object, a blueprint or template that describes the state or behavior supported by objects of that type.

We can define a class as a container that stores the data members and methods together. These data members and methods are common to all the objects present in a particular package.
Every class we use in Java consists of the following components, as described below:

Class Name

This describes the name given to the class that the programmer decides on, according to the predefined naming conventions.

Body of the Class

In Java, the body of a class contains the class members, including fields (variables), methods, and other components that define the behavior and characteristics of objects created from that class. Here's a brief overview:

1. **Fields (Variables):**
   - Fields represent the attributes or properties of objects created from the class.
   - They can be of various data types (e.g., int, String) and may have different access modifiers (e.g., public, private).

```
java
  public class MyClass {
     // Fields
     private int age;
     private String name;

     // ... other class members
  }
```

2. **Methods:**

- Methods define the behavior of the class. They represent actions or operations that objects of the class can perform.
- Methods can take parameters, return values, and may have different access modifiers.

```
public class MyClass {
   // Fields

   // Methods
```

```
    public void setName(String newName) {
        name = newName;
    }

    public String getName() {
        return name;
    }

    // ... other class members
}
```

Type of Classes
In Java, we classify classes into two types:

- Built-in Classes
- User-defined Classes

Built-in Classes
Built-in classes are just the predefined classes that come along with the Java Development Kit (JDK). We also call these built-in classes libraries. Some examples of built-in classes include:

- java.lang.System
- java.util.Date
- java.util.ArrayList
- java.lang.Thread

Library: collection of packages
Package: contains several classes
class: contains several methods
Method: a set of instructions

Using Predefined Classes and Methods
To use a method, you must know:

- Name of the class containing the method (like **Math**)
- Name of th
- e package containing the class (like **java.lang**)
- Name of the method (like **pow**) and list of parameters

```
import java.lang.*; // imports package
Math.pow( 2, 3 ); // calls power method in class Math
Math.pow( x, y ); // another call
```

Following are some invalid variable declarations in Java:

```
boolean break = false; // not allowed as break is keyword
int boolean = 8; // not allowed as boolean is keyword
boolean goto = false; // not allowed as goto is keyword
String final = "hi"; // not allowed as final is keyword
```

Using predefined class name as User defined class name
Question : Can we have our class name as one of the predefined class name in our program?
Answer : Yes we can have it. Below is example of using **Number** as user defined class

```
// Number is predefined class name in java.lang package
// Note : java.lang package is included in every java
program by default
public class Number
{
        public static void main (String[] args)
        {
                System.out.println("It works");
        }
}
```

Output:

```
It works
```

Using String as User Defined Class:

```
// String is predefined class name in java.lang package
// Note : java.lang package is included in every java program by default

public class String
{
        public static void main (String[] args)
        {
                System.out.println("I got confused");
        }
}
```

4.1 Defining Class

User-defined classes are rather self-explanatory. The name says it all. They are classes that the user defines and
manipulates in the real-time programming environment. User-defined classes are broken down into three types:

Concrete Class

Concrete class is just another standard class that the user defines and stores the methods and data members in.

Syntax:

```
class con{
        //class body;
}
```

Rules for Creating Classes

The following rules are mandatory when you're working with Java classes:

1. The keyword "class" must be used to declare a class
2. Every class name should start with an upper case character, and if you intend to include multiple words in a class name, make sure you use the camel case
3. A Java project can contain any number of default classes but should not hold more than one public class
4. You should not use special characters when naming classes
5. You can implement multiple interfaces by writing their names in front of the class, separated by commas
6. You should expect a Java class to inherit only one parent class

Key Differences Between Java Classes and Objects

The key differences between a class and an object are:
Class:
A class is a blueprint for creating objects
A class is a logical entity
The keyword used is "class"
A class is designed or declared only once
The computer does not allocate memory when you declare a class
Objects:
An object is a copy of a class
An object is a physical entity
The keyword used is "new"
You can create any number of objects using one single class
The computer allocates memory when you declare a class

Create a Class

To create a class, use the keyword class:

Main.java
Create a class named "Main" with a variable x:
```
public class Main {
int x = 5;
}
```

4.3 Adding Method

In Java, you can add a new method to a class by declaring a method within the class. Here's a general outline of how you can add a method to a class in Java:

```
public class MyClass {
    // Fields (attributes) of the class

    // Constructors if needed

    // Existing methods of the class

    // New method declaration
    // accessModifier returnType methodName(parameters) {
    //     // Method body (implementation)
    //     // Optionally, return a value if the returnType is not void
    // }

    // Other members of the class
}
```

Let's break down the components:

1. **Access Modifier:** This defines the visibility of the method. Common access modifiers include public, private, and protected.
2. **Return Type:** This specifies the type of the value that the method returns. If the method doesn't return any value, you use void.
3. **Method Name:** This is the name of the method. It should follow Java naming conventions.
4. **Parameters:** These are input values that the method expects. Parameters are optional; you can have methods without parameters.
5. **Method Body:** This is enclosed within curly braces {}. It contains the actual implementation of the method.

Here's an example illustrating the addition of a new method to a class:

```
java
public class MyClass {
    private int number;

    // Constructor
    public MyClass(int number) {
        this.number = number;
    }

    // Existing method
    public void displayNumber() {
        System.out.println("Number: " + number);
    }

    // New method
    public void multiplyByTwo() {
        number *= 2;
    }

    public static void main(String[] args) {
        MyClass myObject = new MyClass(5);

        // Using existing method
        myObject.displayNumber(); // Output: Number: 5

        // Using new method
        myObject.multiplyByTwo();

        // Displaying the updated number
        myObject.displayNumber(); // Output: Number: 10
    }
}
```

In this example, multiplyByTwo is a new method added to the MyClass. It multiplies the number field by 2 when called. The main method demonstrates how to use both the existing displayNumber method and the newly added multiplyByTwo method.

4.4 Static Variables, Method, Blocks and Class

Java static variable

If you declare any variable as static, it is known as a static variable.

- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

Advantages of static variable

It makes your program **memory efficient** (i.e., it saves memory).

Example of static variable

```
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value
 Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
        //creating objects
        Counter2 c1=new Counter2();
        Counter2 c2=new Counter2();
        Counter2 c3=new Counter2();
        }
}
```

Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

Example of static method

```
        //Java Program to demonstrate the use of a static method.
        class Student{
            int rollno;
            String name;
            static String college = "ITS";
            //static method to change the value of static variable
            static void change(){
            college = "BBDIT";
            }
            //constructor to initialize the variable
            Student(int r, String n){
```

```
        rollno = r;
        name = n;
        }
        //method to display values
        void display(){System.out.println(rollno+" "+name+" "+college);}
    }

    //Test class to create and display the values of object
    public class TestStaticMethod{
        public static void main(String args[]){
        Student.change();//calling change method
        //creating objects
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        Student s3 = new Student(333,"Sonoo");
        //calling display method
        s1.display();
        s2.display();
        s3.display();
        }
    }
```

Output:

```
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT
```

Another example of a static method that performs a normal calculation

//Java Program to get the cube of a given number using the static method

```
    class Calculate{
      static int cube(int x){
      return x*x*x;
      }

      public static void main(String args[]){
      int result=Calculate.cube(5);
      System.out.println(result);
      }
    }
```

Output:

```
125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

4.5 Access Control

Access control is a mechanism, an attribute of encapsulation which restricts the access of certain members of a class to specific parts of a program. Access to members of a class can be controlled using the *access modifiers*. There are four access modifiers in Java. They are:

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

If the member (variable or method) is not marked as either *public* or *protected* or *private*, the access modifier for that member will be *default*. We can apply access modifiers to classes also.

Among the four access modifiers, *private* is the most restrictive access modifier and *public* is the least restrictive access modifier. Syntax for declaring a access modifier is shown below:

access-modifier data-type variable-name;

Example for declaring a *private* integer variable is shown below:

private int side;

In a similar way we can apply access modifiers to methods or classes although *private* classes are less common.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

The private access modifier is accessible only within the class.

Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
        private int data=40;
        private void msg(){System.out.println("Hello java");}
}
```

```
public class Simple{
        public static void main(String args[]){
          A obj=new A();
          System.out.println(obj.data);//Compile Time Error
          obj.msg();//Compile Time Error
  }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
class A{
        private A(){}//private constructor
        void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
        A obj=new A();//Compile Time Error
   }
}
```

Note: A class cannot be private or protected except nested class.

---

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
        void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
```

```
class B{
  public static void main(String args[]){
            A obj = new A();//Compile Time Error
             obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
//save by A.java
package pack;
public class A{
          protected void msg(){System.out.println("Hello");}
}



//save by B.java
package mypack;
import pack.*;
  class B extends A{
             public static void main(String args[]){
                        B obj = new B();
                        obj.msg();
  }
}
```

Output:

```
Hello
```

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
//save by A.java

package pack;
public class A{
        public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

class B{
          public static void main(String args[]){
                   A obj = new A();
                  obj.msg();
        }
}
```

Output:
```
Hello
```

4.6 Method Parameters

A method is a block of code which only runs when it is called.
You can pass data, known as parameters, into a method.
Methods are used to perform certain actions, and they are also known as functions.
Why use methods? To reuse code: define the code once, and use it many times.

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.
Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.
An argument is the actual value that is passed to a method when it is invoked. It corresponds to the parameter defined in the method signature.
The following example has a method that takes a String called fname as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

Example

```
public class Main {
static void myMethod(String fname) {
System.out.println(fname + " Refsnes");
}
public static void main(String[] args) {
        myMethod("Liam");
        myMethod("Jenny");
        myMethod("Anja");
}
}
// Liam Refsnes
// Jenny Refsnes
// Anja Refsnes
```

Multiple Parameters

You can have as many parameters as you like:

Example

```
public class Main {
        static void myMethod(String fname, int age) {
        System.out.println(fname + " is " + age);
        }
        public static void main(String[] args) {
                myMethod("Liam", 5);
                myMethod("Jenny", 8);
                myMethod("Anja", 31);
        }
}
// Liam is 5
// Jenny is 8
// Anja is 31
```

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

Example

```
public class Main {
        static int myMethod(int x) {
                return 5 + x;
```

```
    }
    public static void main(String[] args) {
        System.out.println(myMethod(3));
    }
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's two parameters:

Example

```
public class Main {
static int myMethod(int x, int y) {
return x + y;
}
public static void main(String[] args) {
System.out.println(myMethod(5, 3));
}
}
// Outputs 8 (5 + 3)
```

You can also store the result in a variable (recommended, as it is easier to read and maintain):

Example

```
public class Main {
static int myMethod(int x, int y) {
return x + y;
}
public static void main(String[] args) {
int z = myMethod(5, 3);
System.out.println(z);
}
}
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use if...else statements inside methods:

Example

```
public class Main {
    // Create a checkAge() method with an integer variable called age
    static void checkAge(int age) {
    // If age is less than 18, print "access denied"
    if (age < 18) {
    System.out.println("Access denied - You are not old enough!");
    // If age is greater than, or equal to, 18, print "access granted"
```

```
        } else {
        System.out.println("Access granted - You are old enough!");
        }
        }
        public static void main(String[] args) {
        checkAge(20); // Call the checkAge method and pass along an age of 20
        }
}
// Outputs "Access granted - You are old enough!"
```

4.7 Creating Objects

In Java, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.

To create an object of MyClass, specify the class name, followed by the object name, and use the keyword new:

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);
        }
}
```

Multiple Objects

You 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

You 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)).

Remember that 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);
	}
}
```

When both files have been compiled:
```
C:\Users\Your Name>javac Main.java
C:\Users\Your Name>javac Second.java
```
Run the Second.java file:
```
C:\Users\Your Name>java Second
```
And the output will be:
```
5
```

Java Class Attributes
In the previous chapter, we used the term "variable" for x in the example (as shown below). It is actually an **attribute** of the class. Or you could say that 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 (.):
The following example will create an object of the Main class, with the name myObj. We use the x attribute on the
object to print its value:

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
You can also modify attribute values:

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:

Example
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 you 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);
    }
}
```

Multiple Objects
If you create multiple objects of one class, you can change the attribute values in one object, without affecting the
attribute values in the other:

Example
Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged:

```
public class Main {
    int x = 5;
    public static void main(String[] args) {
        Main myObj1 = new Main(); // Object 1
        Main myObj2 = new Main(); // Object 2
        myObj2.x = 25;
        System.out.println(myObj1.x); // Outputs 5
        System.out.println(myObj2.x); // Outputs 25
    }
}
```

Multiple Attributes
You can specify as many attributes as you want:

Example

```
public class Main {
    String fname = "John";
    String lname = "Doe";
    int age = 24;

    public static void main(String[] args) {
        Main myObj = new Main();
        System.out.println("Name: " + myObj.fname + " " + myObj.lname);
        System.out.println("Age: " + myObj.age);
    }
}
```

4.8 Accessing class members

In Java, you can access class members, including fields, methods, and nested classes, using the dot (.) operator. The ability to access these members depends on their access modifiers (e.g., public, private, protected) and the context in which you are trying to access them.

Here's a brief overview of how you can access different class members:

1. **Accessing Fields:**

   Fields are variables declared within a class. They can have different access modifiers like public, private, or protected.

   Example:

   ```
     public class MyClass {
       public int publicField;
       private int privateField;

       public void accessFields() {
               // Accessing public field
               int value = publicField;

               // Accessing private field within the same class
               int privateValue = privateField;
   ```

```
        }
    }
```

2. **Accessing Methods:**

   Methods are functions defined within a class. They can also have different access modifiers.

   Example:

```
public class MyClass {
    public void publicMethod() {
        // Code here
    }

    private void privateMethod() {
        // Code here
    }

    public void accessMethods() {
        // Accessing public method
        publicMethod();

        // Accessing private method within the same class
        privateMethod();
    }
}
```

4.9 Setters and Getters

In Java, setters and getters are methods used to set and retrieve the values of private fields in a class. They are part of the encapsulation concept, which helps in controlling access to the internal state of an object. Here's how you typically define setters and getters in Java:

```
public class MyClass {
    // Private fields
        private int myField;
        private String myStringField;

        // Setter method for 'myField'
        public void setMyField(int value) {
                this.myField = value;
        }

        // Getter method for 'myField'
        public int getMyField() {
                return this.myField;
        }

        // Setter method for 'myStringField'
        public void setMyStringField(String value) {
             this.myStringField = value;
        }

        // Getter method for 'myStringField'
        public String getMyStringField() {
            return this.myStringField;
        }
}
```

In the above example:

- The `setMyField` method is a setter for the `myField` variable, allowing you to set its value.
- The `getMyField` method is a getter for the `myField` variable, allowing you to retrieve its value.

This pattern helps in maintaining the principle of encapsulation, as it hides the internal details of the class and provides controlled access to its fields.

You can use these setters and getters to manipulate the state of an object while still controlling how the data is accessed and modified. Additionally, this approach allows you to implement validation or additional logic in the setters and getters if needed.

Benefits of Setters and Getters:

1. **Controlled Access:** Setters and getters allow you to control how external code interacts with the internal state of an object. You can validate input values, enforce constraints, or perform other logic before allowing changes.
2. **Flexibility:** By using getters, you can provide a flexible interface for accessing the state of an object. If the internal representation of the object changes, you can update the getter methods to maintain compatibility with existing code.
3. **Encapsulation:** Setters and getters contribute to encapsulation by keeping the internal details of a class hidden. This helps in managing complexity and improving the maintainability of the code.

In summary, setters and getters are fundamental to creating well-encapsulated and maintainable Java classes. They provide a controlled way to modify and access the internal state of objects while allowing for flexibility and encapsulation.

4.10 Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.
Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

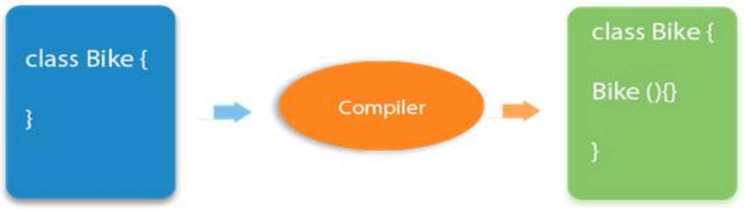
A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>(){}
```

Example of default constructor
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
//Java Program to create and call a default constructor
class Bike1{
        //creating a default constructor
        Bike1(){System.out.println("Bike is created");}
        //main method
        public static void main(String args[]){
                //calling a default constructor
                Bike1 b=new Bike1();
        }
}
```



Output:
```
Bike is created
```

Q) What is the purpose of a default constructor?

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example of default constructor that displays the default values

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
	int id;
	String name;
	//method to display the value of id and name
	void display(){System.out.println(id+" "+name);}

	public static void main(String args[]){
		//creating objects
		Student3 s1=new Student3();
		Student3 s2=new Student3();
		//displaying values of the object
		s1.display();
		s2.display();
	}
}
```

Output:

```
0 null
0 null
```

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
   int id;
   String name;
```

```
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
            id = i;
            name = n;
    }
    //creating three arg constructor
    Student5(int i,String n,int a){
            id = i;
            name = n;
            age = a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
           Student5 s1 = new Student5(111,"Karan");
           Student5 s2 = new Student5(222,"Aryan",25);
          s1.display();
          s2.display();
   }
}
```

Output:

```
111 Karan 0
222 Aryan 25
```

4.11 Overloading Methods

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
	static int add(int a,int b){return a+b;}
	static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
	public static void main(String[] args){
		System.out.println(Adder.add(11,11));
		System.out.println(Adder.add(11,11,11));
	}
}
```

Output:

```
22
33
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
	static int add(int a, int b){return a+b;}
	static double add(double a, double b){return a+b;}
}
class TestOverloading2{
	public static void main(String[] args){
		System.out.println(Adder.add(11,11));
		System.out.println(Adder.add(12.3,12.6));
	}
}
```

Output:

```
22
24.9
```

4.12 Call by value, Call by reference

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is call-by-value. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is call-by-reference. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.

When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method. For example, consider the following program:

```
 // Primitive types are passed by value.
class Test {
        void meth(int i, int j) {
                 i *= 2;
                j /= 2;
        }
}
class CallByValue {
        public static void main(String args[]) {
                Test ob = new Test();
                int a = 15, b = 20;
                System.out.println("a and b before call: " + a + " " + b);
                ob.meth(a, b);
                System.out.println("a and b after call: " + a + " " + b);
        }
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside meth( ) have no effect on the values of a and b used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference. Changes to the object inside the method do affect the object used as an argument. For example, consider the following program:

```
// Objects are passed through their references.

class Test {
             int a, b;
            Test(int i, int j) {
                        a = i;
                        b = j;
            }
            // pass an object
            void meth(Test o) {
                         o.a *= 2;
                        o.b /= 2;
            }
}
class PassObjRef {
            public static void main(String args[]) {
                        Test ob = new Test(15, 20);
                        System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
                        ob.meth(ob);
                         System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
            }
}
```

This program generates the following output:

```
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10
```

As you can see, in this case, the actions inside meth( ) have affected the object used as an argument.

Note: When an object reference is passed to method, the reference itself is passed by use if call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

4.13 this keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the current object. That is, **this** is alwars a reference to the object on which the method was invoked. You can use **this** anywhere a reference to the object of a currect class' type is permitted.

To better understand what this refers to, consider the following version of Box():

```
// a redundant use of this.
Box(double w, double h, double d) {
        this.width = w;
        this.height = h;
        this.depth = d;
}
```

Inside **Box**(), **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other context like:

- this can be used to get the current object.
- this can be used to invoke current object's method.
- this() can be used to invoke current class constructor
- this can be passed as a parameter to a method call.
- this can be passed as a parameter to a constructor.
- this can be used to return the current object from the method.

Another example of **this** is:

```
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
```

4.14 final modifier

In Java, the final modifier is used to apply restrictions on classes, methods, and variables. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) Java final variable

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

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

Output: Compile Time Error

2) Java final method

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

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}
```

```
    public static void main(String args[]){
    Honda honda= new Honda();
    honda.run();
    }
  }
```

Output:Compile Time Error

3) Java final class

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

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

Output:running...

4.15 Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a *nested class* and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Nested classes are divided into two categories: non-static and static. Non-static nested classes are called ***inner classes***. Nested classes that are declared static are called ***static nested classes***.

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
    static class StaticNestedClass {
        ...
    }
}
```

A nested class is a member of its enclosing class. Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared private. Static nested classes do not have access to other members of the enclosing class. As a member of the OuterClass, a nested class can be declared private, public or protected.

Why Use Nested Classes?

Compelling reasons for using nested classes include the following:

- **It is a way of logically grouping classes that are only used in one place**: If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such "helper classes" makes their package more streamlined.
- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

Inner Classes

As with instance methods and variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields. Also, because an inner class is associated with an instance, it cannot define any static members itself.

Objects that are instances of an inner class exist *within* an instance of the outer class. Consider the following classes:

```
class OuterClass {
    ...
    class InnerClass {
        ...
    }
}
```

An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this syntax:

```
OuterClass outerObject = new OuterClass();
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

There are two special kinds of inner classes: local classes and anonymous classes.

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class. And like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference. Inner Class and Nested Static Class Example demonstrates this.

Note: A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class. In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for packaging convenience.

You instantiate a static nested class the same way as a top-level class:

```
StaticNestedClass staticNestedObject = new StaticNestedClass();
```

Inner Class and Nested Static Class Example

The following example, OuterClass, along with TopLevelClass, demonstrates which class members of OuterClass an inner class (InnerClass), a nested static class (StaticNestedClass), and a top-level class (TopLevelClass) can access:

OuterClass.java

```
public class OuterClass {

    String outerField = "Outer field";
    static String staticOuterField = "Static outer field";

    class InnerClass {
        void accessMembers() {
            System.out.println(outerField);
            System.out.println(staticOuterField);
        }
    }

    static class StaticNestedClass {
        void accessMembers(OuterClass outer) {
            // Compiler error: Cannot make a static reference to the non-static
            //     field outerField
            // System.out.println(outerField);
            System.out.println(outer.outerField);
            System.out.println(staticOuterField);
        }
    }

    public static void main(String[] args) {
        System.out.println("Inner class:");
        System.out.println("------------");
        OuterClass outerObject = new OuterClass();
        OuterClass.InnerClass innerObject = outerObject.new InnerClass();
        innerObject.accessMembers();

        System.out.println("\nStatic nested class:");
        System.out.println("--------------------");
        StaticNestedClass staticNestedObject = new StaticNestedClass();
        staticNestedObject.accessMembers(outerObject);

        System.out.println("\nTop-level class:");
        System.out.println("--------------------");
        TopLevelClass topLevelObject = new TopLevelClass();
        topLevelObject.accessMembers(outerObject);
    }
}
```

TopLevelClass.java

```
public class TopLevelClass {

    void accessMembers(OuterClass outer) {
        // Compiler error: Cannot make a static reference to the non-static
        //     field OuterClass.outerField
        // System.out.println(OuterClass.outerField);
        System.out.println(outer.outerField);
        System.out.println(OuterClass.staticOuterField);
    }
}
```

This example prints the following output:

```
Inner class:
------------
Outer field
```

```
Static outer field

Static nested class:
--------------------
Outer field
Static outer field

Top-level class:
--------------------
Outer field
Static outer field
```

Note that a static nested class interacts with the instance members of its outer class just like any other top-level class. The static nested class StaticNestedClass can't directly access outerField because it's an instance variable of the enclosing class, OuterClass. The Java compiler generates an error at the highlighted statement:

```
static class StaticNestedClass {
   void accessMembers(OuterClass outer) {
     // Compiler error: Cannot make a static reference to the non-static
     //     field outerField
     System.out.println(outerField);
   }
}
```

To fix this error, access outerField through an object reference:

```
System.out.println(outer.outerField);
```

Similarly, the top-level class TopLevelClass can't directly access outerField either.

4.16 Wrapper Classes in Java

In Java, wrapper classes are classes that provide a way to use primitive data types (e.g., int, char, boolean) as objects. The Java programming language is designed to work primarily with objects, so when you need to treat primitive types as objects, you can use wrapper classes.

The wrapper classes in Java include:

1. Integer for int
2. Double for double
3. Boolean for boolean
4. Character for char
5. Byte for byte
6. Short for short
7. Long for long
8. Float for float

1. **Autoboxing:**
    - Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class object.
    - It happens implicitly when you assign a primitive value to an object of the corresponding wrapper class.

```
public class AutoboxingExample {
    public static void main(String[] args) {
        int primitiveInt = 42;

        // Autoboxing: converting int to Integer, Same as Integer wrappedInt = Interger.valueOf(i);
        Integer wrappedInt = primitiveInt; //autoboxing

        System.out.println("Primitive Int: " + primitiveInt);
        System.out.println("Wrapped Int: " + wrappedInt);
    }
}
```

Output:

```
Primitive Int: 42
Wrapped Int: 42
```

In this example, the value of primitiveInt is automatically wrapped into an Integer object when assigned to wrappedInt.

2. **Unboxing:**

- Unboxing is the automatic conversion of a wrapper class object to its corresponding primitive type.
- It happens implicitly when you assign a wrapper object to a primitive variable.

```
public class UnboxingExample {
    public static void main(String[] args) {
        Integer wrappedInt = 64;

        // Unboxing: converting Integer to int, Same as int primitiveInt = wrappedInt.intValue();
        int primitiveInt = wrappedInt; //unboxing

        System.out.println("Wrapped Int: " + wrappedInt);
        System.out.println("Primitive Int: " + primitiveInt);
    }
}
```

Output:

```
Wrapped Int: 64
Primitive Int: 64
```

In this example, the value of wrappedInt is automatically unwrapped into an int when assigned to primitiveInt.

Autoboxing and unboxing simplify the process of working with both primitive types and objects in Java, providing a seamless transition between the two. They were introduced in Java 5 to make the code more concise and readable when dealing with collections and other situations where objects are preferred over primitive types.

4.17 Garbage Collection

Garbage collection in Java 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.
In C/C++, a programmer is responsible for both the creation and destruction of objects. Usually, programmer neglects the destruction of useless objects. Due to this negligence, at a certain point, sufficient memory may not be available to create new objects, and the entire program will terminate abnormally, causing **OutOfMemoryErrors**.
But in Java, the programmer need not care for all those objects which are no longer in use. Garbage collector destroys these objects. The main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**. The garbage collector is the best example of the Daemon thread as it is always running in the background.
Java garbage collection is an automatic process. Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in-use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused or unreferenced object is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed. The programmer does not need to mark objects to be deleted explicitly. The garbage collection implementation lives in the JVM.

Finalization

- Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.
- *finalize()* method is present in Object class with the following prototype.

protected void finalize() throws Throwable

Based on our requirement, we can override *finalize()* method for performing our cleanup activities like closing connection from the database.

1. The *finalize()* method is called by Garbage Collector, not JVM. However, Garbage Collector is one of the modules of JVM.
2. Object class *finalize()* method has an empty implementation. Thus, it is recommended to override the *finalize()* method to dispose of system resources or perform other cleanups.

3. The *finalize()* method is never invoked more than once for any object.
4. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored, and the finalization of that object terminates.

Advantages of Garbage Collection in Java

- It makes java memory-efficient because the garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector(a part of JVM), so we don't need extra effort.

Example:

```
public class TestGarbage1 {

   // finalize method to be called by the garbage collector
   public void finalize() {
      System.out.println("object is garbage collected");
   }

   public static void main(String args[]) {
      // Creating two instances of TestGarbage1
      TestGarbage1 s1 = new TestGarbage1();
      TestGarbage1 s2 = new TestGarbage1();

      // Setting references to null, making objects eligible for garbage collection
      s1 = null;
      s2 = null;

      // Explicitly requesting garbage collection (not guaranteed to execute immediately)
      System.gc();
   }
}
```

Explanation:

- TestGarbage1 class has a finalize() method, which will be called by the garbage collector before an object is reclaimed.
- Two objects (s1 and s2) of type TestGarbage1 are created.
- Both references (s1 and s2) are set to null, indicating that the objects are no longer reachable and are eligible for garbage collection.
- System.gc() is called to suggest to the JVM that garbage collection should be performed. The actual execution of the garbage collector is not guaranteed at this point.
- If the garbage collector runs (which is non-deterministic), it will call the finalize() method for each object before collecting them.
- The output "object is garbage collected" is printed for each object that undergoes garbage collection.

It's important to note that in practical Java programming, explicit requests for garbage collection (using System.gc()) are generally discouraged, and the garbage collector is designed to run automatically as needed. The output may vary depending on the JVM implementation and configuration.