

# OBJECT ORIENTED PROGRAMMING

## CHAPTER TWO

### CLASS AND OBJECT

**Object-oriented programming (OOP)** is a programming paradigm that organizes code by creating objects, which are instances of classes. It focuses on modeling real-world entities, where objects have both attributes (data) and behaviors (operations or methods) associated with them.

In OOP, classes serve as blueprints or templates for creating objects. A class defines the properties and behaviors that its objects will possess. Properties are represented as attributes or variables, which store data specific to each object. Behaviors are implemented as methods or functions, which define the actions that objects can perform.

When an object is created from a class, it is said to be an instance of that class. Each object has its own set of attributes, which may have different values. However, the methods defined in the class are shared among all objects of that class and can operate on the object's attributes.

#### **Procedural Programming:**

1. **Focus:** Procedural programming focuses on procedures or functions that operate on data. It follows a step-by-step approach to execute a sequence of instructions.
2. **Data and Operations:** In procedural programming, data and operations are separate. Functions operate on data that is passed to them explicitly. Data is often stored in global variables, accessible to multiple functions.
3. **Code Structure:** The code in procedural programming is organized around functions. Functions take input parameters, perform a series of steps, and produce output. Control structures like

loops and conditionals are used for flow control.

4. **Reusability:** Code reuse in procedural programming is achieved through functions. Functions can be called from different parts of the program, reducing code duplication.
5. **Modifiability:** Procedural code can be more challenging to modify and maintain, especially as the codebase grows larger. Modifying one function may require changes in multiple places that access or modify the same data.
6. **State Management:** Data in procedural programming can be shared and modified freely across functions, making it harder to manage and track the state of the program accurately.

#### **Object-Oriented Programming:**

1. **Focus:** Object-oriented programming focuses on objects, which are instances of classes. It emphasizes the organization of code around real-world entities and their interactions.
2. **Data and Operations:** In object-oriented programming, data and operations are encapsulated within objects. Objects have their own data (attributes) and can perform actions (methods) on that data. Data hiding and encapsulation allow objects to control access to their internal state.
3. **Code Structure:** Code in object-oriented programming is organized around classes and objects. Classes define the structure and behavior of objects, while objects are instances of classes that interact with each other through methods.
4. **Reusability:** OOP promotes code reuse through inheritance and composition. Inheritance allows the creation of derived

classes that inherit properties and behaviors from base classes. Composition allows objects to be composed of other objects, enabling complex structures and behaviors to be built from simpler components.

5. **Modifiability:** OOP promotes modularity, making code easier to modify and maintain. Objects encapsulate data and behavior, providing clear interfaces for interaction. Modifying one class or object typically has a localized impact and does not require changes in unrelated parts of the program.
6. **State Management:** OOP encourages better state management by encapsulating data within objects. Objects maintain their state internally, and interactions are performed through well-defined methods, ensuring controlled access to data.

Both procedural and object-oriented programming have their strengths and are suited for different scenarios. Procedural programming is often more straightforward for small-scale programs or when the focus is on a specific set of tasks. Object-oriented programming excels in complex systems where entities, their attributes, and interactions need to be modeled accurately and maintained over time.

**Thinking in Terms of Objects:** Java programming encourages thinking in terms of objects. Instead of approaching a problem procedurally or focusing solely on algorithms, Java developers are encouraged to identify the relevant objects in the problem domain and design classes to represent them. This shift in mindset allows for a more intuitive and structured approach to problem-solving, aligning the program's structure with the real-world entities it models.

**Java as a Collection of Cooperating Objects:** In Java, a program can be viewed as a collection of cooperating objects. This means that the program consists of multiple objects that interact with each other, exchanging data and invoking each other's methods to accomplish tasks. The

collaboration among objects mirrors the relationships and interactions observed in the real world, enabling developers to create more robust and comprehensive software solutions.

### **Objects as Representations of Real-World Entities:**

Objects in Java represent entities in the real world. Almost anything with distinct characteristics and behaviors can be represented as an object. Examples given in the statement include students, desks, circles, buttons, and loans. By modeling these entities as objects, developers can capture their essential properties and behaviors, making it easier to design and implement software that accurately reflects the real-world scenario being addressed.

## **OBJECTS IN JAVA**

An object has a unique identity , state and behavior.

**Unique Identity:** An object has a unique identity that distinguishes it from other objects. This identity is typically assigned automatically by the programming language or runtime environment. It allows the program to differentiate between different instances of the same class. In Java, for example, each object has a unique memory address or reference that can be used to access and manipulate the object.

**Data Fields (Properties):** Data fields, also known as properties or instance variables, represent the object's data or attributes. They hold the values that make up the state of the object. In the case of a circle object, the data field "radius" would hold the value representing the length of the circle's radius. Data fields can have different types (e.g., int, double, String) depending on the nature of the data they store.

**Behaviors:** Behaviors define what an object can do or the actions it can perform. They are typically implemented as methods associated with the object. In the example of a circle object, one behavior mentioned is the ability to compute the area of the circle using the method "getArea()". The getArea() method would use the current value of the radius data field to calculate the

area of the circle based on the defined mathematical formula.

The example of a circle object illustrates the concepts of object-oriented programming. The object (circle) has a unique identity, a state represented by its data fields (such as radius), and behaviors defined by its methods (such as computing the area using the `getArea()` method). This encapsulation of data and behavior within an object allows for a more intuitive and structured way of representing and working with entities in a program.

**Example :-** If we create a software object that models our television. Let's define the variables representing the television's current state and the methods describing the permissible actions in detail:

#### **Variables (Current State of the Television):**

- The "status" is on.
- Current Channel
- Current Volume
- Input From Remote

#### **Methods (Permissible Actions):**

- Turn On/Off
- Change Channel
- Change Volume
- accept Input From Remote

These variables and methods encapsulate the state and behavior of the television object in the software. The variables represent the current state of the television, including its status (on/off), channel setting, volume setting, and input from the remote control. The methods provide the permissible actions, allowing the user to turn the television on/off, change channels, adjust the volume, and interact with the remote control. By modeling the television as an object with variables and methods, the software can simulate

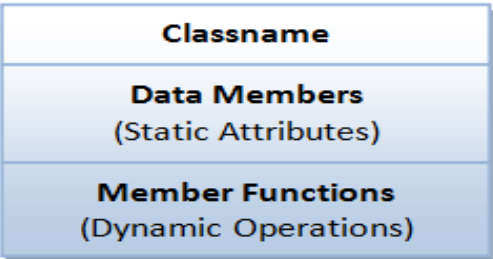
the behavior and functionality of a real television.

### **CLASSES AND INSTANCES**

The below visualization of a class as a three-compartment box, which helps in understanding the structure and components of a class. Let's define each compartment in detail:

1. **Name (or Identity):** The "Name" compartment of a class represents the identifier or name of the class. It serves as a unique label that distinguishes the class from other classes in the program. The name of the class is typically chosen to reflect the purpose or nature of the objects that will be instantiated from it. For example, a class representing a Circle could be named "Circle" or a class representing a Student could be named "Student."
2. **Variables (or Attributes, State, Fields):** The "Variables" compartment contains the static attributes or properties of the class. These attributes define the state or characteristics of the objects created from the class. They represent the data associated with each object and store information that can vary among different instances of the class. For example, in a Circle class, the radius and color could be attributes that define the state of individual Circle objects. These attributes are declared within the class and can have different data types (e.g., int, double, String) based on the nature of the data they hold.
3. **Methods (or Behavior, Function, Operations):** The "Methods" compartment contains the dynamic behaviors or operations that can be performed by the objects of the class. Methods define the actions that objects can take or the behaviors they exhibit. They encapsulate the algorithms or series of steps to perform certain tasks or manipulate the data associated with the objects. For

example, in a Circle class, methods like calculateArea() and calculateCircumference() could be defined to compute the respective values based on the radius attribute of the Circle objects.



A class is a 3-compartment box encapsulating data and functions

Together, these compartments represent the structure and components of a class. The "Name" compartment provides a unique identifier for the class, the "Variables" compartment holds the static attributes that define the state of the objects, and the "Methods" compartment defines the dynamic behaviors or operations that objects can perform. This visualization helps in understanding the role and purpose of a class in object-oriented programming, as well as the encapsulation of data and behavior within it.

Examples of classes

Name (Identifier)	Student	Circle
	id name	radius color
	getName() printGrade()	getRadius() getArea()

SoccerPlayer	Car
name number xLocation yLocation	plateNumber xLocation yLocation speed
run() jump() kickBall()	move() park() accelerate()

Examples of classes ific

occurrence of a class. It represents a single object created based on the structure defined by a class.

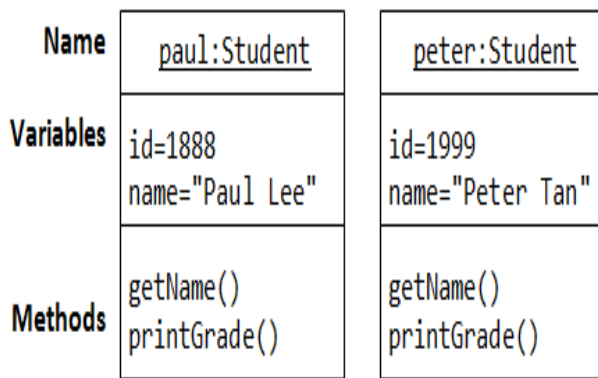
Each instance has its own set of data and can invoke the methods defined in the class. Think of an instance as a concrete manifestation of the abstract blueprint defined by the class.

**Instantiation:** Instantiation is the process of creating an instance of a class. It involves allocating memory for the object and initializing its attributes and state based on the class definition. When an instance is created, it possesses all the attributes and behaviors defined by the class. It becomes a distinct entity that can be manipulated independently of other instances.

**Similar Properties:** All instances of a class share similar properties or attributes as described in the class definition. These attributes represent the state or characteristics of the objects. For example, if we have a class called "Student," all instances of that class will have attributes like name, age, and student ID. However, the values of these attributes may differ among different instances, making each instance unique.

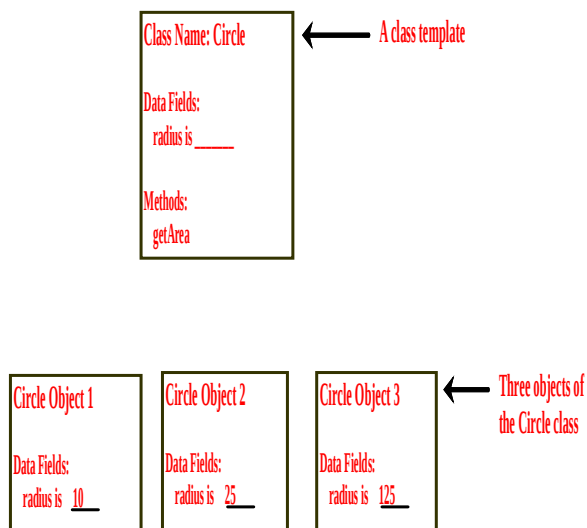
**Object:** The term "object" is often used interchangeably with "instance," but it can also have a broader meaning depending on the context. In a general sense, an object can refer to both a class and an instance. When used to refer to a class, it represents the abstract blueprint or template that defines the structure and behavior of objects. When used to refer to an instance, it represents a specific occurrence or realization of that class, possessing its own unique data and behavior.

So an instance is a specific occurrence or realization of a class, representing a unique object with its own data and behaviors. Instantiation is the process of creating an instance based on a class. Instances share similar properties as defined in the class but can have different values for their attributes. The term "object" can refer to both a class and an instance, depending on the context in which it is used.



**2 instances of the class Student**

The above class diagrams are drawn according to the UML (Unified Modeling Language). A class is represented as a 3-compartment box, containing name, variables and method. Class name is shown in bold and centralized. Instance name is shown as **instanceName:classname** and underlined.



Each object within a class retains its own states and behaviors. A single class can be used to instantiate multiple objects. This means that we can have many active objects or instances of a class. A class usually represents a noun, such as a person, place.

## Constructors in Java

In Java, constructors are special methods within a class that are used to initialize objects. They are invoked automatically when an object of the class is created using the new keyword or through other mechanisms like cloning or

deserialization. The purpose of constructors is to ensure that newly created objects are properly initialized before they can be used.

Here are the key points to define constructors in Java:

1. **Name:** Constructors have the same name as the class in which they are defined. This ensures that the constructor is associated with that particular class.
2. **No Return Type:** Constructors do not have a return type, not even void. This is because constructors are automatically called when an object is created and are responsible for initializing the object, rather than returning a value.
3. **Initialization:** Constructors initialize the instance variables (attributes) of an object, setting their initial values. This can involve assigning default values or accepting parameters to initialize the object's state.
4. **Overloading:** Like other methods, constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists. Overloaded constructors provide flexibility in creating objects by accepting different sets of input values.
5. **Default Constructor:** If a class does not explicitly define any constructors, a default constructor is automatically provided by Java. This default constructor takes no parameters, calls the default parent constructor super () and initializes instance variables with default values (e.g., 0 for numeric types, null for object references, false for boolean).
6. **Access Modifiers:** Constructors can have access modifiers like public, private, or protected. These modifiers control the visibility and accessibility of the constructor.
7. **Chaining:** Constructors can also be chained within a class using the this()



keyword. This allows one constructor to call another constructor of the same class, simplifying the initialization process and reducing code duplication.

Constructors play a crucial role in object initialization, allowing for the proper setup of object state. They ensure that objects are created in a valid and consistent state before they are used in the program. Constructors, along with getter and setter methods, form the basis for encapsulation, enabling controlled access and manipulation of object data.

### Constructors Difference From Normal Methods

Constructors differ from normal methods in several ways:

1. **Name:** Constructors have the same name as the class in which they are defined, while normal methods have unique names that reflect their purpose or functionality.
2. **Return Type:** Constructors do not have a return type, not even void. They are automatically called when an object is created and are responsible for initializing the object's state. Normal methods, on the other hand, have a return type, which specifies the type of value they return after performing their operation.
3. **Invocation:** Constructors are automatically invoked during object creation using the new keyword or other mechanisms like cloning or deserialization. They are not called explicitly like normal methods, which are invoked by using their name along with parentheses and passing any required parameters.
4. **Purpose:** The primary purpose of constructors is to initialize object instances by setting initial values to the object's attributes or performing any necessary setup operations. Normal methods, on the other hand, perform specific actions or operations on objects, often utilizing the object's state.
5. **Accessibility:** Constructors can have access modifiers (e.g., public, private, protected) that control their visibility and accessibility. Normal methods can also have access modifiers, but they are not restricted to the same limitations as constructors. Constructors are typically used to ensure the proper creation of objects, while normal methods can have a wider range of purposes and access levels.
6. **Overloading:** Constructors can be overloaded, meaning a class can have multiple constructors with different parameter lists (signatures). This allows for flexibility in creating objects with different initializations. Normal methods can also be overloaded, providing variations of the **same functionality** with different parameter lists, but overloading constructors is more common.
7. **Usage:** Constructors are typically used at the time of object creation to set up the initial state, while normal methods are used to perform various operations and actions on objects after they have been created.

### Default constructor example

```
java Copy code
public class ExampleClass {
    private int intValue;
    private String stringValue;

    public void displayInfo() {
        System.out.println("int value: " + intValue);
        System.out.println("String value: " + stringValue);
    }

    public static void main(String[] args) {
        ExampleClass example = new ExampleClass(); // Creating an object using default constructor
        example.displayInfo();
    }
}
```

## Example of constructors

```
java Copy code

public class Person {
    private String name;
    private int age;

    // Default Constructor
    public Person() {
        name = "John Doe";
        age = 30;
    }

    // Constructor with Parameters
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Getter methods
    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    // Setter methods
    public void setName(String name) {
        this.name = name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    // Method to display person's information
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }

    public static void main(String[] args) {
        // Creating objects using constructors
        Person person1 = new Person(); // Default constructor
        Person person2 = new Person("Alice", 25); // Constructor with parameters

        // Accessing object's attributes and invoking methods
        System.out.println("Person 1:");
        person1.displayInfo();
        System.out.println();

        System.out.println("Person 2:");
        person2.displayInfo();
    }
}
```

## Output

```
java Copy code

public class Car {
    private String make;
    private String model;
    private int year;

    // Default constructor
    Regenerate response
```

## CREATING OBJECTS USING CONSTRUCTORS

Now let's describe the process of creating objects from classes using the new operator in Java. Let's define each part in detail:

### Object Creation using the new Operator:

The new operator is used to create a new instance of a class.

**Syntax:**    `ClassName objectRefVar = new ClassName([parameters]);`

or `ClassName objectRefVar;`  
`objectRefVar = new ClassName([parameters]);`

The `ClassName` represents the name of the class from which the object is being created. `objectRefVar` is a reference variable that will refer to the newly created object.

The **new keyword** allocates memory for the object and initializes its attributes with default values or as specified in constructors.

### Object Reference Variable :-

The `objectRefVar` is a variable of the class type that serves as a reference to the created object. It allows us to access and manipulate the object's attributes and invoke its methods.

### Constructors and Object Creation:

Objects can be created using any available constructors defined in the class. The constructors define how the object is initialized and allow for specific values to be passed during object creation. If a class has constructors other than the default constructor, appropriate parameters must be provided to create objects using those constructors. By passing different sets of parameters, objects can be created with different initial states, allowing for flexibility and customization.

```
java Copy code

public class Rectangle {
    private int length;
    private int width;

    // Parameterized constructor
    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public void displayInfo() {
        System.out.println("Length: " + length);
        System.out.println("Width: " + width);
    }

    public static void main(String[] args) {
        // Creating objects using the new operator and constructor
        Rectangle rectangle1 = new Rectangle(5, 3); // Object created with
        Rectangle rectangle2 = new Rectangle(7, 4); // Object created with

        // Accessing object attributes and invoking methods
        System.out.println("Rectangle 1:");
        rectangle1.displayInfo();
        System.out.println();

        System.out.println("Rectangle 2:");
        rectangle2.displayInfo();
    }
}
```

## ACCESSING OBJECTS

The given statement highlights the essence of object-oriented programming, which revolves around accessing an object's data fields and invoking its methods using the dot operator (also known as the object member access operator). Let's define this in detail:

### 1. Referencing the Object's Data Fields:

- Once an object is created, its data fields (also known as attributes or properties) can be accessed and modified using the dot operator.
- The dot operator is used to connect the object reference variable with the specific data field, allowing direct access to the object's state.
- **Syntax:** `objectRefVar.dataField`
- **Example:** `myCircle.radius = 100;`

### 2. Invoking the Object's Methods:

- Objects have behaviors associated with them, represented by methods.
- Methods are functions or operations that can be performed by the object, manipulating its data or providing useful functionality.
- To invoke an object's method, the dot operator is used to connect the object reference variable with the specific method, followed by any required arguments.
- **Syntax:**  
`objectRefVar.methodName(arguments)`
- **Example:** `double area = myCircle.getArea();`

### 3. Object-Oriented Programming Focus:

- Object-oriented programming (OOP) revolves around objects as the focal point of the design and implementation process.

- In OOP, objects represent entities from the real world or abstract concepts and encapsulate both their data (attributes) and behavior (methods).
- The dot operator plays a crucial role in OOP by enabling access to an object's internal data and the invocation of its methods, allowing interaction and manipulation of the object's state.

By using the **dot operator**, object-oriented programming facilitates a natural and intuitive way to work with objects. It emphasizes the concept of objects as self-contained entities with their own state and behavior. Objects can be accessed and modified through their data fields, providing direct control over their state. Additionally, objects can exhibit their behavior by invoking methods, enabling the execution of specific operations or functionalities associated with the object.

#### Example 1 :- Referencing object's Data Fields

```
java Copy code

public class Circle {
    public double radius;

    public static void main(String[] args) {
        Circle myCircle = new Circle();
        myCircle.radius = 5.0; // Accessing and modifying the radius data field
        System.out.println("Circle radius: " + myCircle.radius);
    }
}
```

#### Example 2 :- Invoking Objects Methods

```
java Copy code

public class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public int calculateArea() {
        return length * width; // Method to calculate the area
    }

    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle(4, 3);
        int area = rectangle.calculateArea(); // Invoking the calculateArea method
        System.out.println("Rectangle area: " + area);
    }
}
```



## Data Fields

Data fields, also known as instance variables or attributes, are variables declared within a class to hold data or state information associated with objects created from the class. They define the characteristics or properties of the objects and represent their state or data.

Data fields can be of two types: **primitive types** or **reference types**. Primitive types include `int`, `double`, `boolean`, `char`, etc., and hold simple values directly. Reference types include classes, arrays, and interfaces, and hold references or addresses to objects stored in memory.

When data fields are declared but not explicitly initialized, they are assigned default values. The default value depends on the type of the data field: **For reference types: The default value is null**, indicating that the reference doesn't point to any object. **For numeric types (`int`, `double`, etc.): The default value is 0**. **For boolean type: The default value is false**. **For char type: The default value is `\u0000`**, which represents the null character.

It's important to note that the default values are assigned if no explicit initialization is done for the data fields. However, if the data fields are explicitly initialized, their values will be set to the assigned values instead of the default values.

```
java Copy code

public class ExampleClass {
    private int number;        // Default value: 0
    private String text;       // Default value: null
    private boolean flag;      // Default value: false
    private char character;     // Default value: '\u0000'

    public static void main(String[] args) {
        ExampleClass example = new ExampleClass();
        System.out.println("number: " + example.number);
        System.out.println("text: " + example.text);
        System.out.println("flag: " + example.flag);
        System.out.println("character: " + example.character);
    }
}
```

## DEFAULT VALUE IN LOCAL VARIABLES

Java does not assign any default value to a local variable declared inside a method. Let's explain this in detail:

### Local Variables:

- Local variables are variables declared within a method, constructor, or block of code.
- They are created when the method or block is entered and destroyed when the method or block is exited.
- Local variables are typically used to store temporary or intermediate data within a specific scope.

### Default Values:

- In Java, local variables do not have default values assigned to them by the compiler or runtime.
- Unlike instance variables (data fields) that are automatically assigned default values, local variables must be explicitly initialized before they can be used.
- If a local variable is declared but not initialized before accessing its value, a compilation error will occur.

### Explicit Initialization:

- Local variables must be explicitly initialized before they are accessed or used within the method or block.
- Initialization involves assigning an initial value to the variable using the assignment operator (`=`).
- Local variables can be initialized with literal values, values from method parameters, or computed values based on other variables

```
java Copy code

public class ExampleClass {
    public void exampleMethod() {
        int number; // Declaration of a local variable

        // Attempting to access the local variable without initialization
        System.out.println(number); // Compilation error: Variable 'number'

        // Initializing the local variable before using it
        number = 10;
        System.out.println(number); // Valid output: 10
    }
}
```

## Initializing Data Fields

data fields can be initialized during their declaration and the common approach of initializing data fields inside constructors:

### Initialization during Declaration :-

- x Data fields can be initialized directly at the point of their declaration within the class.
- x This involves assigning an initial value to the data field immediately after declaring it, using the assignment operator (=).
- x Initialization during declaration is particularly useful when a data field has a fixed or default value that applies to all instances of the class.
- x This initialization approach allows data fields to have specific initial values without the need for explicit initialization in each constructor.

### Example of Initialization during Declaration:

```
java Copy code
public class ExampleClass {
    private int number = 10; // Initialization during declaration
    private String text = "Hello"; // Initialization during declaration
}
```

### Initialization inside Constructors:

- x The most common way to initialize data fields is inside constructors.
- x Constructors are special methods within a class that are used to initialize objects when they are created.
- x By including parameterized constructors or overloaded constructors, you can provide different ways to initialize the data fields based on specific values passed during object creation.
- x Initialization inside constructors allows for more flexibility and customization of the initial state of objects.
- x Constructors can perform additional operations and logic while initializing the

data fields, such as validation or computation, before assigning values.

### Example of Initialization inside Constructors:

```
java Copy code
public class ExampleClass {
    private int number;
    private String text;

    public ExampleClass(int number, String text) {
        this.number = number; // Initializing data field inside constructor
        this.text = text; // Initializing data field inside constructor
    }
}
```

## TYPES OF VARIABLES

### Local Variables:

- Local variables are variables declared within a method, constructor, or block of code.
- They are created when the method or block is entered and destroyed when the method or block is exited.
- Local variables have limited scope, meaning they can only be accessed within the block of code in which they are declared.
- Local variables must be explicitly initialized before they are used, as Java does not assign default values to them.

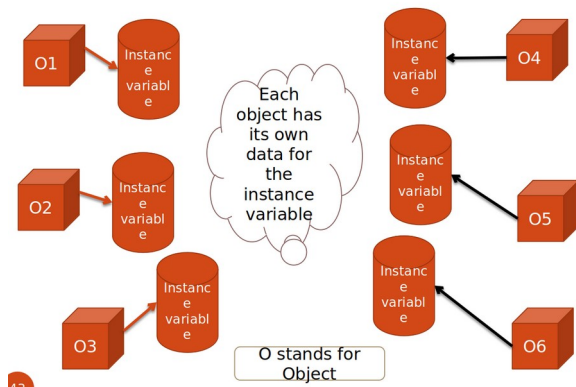
### Example of local variable :-

```
java Copy code
public class ExampleClass {
    public void exampleMethod() {
        int number = 10; // Declaration and initialization of a local variable
        System.out.println(number); // Accessing the local variable
    }
}
```

### Instance Variables:

- Instance variables, also known as data fields or member variables, are declared within a class but outside any method, constructor, or block.
- Instance variables belong to an instance of the class (an object) and persist as long as the object exists.

- Each instance of the class has its own set of instance variables, allowing objects to maintain their individual state.
- Instance variables are initialized with default values if not explicitly assigned values.



### Example of Instance Variable:

```
java
Copy code

public class ExampleClass {
    private int number; // Declaration of an instance variable

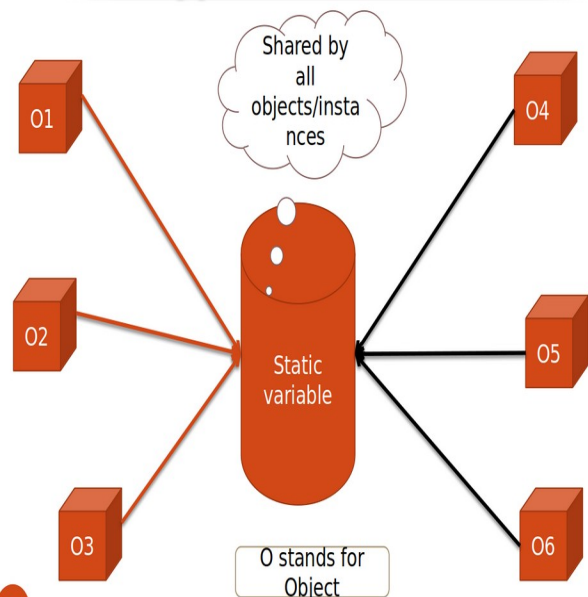
    public void setNumber(int value) {
        number = value; // Assigning a value to the instance variable
    }

    public void displayNumber() {
        System.out.println(number); // Accessing the instance variable
    }
}
```

### Static Variables:

- Static variables, also known as class variables, are declared within a class but outside any method, constructor, or block, using the static keyword.
- Unlike instance variables, static variables belong to the class itself rather than individual instances (objects).
- Static variables are shared among all instances of the class, meaning they have the same value for all objects.
- Static variables are initialized once, when the class is loaded, and they retain their value until the program terminates.

- Static variables can be accessed directly through the class name, without the need to create an object of the class.



### Examples of static variables

```
java
Copy code

public class ExampleClass {
    private static int count = 0; // Declaration and initialization of a static variable

    public ExampleClass() {
        count++; // Incrementing the static variable in the constructor
    }

    public static void main(String[] args) {
        ExampleClass obj1 = new ExampleClass();
        ExampleClass obj2 = new ExampleClass();
        System.out.println("Number of instances: " + count); // Accessing the static variable
    }
}
```

## INSTANCE VARIABLES AND METHODS

### Instance Variables:

- Instance variables, also known as member variables or attributes, are associated with individual instances (objects) of a class.
- Each object of a class has its own set of instance variables, which define the object's state or properties.
- Instance variables are declared within a class but outside any method or constructor.
- They hold unique values for each object and are not shared among objects of the same class.

```

java Copy code

public class Circle {
    private double radius; // Instance variable

    public Circle() {
        radius = 0.0; // Default value for radius
    }

    public Circle(double r) {
        radius = r; // Assigning a value to radius
    }
}

```

In this example, the Circle class has an instance variable `radius`, which represents the radius of a circle. Each object of the Circle class will have its own `radius` value, independent of other objects. The instance variable `radius` is accessed using the dot operator (`objectName.radius`).

### Instance Methods:

- Instance methods, also called member methods, are associated with individual instances (objects) of a class.
- They operate on the instance variables and perform actions or provide functionality specific to each object.
- Instance methods are defined within a class and can access and manipulate the instance variables of the object they belong to.
- They are invoked using the dot operator on an object reference.

```

java Copy code

public class Circle {
    private double radius; // Instance variable

    public Circle() {
        radius = 0.0; // Default value for radius
    }

    public void setRadius(double r) {
        radius = r; // Updating the radius value
    }

    public double getArea() {
        return Math.PI * radius * radius; // Calculating and returning the
    }
}

```

In this example, the Circle class has two instance methods: `setRadius()` and `getArea()`. The `setRadius()` method allows updating the radius value of a circle object, and the `getArea()` method calculates and returns the area of the circle. Each object of the class can invoke these methods independently, and the methods operate on the specific instance variables of the object.

## STATIC VARIABLES AND METHODS

### Static (Class) Variables:

- Static variables, also known as class variables, are associated with a class itself rather than with individual instances (objects) of the class.
- There is only one copy (one shared memory) of a static variable shared among all instances of the class.
- Changes to the value of a static variable are reflected across all instances of the class.
- Static variables are defined using the `static` keyword before the variable declaration.

### Example:-

```

java Copy code

public class FamilyMember {
    static String surname = "Johnson"; // Static variable
    String name;
    int age;
    // ...
}

```

In this example, the FamilyMember class has a static variable `surname` to store the common surname for all family members. Each instance of the class (FamilyMember objects) will have its own name and age variables, which are specific to each family member. However, the surname variable is shared among all instances of the class. Changes to the surname variable will affect all instances.

### Accessing Static Variables:

- Static variables can be accessed using the dot notation, just like instance variables.
- You can access static variables either through an instance of the class or using the class name itself.
- Both approaches will retrieve or modify the value of the static variable.

### Example :-

```
java Copy code

public class FamilyMember {
    static String surname = "Johnson";
    String name;
    int age;
    // ...
}

public class Main {
    public static void main(String[] args) {
        FamilyMember dad = new FamilyMember();
        System.out.println("Family's surname is: " + dad.surname); // Access
        System.out.println("Family's surname is: " + FamilyMember.surname);
    }
}
```

In this example, the Main class contains the main method. We create an instance dad of the FamilyMember class and use it to access the static variable surname using the dot operator (dad.surname). We can also directly access the static variable using the class name (FamilyMember.surname). Both approaches will yield the same value.

```
csharp Copy code

Family's surname is: Johnson
Family's surname is: Johnson
```

## Accessing static and instance variables

### Static Variables and Methods:

- Static variables and methods can be accessed from both instance methods and static methods within a class.
- Static variables and methods belong to the class itself and are not tied to any specific instance (object) of the class.
- They can be accessed using the class name or through an instance of the class.
- Static variables and methods can be accessed directly from static methods without the need for an instance of the class.

### Instance Variables and Methods:

- Instance variables and methods can only be accessed from instance methods of a class.
- Instance variables and methods belong to individual instances (objects) of the class and hold data specific to each object.

- They can be accessed and modified through the object's reference (instance) within instance methods.

```
java Copy code

public class Foo {
    int i = 5; // Instance variable
    static int k = 2; // Static variable

    public static void main(String[] args) {
        int j = i; // Error: Cannot access instance variable 'i' in a sta
        m1(); // Error: Cannot call instance method 'm1()' in a sta
    }

    public void m1() {
        i = i + k + m2(3, 4); // Correct: Accessing instance and static va
    }

    public static int m2(int i, int j) {
        return (int) Math.pow(i, j); // Correct: Using static method 'pow'
    }
}
```

In this example, the Foo class contains both instance and static variables/methods. In the main method (a static method), an attempt to access the instance variable i and call the instance method m1() results in compilation errors. This is because static methods cannot directly access instance variables or call instance methods.

However, within the instance method m1(), the instance variable i and static variable k can be accessed and modified correctly. The static method m2() can also be called within m1() since it is an instance method.

In the given example, the m2() method is declared as a static method. In a static method, you cannot directly access instance variables (non-static variables) without having an instance (object) of the class. Therefore, you cannot use the instance variables i and j directly in the static method.

However, if you want to use the values of instance variables i and j in the m2() method, you can pass them as parameters when calling the method from an instance method or by creating an instance and accessing the variables through the instance. By doing so, you can effectively use the values of instance variables within the static method.



## CONSTANTS IN JAVA

### Constants:

- Constants are fixed values that do not change during the execution of a program.
- In Java, constants are typically declared using the final keyword to indicate that their values cannot be modified once assigned.
- Constants are often used to represent values that are universal and shared across multiple instances or objects of a class.

### Declaring Constants as final static:

- When a constant is declared as final static, it means that the constant value is both final (unchangeable) and static (shared by all instances of the class).
- Declaring constants as final static ensures that the constant retains its value throughout the program's execution and is accessible without the need for an instance of the class.

```
java Copy code

public class MathConstants {
    public static final double PI = 3.14159265; // Declaring PI as a final
    public static final int MAX_VALUE = 100;    // Declaring MAX_VALUE as a
}
```

In this example, the MathConstants class declares two constants: PI and MAX\_VALUE. Both constants are declared as final static. The PI constant represents the mathematical value of pi, and the MAX\_VALUE constant represents the maximum allowed value.

By declaring these constants as final static, their values remain constant throughout the program's

```
java Copy code

public class Main {
    public static void main(String[] args) {
        double radius = 5.0;
        double circumference = 2 * MathConstants.PI * radius;
        System.out.println("Circumference: " + circumference);

        int value = 75;
        if (value > MathConstants.MAX_VALUE) {
            System.out.println("Value exceeds the maximum allowed.");
        }
    }
}
```

execution, and they can be accessed without the need to create an instance of the MathConstants class.

In this usage example, the main method of the Main class calculates the circumference of a circle using the PI constant from the MathConstants class. It also checks if a value exceeds the maximum allowed value using the MAX\_VALUE constant.

By declaring constants as final static, they provide a way to define universally applicable values that remain constant across all instances of the class. They can be accessed directly through the class name, without the need for an instance, ensuring consistent and efficient usage of shared constant values.

It's worth noting that conventionally, constant variable names are written in uppercase letters with underscores to improve readability and indicate their constant nature.

## MODIFIERS IN JAVA

An access modifier in Java is a keyword used to set the level of access or visibility of classes, member variables (fields), and methods within a program. It determines how these elements can be accessed or invoked by other parts of the program, including other classes and packages.

The access modifiers in Java include public, private, protected, and the default (package-private) modifier. Each access modifier specifies different levels of accessibility:

- **public:** Allows unrestricted access from any class or package.
- **private:** Restricts access to within the same class.
- **protected:** Allows access within the same package and subclasses, even if they are in a different package.
- **Default (Package-private):** Allows access within the same package only.

By using access modifiers, you can control the visibility and accessibility of your code, ensuring

that certain elements are accessible only to the appropriate parts of the program while hiding or restricting access to others. Access modifiers play a crucial role in encapsulation, information hiding, and establishing proper boundaries in object-oriented programming.

## CLASS LEVEL AND MEMBER LEVEL ACCESS MODIFIERS

### Class Level Access Modifiers:

- Class level access modifiers are used to control the accessibility of classes themselves.
- In Java, there are two class level access modifiers: **public** and **no modifier**.
- If a class is declared as **public**, it can be accessed from anywhere in the program, including from other packages.
- If a class has no explicit access modifier (i.e., no modifier keyword is used), it is considered to have package-private (default) access. In this case, the class can only be accessed within the same package.

The class level access modifiers determine the visibility of the class itself, defining whether it can be used or accessed by other classes or packages.

### Member Level Access Modifiers:

- Member level access modifiers are used to control the accessibility of member variables (fields) and methods within a class.
- In Java, all four access modifiers (**public**, **private**, **protected**, and **no modifier**) can be used at the member level.
- **public**: A member variable or method declared as **public** can be accessed from

anywhere in the program, including from other classes or packages.

- **private**: A member variable or method declared as **private** can only be accessed within the same class. It is not visible or accessible from outside the class, including subclasses or other classes in the same or different packages.
- **protected**: A member variable or method declared as **protected** can be accessed within the same package and in subclasses, even if they are in different packages. It provides a level of accessibility between **public** and **private**.
- **No Modifier (Package-private)**: A member variable or method with no explicit access modifier (i.e., no modifier keyword is used) has package-private (default) access. It can be accessed within the same package but not from outside the package.

Member level access modifiers control the visibility and accessibility of the individual members (variables and methods) within a class, defining who can use or invoke them and from where.

It's important to note that the accessibility rules apply based on the access modifiers at both the class and member level. For example, if a class has a **public** access modifier, but its member variables or methods have **private** access modifiers, those members will not be accessible from outside the class.

By utilizing the appropriate access modifiers, you can define the level of encapsulation, information hiding, and interaction between classes, packages, and their members. Access modifiers are essential for maintaining proper access control and ensuring the integrity and security of your code.

Access Modifiers	Same Class	Same Package	Subclass	Other packages
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

## THIS KEYWORD

1) The "this" keyword is used as a reference to the current object within an instance method or constructor in Java:

### 1. Reference to the Current Object:

- Within an instance method or constructor of a class, the "this" keyword is a reference to the current object.
- The current object is the object on which the method or constructor is being called.
- It allows you to access the instance variables and invoke other instance methods of the object.

### 2. Referring to the Invoking Object:

- When an instance method is called, the "this" keyword is automatically set to refer to the object that invoked the method.
- It provides a way to refer to the object from within the method, enabling access to its member variables and methods.

```

class Foo {
    int i = 5;
    static double k = 0;

    void setI(int i) {
        this.i = i;
    }

    static void setK(double k) {
        Foo.k = k;
    }
}

```

Suppose that f1 and f2 are two objects of Foo

Invoking f1.setI(10) is to execute  
 → f1.i = 10, where this is replaced by f1

Invoking f2.setI(45) is to execute  
 → f2.i = 45, where this is replaced by f2

2) In Java, the "this" keyword is a reference variable that refers to the current instance of a class. It is used to differentiate between instance variables and local variables or method

parameters that have the same name. The "this" keyword is primarily used in the following scenarios:

### To refer to instance variables:

When a local variable or parameter has the same name as an instance variable, the "this" keyword is used to explicitly refer to the instance variable. It helps to avoid naming conflicts and ensures that the correct instance variable is accessed within the class.

```

public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b){
        x = a;
        y = b;
    }
}

```

But it could have been written like this

```

public class Point {
    public int x = 0;
    public int y = 0;

    //constructor public
    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

```

### c) To invoke one constructor from another:

The "this" keyword can be used to invoke another constructor of the same class from within a constructor. It allows constructor chaining, where one constructor initializes common fields and then delegates to another constructor for further initialization.

```
public class Rectangle {  
    private int x, y;  
    private int width, height;  
    public Rectangle(){  
        this(0, 0, 0, 0);  
    }  
    public Rectangle(int width, int height){  
        this(0, 0, width, height);  
    }  
    public Rectangle(int x, int y, int width, int height){  
        this.x = x;  
        this.y = y;  
        this.width = width;  
        this.height = height;  
    }  
}
```

77

The no-argument constructor calls the four argument constructor with four 0 values , the two-argument constructor calls the four argument constructor with two 0 values. As before the compiler determines which constructor to call , based on the number and the type of arguments. And if present , the invocation of another constructor must be the first line in the constructor.

## CHAPTER THREE

### OOP PRINCIPLES

#### ENCAPSULATION AND DATA ABSTRACTION

Imagine you have a treasure chest with valuable items. You want to keep the items safe and secure, so you put them inside the treasure chest and lock it. You don't want anyone to directly access the items without your permission or knowledge.

In this example:

- The treasure chest represents a class in programming.
- The valuable items represent the data (attributes) inside the class.
- The act of putting the items inside the chest and locking it represents encapsulation.

The literal definition of encapsulation, in a general sense, is the act of enclosing or containing something within a capsule or container. In the context of object-oriented programming, encapsulation refers to the bundling of data (attributes) and methods (behaviors) together within a class and controlling access to them.

The term "encapsulation" in programming is derived from its literal meaning of enclosing or encapsulating related elements within a protective container, similar to how we encapsulate objects in real-life containers to protect them.

In programming, encapsulation allows us to group related data and behaviors together, providing a way to organize and structure code. By encapsulating data and methods within a class, we establish boundaries and control access to the internal workings of the class. This helps to safeguard the integrity of the data and provides a well-defined interface for interacting with the class.

The primary point of encapsulating data fields and methods together in a class is to promote proper information hiding and to provide a well-

defined interface for interacting with the class. Let's explore the benefits and reasons behind this:

#### Information Hiding:

- Encapsulating data fields and methods within a class allows you to hide the internal details of how the class works.
- By encapsulating data, you prevent direct access to the internal state of the object from outside the class.
- This helps protect the data integrity, ensuring that it can only be modified through controlled methods, thus preventing unintended or incorrect usage.
- Hiding the internal implementation details also reduces complexity and enhances code maintainability.

#### Modularity and Organization:

- Encapsulation promotes modular design by grouping related data fields and methods together within a class.
- It provides a logical and organized structure for your code, making it easier to understand, read, and maintain.
- With encapsulation, each class represents a distinct entity or concept, encapsulating its specific data and behaviors, which improves code organization and separation of concerns.

#### Access Control:

- Encapsulation allows you to control the visibility and access levels of data fields and methods.
- By using access modifiers (e.g., public, private, protected), you can restrict access to certain members of the class.
- This enables you to define a clear interface for interacting with the class, hiding unnecessary implementation details and exposing only the essential methods and attributes.



Let's take an example of a simple class called Dog:

```
java Copy code
public class Dog {
    private String name; // Private data field (attribute)

    public void setName(String dogName) { // Public method to set the name
        name = dogName;
    }

    public String getName() { // Public method to get the name
        return name;
    }
}
```

In this example, the Dog class has a private data field name that cannot be directly accessed from outside the class. We have encapsulated the name variable to ensure it is not modified without going through a controlled process.

To interact with the name attribute, the class provides two public methods: **setName()** and **getName()**. These methods act as an interface for accessing and modifying the name attribute.

For example, let's say we have an instance of the Dog class named myDog. To set the name of the dog, we use the setName() method, and to retrieve the name, we use the getName() method:

```
java Copy code
public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.setName("Buddy"); // Set the name using the public method
        System.out.println("My dog's name is: " + myDog.getName()); // Get
    }
}
```

In this code, we create an instance of the Dog class called myDog. We use the setName() method to set the name of the dog as "Buddy" and the getName() method to retrieve and print the name.

Encapsulation helps protect the internal state of an object and allows controlled access to its attributes and behaviors. By encapsulating data and methods within a class, we can ensure that they are used correctly and maintain the integrity and security of the object's internal workings.

## ENCAPSULATION IN ACHIEVING DATA ABSTRACTION

Encapsulation plays a crucial role in achieving data abstraction. Let's explore how encapsulation is used as a data abstraction process:

### Bundling Data and Methods:

- Encapsulation allows you to bundle related data fields (attributes) and methods (behaviors) together within a class.
- Data fields represent the state or properties of an object, while methods define the operations or behaviors that can be performed on the data.
- By encapsulating data and methods within a class, you establish a cohesive unit that represents an abstraction of a real-world entity or concept.

### Hiding Implementation Details:

- Encapsulation enables you to hide the internal implementation details of a class from external entities.
- The internal state of an object, represented by its data fields, is kept private or protected, preventing direct access from outside the class.
- This information hiding ensures that the internal implementation details of a class are not exposed, promoting a higher level of abstraction.

### Providing an Interface:

- Encapsulation provides a well-defined interface through which external entities can interact with the class.
- The class exposes a set of public methods, while keeping the internal implementation details hidden.
- The public methods act as an abstraction layer, allowing users to manipulate the class's data and invoke its behaviors without needing to know the underlying implementation details.

## Abstracting Complexity:

- By encapsulating complex data structures and algorithms within a class, encapsulation serves as a form of data abstraction.
- Users of the class only need to understand and interact with the simplified interface provided by the public methods.
- The complex details of how the data is stored, processed, or manipulated are abstracted away, making it easier for users to work with the class.

## Separating Interface from Implementation:

- Encapsulation separates the interface (public methods) from the implementation (internal details) of a class.
- This separation allows for better maintenance and evolution of the codebase.
- Changes made to the internal implementation details do not impact the external users of the class, as long as the interface remains unchanged.
- This separation of concerns enables abstraction by providing a clear distinction between how the class is used and how it internally accomplishes its tasks.

So encapsulation serves as a data abstraction process by bundling related data and methods together, hiding implementation details, and providing a well-defined interface. It allows users to interact with the class through a simplified and abstracted view, focusing on the essential aspects of the data and behaviors while hiding the underlying complexities. Encapsulation plays a vital role in achieving data abstraction and promotes modular, maintainable, and reusable code.

## SETTERS AND GETTERS

public methods, commonly known as getters and setters, serve as access points to the fields (variables) of a class from the external Java

world. Let's define this in more detail and provide examples:

### Getters:

- Getters are public methods that allow external entities to access the values of private fields (variables) of a class.
- Getters typically have the prefix "get" followed by the name of the field they provide access to.
- Getters provide read-only access to the values of the fields, allowing users to retrieve the current state of an object.
- By encapsulating the fields and providing getters, you can control how the values are accessed and potentially add additional logic or transformations if needed.

### Setters:

- Setters are public methods that allow external entities to modify or set the values of private fields of a class.
- Setters typically have the prefix "set" followed by the name of the field they modify.
- Setters provide write access to the fields, allowing users to update the state of an object.
- By encapsulating the fields and providing setters, you can control how the values are modified, validate inputs, or perform other operations when the field is updated.

### Example :-

```
java Copy code
public class EncapTest {
    private String name;
    private int age;

    // Getter for the 'name' field
    public String getName() {
        return name;
    }

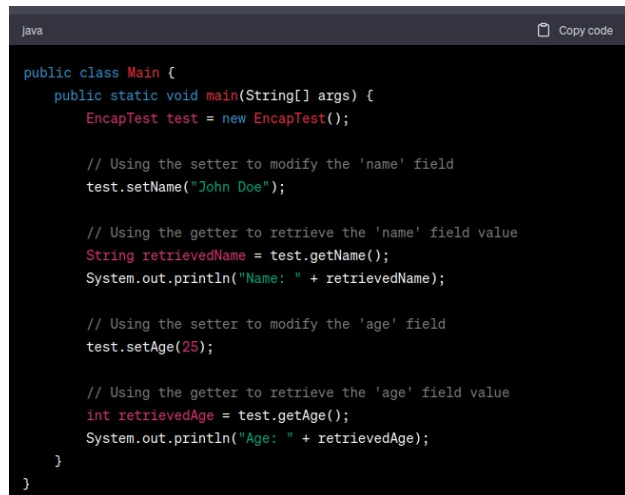
    // Setter for the 'name' field
    public void setName(String newName) {
        name = newName;
    }

    // Getter for the 'age' field
    public int getAge() {
        return age;
    }

    // Setter for the 'age' field
    public void setAge(int newAge) {
        if (newAge >= 0) {
            age = newAge;
        } else {
            System.out.println("Invalid age value. Age cannot be negative.");
        }
    }
}
```

In this example, the EncapTest class has two private fields: name and age. To access and modify these fields, public getters and setters are provided.

External entities can access the fields of the EncapTest class through these getters and setters. For example:



```
java Copy code

public class Main {
    public static void main(String[] args) {
        EncapTest test = new EncapTest();

        // Using the setter to modify the 'name' field
        test.setName("John Doe");

        // Using the getter to retrieve the 'name' field value
        String retrievedName = test.getName();
        System.out.println("Name: " + retrievedName);

        // Using the setter to modify the 'age' field
        test.setAge(25);

        // Using the getter to retrieve the 'age' field value
        int retrievedAge = test.getAge();
        System.out.println("Age: " + retrievedAge);
    }
}
```

In this code, we create an instance of the EncapTest class called test. We use the setter methods (setName() and setAge()) to modify the values of the fields, and we use the getter methods (getName() and getAge()) to retrieve the values.

By encapsulating the fields and providing getters and setters, external entities can access and modify the fields in a controlled manner. This allows for better data encapsulation, validation, and flexibility in managing the state of objects.

It's important to note that not all fields require getters and setters. Some fields may be intended for internal use only or may have more specific access requirements. The decision to provide getters and setters depends on the intended usage and the desired level of encapsulation and control over the fields.

## INHERITANCE IN JAVA

Inheritance is a fundamental concept in object-oriented programming that allows classes to inherit properties (fields and methods) from other classes. It enables the creation of hierarchical

relationships between classes, where a subclass inherits the characteristics of its superclass.

Inheritance promotes code reuse and enables the creation of more specialized classes based on existing classes. The superclass serves as a template or blueprint for creating subclasses, which can add new features or modify existing ones. Subclasses inherit the fields and methods of the superclass and can also introduce their own unique fields and methods.

The superclass is higher in the inheritance hierarchy and represents a more general or abstract concept, while subclasses are lower in the hierarchy and represent more specific or specialized concepts. Subclasses inherit the attributes and behaviors of their superclass and can further refine or specialize them as needed.

Inheritance supports the principle of "is-a" relationship, where a subclass is considered to be a specific type of the superclass. For example, a "Car" class can be a subclass of a more general "Vehicle" class, as a car is a specific type of vehicle.

In Java, inheritance is achieved using the extends keyword. A class can extend only one superclass, but multiple levels of inheritance can be created through a chain of subclasses.

### Private Methods Can Not be Overridden

In Java, an instance method can be overridden by a subclass if it is accessible, meaning it can be accessed and invoked by other classes or subclasses. However, a private method cannot be overridden because it is not accessible outside its own class.

When a method is declared as private in a class, it is intended to be used only within that class. Private methods are not visible or accessible to other classes, including subclasses. Therefore, it is not possible for a subclass to override a private method from its superclass since it cannot access or modify the private method.

This restriction ensures encapsulation and prevents subclasses from altering the behavior of private methods, maintaining the integrity and

consistency of the superclass. Overriding is applicable to methods that are accessible and meant to be overridden, such as public or protected methods.

## APPLICATIONS OF INHERITANCE

**Specialization:** Inheritance allows the creation of specialized classes or objects that inherit data or behavior from a more general or abstract class. The new class or object can add additional data fields or behaviors that are specific to its specialized nature. This enables the modeling of more specific concepts in the program and promotes a more organized and hierarchical structure in the codebase.

**Overriding:** One of the key features of inheritance is the ability to override methods in the subclass. This allows a subclass to provide its own implementation of a method that it inherits from its superclass. By overriding a method, the subclass can modify or extend the behavior defined in the superclass to suit its specific needs. This enables customization and flexibility in the behavior of objects based on their specific class types.

**Code re-use:** Inheritance promotes code reusability by allowing classes to inherit properties and behaviors from existing classes. When a class inherits from another class, it automatically gains access to all the fields and methods defined in the superclass. This eliminates the need to duplicate code and promotes a more modular and efficient development approach. By reusing existing code through inheritance, developers can save time and effort by leveraging the functionality and structure already implemented in the superclass.

## JAVA ONLY SUPPORTS A SINGLE INHERITANCE

In Java, single inheritance means that a class can only extend (inherit from) one superclass. This restriction ensures a clear and unambiguous hierarchy in the class structure. Consequently, it is not allowed for a class to extend multiple classes simultaneously. In the given example, the

statement `public class Dog extends Animal, Mammal{...}` is considered illegal because it attempts to extend both the `Animal` and `Mammal` classes at the same time.

To overcome the limitation of single inheritance, Java provides an alternative mechanism called interfaces. Interfaces allow a class to implement multiple interfaces, which define sets of methods that the class must implement. This enables a form of multiple inheritance where a class can inherit behavior from multiple sources through the implementation of different interfaces. However, it's important to note that interfaces only define method signatures and constants, and do not provide concrete implementations like classes do.

## THE INSTANCE OF OPERATOR

The `instanceof` operator in Java is used to determine whether an object is an instance of a particular class or implements a specific interface. It returns a boolean value indicating whether the object is of the specified type or a subtype of it.

The syntax of the `instanceof` operator is as follows:

### object instanceof Class

Here, `object` is the object being tested, and `Class` is the class or interface being checked against.

The `instanceof` operator evaluates to **true** if the object is an instance of the specified class or implements the specified interface. Otherwise, it evaluates to false.

```
public class Dog extends Mammal{
    public static void main(String args[]){
        Animal a = new Animal();
        Mammal m = new Mammal();
        Dog d = new Dog();
        System.out.println(m instanceof Animal);
        System.out.println(d instanceof Mammal);
        System.out.println(d instanceof Animal);
    }
}
```

This would produce  
following result:

```
true
true
true
```

## THE SUPER KEYWORD

In simpler terms, the super keyword in Java is used to refer to the immediate parent class of a subclass. It provides a way to access and call members (methods, variables, and constructors) of the superclass within the subclass.

When a class extends another class, the subclass inherits all the non-private members of the superclass. These inherited members can be accessed using the super keyword. Here's a breakdown of how super is used:

### 1. Accessing superclass members:

- **Methods:** If a subclass defines a method with the same name as a method in the superclass, the subclass can use super to refer to the superclass version of the method. This allows the subclass to invoke the superclass method implementation, providing the ability to extend or override the functionality.
- **Variables:** If a subclass and superclass have variables with the same name, the subclass can use super to access the superclass variable. This is useful when you want to distinguish between the superclass and subclass variables.

### 2. Calling superclass constructors:

- When a subclass is created, its constructor automatically invokes the constructor of the superclass using super(). This ensures that the initialization code defined in the superclass is executed before the subclass-specific code.

In summary, the super keyword is used to refer to the superclass and allows the subclass to access and call inherited members. It provides a way to invoke superclass methods, access superclass variables, and call superclass constructors. By using super, we can leverage and extend the functionality of the superclass within

the subclass, enabling code reuse and hierarchical relationships between classes.

```
java Copy code

class Vehicle {
    String brand;

    public Vehicle(String brand) {
        this.brand = brand;
    }

    public void displayInfo() {
        System.out.println("Brand: " + brand);
    }
}

class Car extends Vehicle {
    int maxSpeed;

    public Car(String brand, int maxSpeed) {
        super(brand); // Call the constructor of the superclass using super()
        this.maxSpeed = maxSpeed;
    }

    public void displayInfo() {
        super.displayInfo(); // Call the displayInfo() method of the superclass
        System.out.println("Max Speed: " + maxSpeed + " km/h");
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota", 200);
        car.displayInfo();
    }
}
```

In this example, we have two classes: Vehicle and Car. Car is a subclass of Vehicle. Here's how the super keyword is used:

### 1. Constructor usage:

- In the Vehicle class, the constructor takes a brand parameter. The Car class extends Vehicle and has its own constructor.
- Inside the Car constructor, super(brand) is used to invoke the constructor of the Vehicle class. This ensures that the brand parameter is passed to the Vehicle constructor for initialization.

### 2. Method usage:

- Both the Vehicle and Car classes have a displayInfo() method.
- Inside the Car class's displayInfo() method, super.displayInfo() is used to call the displayInfo() method of the superclass (Vehicle class). This allows the subclass to extend the



functionality of the superclass method.

- Additionally, the Car class adds its own functionality to the method by printing the maxSpeed specific to the Car class.

When the program is executed, the main() method creates an instance of the Car class. The displayInfo() method is called on the Car object. It first calls the displayInfo() method of the superclass using super.displayInfo(), which prints the brand. Then, it prints the maxSpeed specific to the Car class.

Brand: Toyota  
Max Speed: 200 km/h

In this example, the super keyword is used to access the superclass constructor and invoke the superclass method within the subclass, allowing for inheritance and extension of functionality.

## WRITING FINAL CLASSES AND FINAL METHODS

The final modifier in Java can be applied to classes, methods, and variables. When applied to a class, it indicates that the class cannot be extended or subclassed. When applied to a method, it indicates that the method cannot be overridden in any subclass. The final modifier can also be applied to variables, which makes them unchangeable or constant.

### Key points about final:

#### Final Class:

- A final class is one that cannot be extended or subclassed.
- It is typically used for classes that are intended to be secure, efficient, or provide specialized functionality that should not be altered.

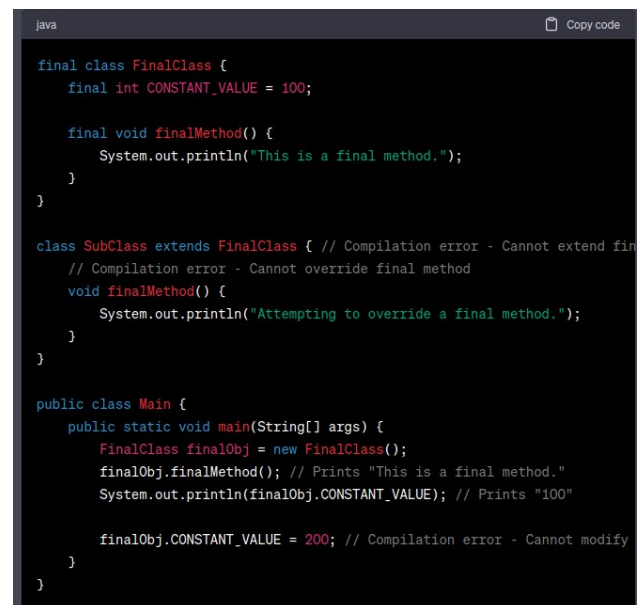
#### Final Method:

- A final method is one that cannot be overridden in any subclass.
- It is commonly used to prevent subclasses from altering the behavior of a crucial

method that is essential to the class's functionality or consistency.

#### Final Variables (Constants):

- A final variable is one that cannot be changed once it is assigned a value.
- It is often used to declare constants, ensuring that their values remain constant throughout the program.
- Final variables must be initialized either when they are declared or within a constructor.



```
java
Copy code

final class FinalClass {
    final int CONSTANT_VALUE = 100;

    final void finalMethod() {
        System.out.println("This is a final method.");
    }
}

class SubClass extends FinalClass { // Compilation error - Cannot extend final class
    // Compilation error - Cannot override final method
    void finalMethod() {
        System.out.println("Attempting to override a final method.");
    }
}

public class Main {
    public static void main(String[] args) {
        FinalClass finalObj = new FinalClass();
        finalObj.finalMethod(); // Prints "This is a final method."
        System.out.println(finalObj.CONSTANT_VALUE); // Prints "100"

        finalObj.CONSTANT_VALUE = 200; // Compilation error - Cannot modify final variable
    }
}
```

In the example above, the FinalClass is declared as final, so it cannot be extended by any subclass. The finalMethod() in the FinalClass is also declared as final, preventing any subclass from overriding it. The CONSTANT\_VALUE variable is declared as final, making it a constant that cannot be modified once assigned a value. When attempting to extend the FinalClass or override the finalMethod() in the SubClass, compilation errors occur due to the usage of the final modifier.

In Java, when a class is declared as final, it means that the class cannot be extended or subclassed. Additionally, all methods within a final class are implicitly marked as final, regardless of whether they are explicitly declared as final or not.

### METHODS THAT WE CAN'T OVERRIDE

The limitations and restrictions on declaring certain types of methods as abstract in Java.

#### Static Methods:

- Static methods belong to the class itself and are not associated with any particular instance of the class.
- Subclasses cannot override static methods because they are not tied to any instance and are invoked directly using the class name.
- Therefore, static methods cannot be declared as abstract because abstract methods are meant to be overridden by subclasses.

#### Private Methods:

- Private methods are not visible or accessible to subclasses.
- Since subclasses cannot access private methods, they cannot override them.
- Therefore, private methods cannot be declared as abstract because abstract methods are intended to be overridden by subclasses

#### Final Methods:

- Final methods are already considered non-overridable, meaning they cannot be overridden by subclasses.
- Final methods are used when we want to prevent any further modification or extension of the method's behavior.
- Making a final method abstract would be redundant because abstract methods are meant to be overridden, which contradicts the purpose of a final method.

- Therefore, final methods cannot be declared as abstract.

#### Final Class:

- A final class cannot be subclassed or extended.
- Since there would be no subclasses for a final class, there is no need to declare any abstract methods within it because there would be no classes to implement those abstract methods.
- Therefore, a final class cannot contain any abstract methods.

In summary, static, private, and final methods cannot be declared as abstract because they either cannot be overridden by subclasses, are not visible to subclasses, or are already considered non-overridable. Similarly, a final class cannot have abstract methods since it cannot be subclassed, and thus, there would be no subclasses to implement those abstract methods.

### METHOD OVERLOADING AND METHOD OVER RIDING

**Method overloading** is a feature in Java that allows multiple methods in a class to have the same name but with different parameters. It enables you to define multiple methods with the same name but different behavior based on the type, number, or order of the parameters.

#### Key points about method overloading:

##### Method Signature:

- Method signature includes the method name and the parameter list.
- Overloaded methods must have a different parameter list, which can differ in the number of parameters, types of parameters, or both.

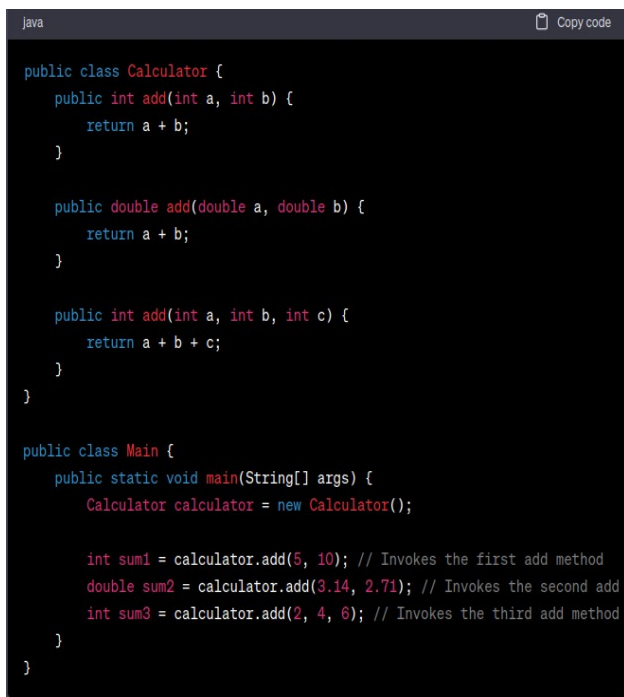
#### Differentiating Overloaded Methods:

- Java determines which overloaded method to invoke based on the arguments passed during the method call.

- The compiler matches the method call with the method signature that best matches the provided arguments.
- The return type and access modifiers of the methods do not play a role in method overloading.

### Benefits of Method Overloading:

- Provides a convenient way to create methods with different behavior based on the input parameters.
- Enhances code readability and maintainability by giving meaningful method names that reflect the intended functionality.



```

java Copy code
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double b) {
        return a + b;
    }

    public int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int sum1 = calculator.add(5, 10); // Invokes the first add method
        double sum2 = calculator.add(3.14, 2.71); // Invokes the second add
        int sum3 = calculator.add(2, 4, 6); // Invokes the third add method
    }
}

```

### Example of method overloading:

In the example above, the `Calculator` class demonstrates method overloading. It has three `add` methods with the same name but different parameter lists. The first `add` method adds two integers, the second `add` method adds two doubles, and the third `add` method adds three integers. During the method calls, Java determines the appropriate method to invoke based on the provided arguments.

**Method Overriding:** Method overriding occurs when a subclass defines a method with the same name, return type, and parameters as a method

in its superclass. It allows the subclass to provide its own implementation of the inherited method, thereby changing or extending its behavior.

### Key points about method overriding:

#### Inheritance Relationship:

- Method overriding is only possible in a subclass that extends a superclass.
- The subclass inherits the method from the superclass and provides its own implementation.

#### Method Signature:

- The overriding method in the subclass must have the same method signature (name, return type, and parameter list) as the method in the superclass.

#### Access Modifiers and Exceptions:

- The overriding method in the subclass must have the same or less restrictive access modifier than the method in the superclass.
- The overriding method can throw the same or narrower checked exceptions or no exceptions at all.

#### Method Invocation:

- When an overridden method is invoked using a subclass object, the subclass method implementation is executed instead of the superclass method.

### Example of method overriding

```

java Copy code

class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal = new Animal();
        animal.makeSound(); // Prints "Animal makes a sound"

        Dog dog = new Dog();
        dog.makeSound(); // Prints "Dog barks"
    }
}

```

In this example, the `Animal` class has a method `makeSound()`. The `Dog` class extends the `Animal` class and overrides the `makeSound()` method with its own implementation. When the `makeSound()` method is invoked using a `Dog` object, the overridden method in the `Dog` class is executed instead of the method in the `Animal` class. This allows the subclass to provide its own behavior while retaining the same method signature as the superclass.

## ABSTRACT CLASS AND METHODS

### ABSTRACT CLASS AND CONCRETE CLASS

An abstract class is a class that is declared with the `abstract` keyword. It serves as a blueprint for other classes and cannot be instantiated on its own. An abstract class can contain abstract methods (methods without a body) and also non-abstract methods (methods with an implementation). The purpose of an abstract class is to provide common functionality and behavior that can be shared by its subclasses.

Here are some key points about abstract classes:

#### 1. Abstract Class Declaration:

- An abstract class is declared using the `abstract` keyword before the class declaration.
- Syntax:** `abstract class ClassName { ... }`

#### 2. Abstract Methods:

- An abstract class can have abstract methods, which are declared without a body and end with a semicolon.
- Abstract methods serve as placeholders and are meant to be overridden by concrete (non-abstract) methods in the subclasses.
- Subclasses of an abstract class must implement all the abstract methods inherited from the abstract class.
- Syntax:** `abstract returnType methodName(parameters);`

#### 3. Concrete Class (Non-abstract Class):

- A concrete class is a regular class that can be instantiated and does not contain any abstract methods.
- It provides the implementation for all inherited abstract methods from its abstract superclass.
- Concrete classes can also have their own additional methods and attributes.
- Concrete classes can be instantiated directly using the `new` keyword.

## DEEP DOWN TO ABSTRACT CLASSES

In Java, the `abstract` keyword is used to declare both abstract classes and abstract methods. An abstract method is a method that is declared in a class but does not have an implementation in that class. Instead, its implementation is provided by the subclasses that inherit from the abstract class.

Here are some key points about abstract methods:

#### 1. Abstract Method Declaration:

- An abstract method is declared using the `abstract` keyword in the parent class.
- It does not have a method body, meaning it does not contain the code that defines the method's functionality.
- It ends with a semicolon (;) instead of curly braces ({}), as seen in regular methods.
- Syntax:** `abstract returnType methodName(parameters);`

## 2. Abstract Methods in Abstract Classes:

- Abstract methods are typically declared in abstract classes, which serve as blueprints for other classes.
- An abstract class can have one or more abstract methods along with regular (non-abstract) methods.
- Subclasses that extend an abstract class must implement all the abstract methods inherited from the abstract class.

## 3. Implementation in Subclasses:

- Subclasses that inherit an abstract method from an abstract class must provide an implementation for that method.

- The implementation of the abstract method in the subclass must have the same method signature as declared in the abstract class.
- The purpose of abstract methods is to define a contract or a required behavior that the subclass must fulfill.

### Example of an abstract class with an abstract method :-

In this example, the abstract class `Animal` defines a common structure for different types of animals. It has an abstract method `makeSound()` that represents the unique sound each animal makes. The `Animal` class also has a non-abstract method `sleep()` that provides a common behavior for all animals.

The `Dog` and `Cat` classes extend the `Animal` class and provide their own implementations of the `makeSound()` method. Each subclass has its own specific sound and can also invoke the `sleep()` method inherited from the `Animal` class.

In the `Main` class, we create instances of `Dog` and `Cat`, and invoke the `makeSound()` and `sleep()` methods specific to each subclass. Although there is no explicit polymorphism in this example, the abstract class provides a shared structure and behavior that can be extended and implemented by its subclasses.

## INTERFACE IN JAVA

An interface in Java is a reference type that defines a contract or a set of methods that a class must implement. It serves as a blueprint for classes, specifying the methods that the implementing classes must provide. Interfaces can also define constants (static final variables) and can be used to achieve multiple inheritance in Java.

### Key points about interfaces:

#### Interface Declaration:

- An interface is declared using the interface keyword in Java.

```
java Copy code

abstract class Animal {
    protected String name;

    public Animal(String name) {
        this.name = name;
    }

    public abstract void makeSound();

    public void sleep() {
        System.out.println(name + " is sleeping.");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        super(name);
    }

    public void makeSound() {
        System.out.println(name + " barks.");
    }
}

class Cat extends Animal {
    public Cat(String name) {
        super(name);
    }

    public void makeSound() {
        System.out.println(name + " meows.");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog("Buddy");
        dog.makeSound(); // Prints "Buddy barks."
        dog.sleep(); // Prints "Buddy is sleeping."

        Cat cat = new Cat("Whiskers");
        cat.makeSound(); // Prints "Whiskers meows."
        cat.sleep(); // Prints "Whiskers is sleeping."
    }
}
```



- The syntax for declaring an interface is as follows:

```
java Copy code  
  
accessModifier interface InterfaceName {  
    // Constants  
    // Method signatures  
}
```

The access modifier for an interface can be public or no modifier (which means it is accessible within the package).

### Interface Methods:

- An interface contains method signatures without any method bodies.
- All methods declared in an interface are by default public and abstract.
- Implementing classes must provide the implementation (method body) for all methods defined in the interface.

### Interface Constants:

- Interfaces can define constants, which are static final variables.
- Constants in interfaces are implicitly public, static, and final, and can be accessed using the interface name.
- These constants represent values that are shared among all implementing classes.

### Implementing Interfaces:

- To implement an interface, a class uses the implements keyword, followed by the interface name.
- A class can implement multiple interfaces, separated by commas.
- The implementing class must provide an implementation for all methods defined in the interface(s).

**Implicitly Abstract:** In Java, every method declared in an interface is implicitly abstract. This means that the method declaration does not contain an implementation (method body) but only the method signature, which consists of the method name, parameter list, and return type. By declaring methods as abstract in an interface, it

indicates that the actual implementation of these methods will be provided by the classes that implement the interface. Abstract methods serve as a contract or a blueprint for the classes implementing the interface, specifying what methods they should define. This allows for consistency and ensures that all implementing classes provide their own implementation of the declared methods.

**Implicitly Public:** Methods in an interface are implicitly public, which means they can be accessed and invoked from anywhere within the program. The public access modifier grants the highest level of visibility to the methods. By default, interface methods are accessible to all other classes and can be called using the interface reference. This accessibility allows objects of different classes implementing the same interface to interact with each other through the common interface methods. The public access modifier ensures that the methods defined in the interface can be utilized by other parts of the program without any restrictions, enabling proper interaction and interoperability between classes and interfaces.

### Concrete interface implements multiple interfaces

In Java, a concrete class refers to a class that provides a complete implementation of all the methods declared in its own class definition as well as any interfaces it implements. By implementing one or more interfaces, a concrete class agrees to fulfill the contract defined by those interfaces and must provide the implementation code for all the methods declared in those interfaces. This allows the concrete class to inherit the abstract methods from the interfaces and provide specific functionality for each of them. By implementing interfaces, the concrete class gains the ability to be treated as an instance of those interfaces, enabling it to be used interchangeably with other objects that implement the same interfaces. This promotes code reusability, modular design, and enhances the flexibility and versatility of the class by adhering to a predefined set of behaviors specified by the interfaces it implements.

```

java
Copy code

interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Bird implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Bird is flying");
    }

    public void swim() {
        System.out.println("Bird is swimming");
    }
}

public class Main {
    public static void main(String[] args) {
        Bird bird = new Bird();
        bird.fly();
        bird.swim();
    }
}

```

**Inability to Instantiate an Interface:** An interface in Java cannot be directly instantiated, meaning you cannot create objects of an interface type using the **new** keyword. Interfaces act as contracts or specifications for classes, defining a set of methods that implementing classes must provide. Therefore, interfaces are meant to be implemented by classes rather than instantiated on their own.

**Absence of Constructors in Interfaces:** Unlike classes, **interfaces do not have constructors**. Constructors are special methods used to initialize objects of a class. **Since interfaces cannot be instantiated, there is no need for constructors in interfaces**. Interfaces focus solely on defining method signatures and do not deal with object instantiation or initialization.

**All Methods in an Interface are Abstract:** Every method declared in an interface is implicitly abstract. **Abstract methods are methods that are declared without an implementation**. They provide a blueprint or contract that implementing classes must adhere to by providing their own implementation for each method. By default, all methods in an interface lack a method body and are meant to be overridden in the implementing classes.

**Absence of Instance Fields in Interfaces:** Interfaces **cannot contain instance fields**, which are

variables that hold data specific to each instance of a class. Unlike classes, which can have instance fields to store and manage data, interfaces focus on defining behavior through methods rather than managing state through fields. Interfaces are primarily concerned with providing a common set of methods that implementing classes must support.

**Implementation of Interfaces by Classes:** In Java, **a class implements an interface, rather than extending it as with class inheritance**. This means that a class declares its intent to fulfill the contract defined by an interface and provides the implementation for all the methods declared in that interface. By implementing an interface, a class ensures that it provides the necessary functionality required by the interface.

**Interface Extension:** **An interface in Java can extend one or more other interfaces**. This allows for the creation of more specialized interfaces that inherit and combine the methods from multiple parent interfaces. Interface extension enables the organization and hierarchy of interfaces, allowing for greater modularity and code reuse in a flexible and scalable manner. Implementing a derived interface will require implementing all the methods from both the derived interface and its parent interfaces.

```

java
Copy code

interface Shape {
    void draw();
}

interface Colorable {
    void setColor(String color);
}

interface Drawable extends Shape, Colorable {
    void setCoordinates(int x, int y);
}

class Circle implements Drawable {
    private String color;
    private int x, y;

    public void draw() {
        System.out.println("Drawing a circle");
    }

    public void setColor(String color) {
        this.color = color;
    }

    public void setCoordinates(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void printDetails() {
        System.out.println("Color: " + color);
        System.out.println("Coordinates: (" + x + ", " + y + ")");
    }
}

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.draw();
        circle.setColor("Red");
        circle.setCoordinates(10, 20);
        circle.printDetails();
    }
}

```

**STATIC FINAL DATA FIELDS** In Java, the data fields declared in an interface are implicitly static and final. This means that they are considered constant values and cannot be modified by any class, including subclasses that implement the interface. Subclasses are not allowed to modify the values of interface data fields because they are meant to represent constants that define the behavior or characteristics shared by all classes implementing the interface.

When a subclass implements an interface, it inherits the data fields of the interface as static and final variables. These variables are accessible to the subclass, but they cannot be modified or

overridden. They are treated as constants and their values remain the same across all instances of the subclass. This ensures consistency and prevents unintended changes to the predefined behavior defined by the interface.

If a subclass needs to have different values for the data fields, it can declare its own instance variables and provide its own implementation logic. The subclass can define its own data fields with different values or behaviors specific to that subclass. By doing so, the subclass can extend the functionality of the interface and introduce its own unique characteristics while still fulfilling the contract of the interface.

```

java
Copy code

interface Shape {
    double PI = 3.14159;
    String DESCRIPTION = "This is a shape.";

    void draw();
}

class Circle implements Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public void draw() {
        System.out.println("Drawing a circle with radius: " + radius);
    }

    public void printDescription() {
        System.out.println("Description: " + DESCRIPTION); // Accessing inte
    }
}

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5.0);
        circle.draw();
        circle.printDescription();
        System.out.println("Value of PI: " + Shape.PI); // Accessing interfa
    }
}

```

**Multiple Interfaces :-** In Java, a class can implement multiple interfaces, which allows it to inherit and utilize the behavior and contracts defined by multiple interfaces. This is Java's form of achieving multiple inheritance, where a class can inherit characteristics from multiple sources without directly extending multiple classes. By implementing multiple interfaces, a class gains the ability to provide implementations for all the methods defined in each interface it implements. This allows for greater flexibility and re-usability in code design, as different interfaces can be

combined to provide specific functionality and behavior to a class. This feature enables Java programmers to achieve the benefits of multiple inheritance while avoiding some of the complexities and issues associated with traditional multiple inheritance in other programming languages.

## POLYMORPHISM

Polymorphism in simpler terms can be understood as having the ability of an object to take on many forms. Just like a toy that can be transformed into different shapes, polymorphism allows an object to behave differently depending on the context in which it is used.

Imagine you have a toy car that can be a racing car, a police car, or a fire truck, depending on how you want to play with it. Similarly, in programming, polymorphism allows an object to act differently based on its specific type or the way it is used.

### Programming in the General:

- Polymorphism allows you to write programs in a more general manner, rather than dealing with specific types or implementations.
- Instead of writing code that explicitly handles each specific object type, polymorphism allows you to treat objects of different types as if they were objects of a common superclass.
- This general approach simplifies programming by enabling you to work with a shared set of behaviors and properties defined in the superclass.

### Shared Superclass and Subclasses:

- Polymorphism involves a shared superclass (Animal) and subclasses (Fish, Frog, and Bird) representing different animal types.
- Each subclass extends the common superclass, inheriting its properties and behaviors.

- The superclass (Animal) contains a method called move and maintains the current location of an animal using x-y coordinates.
- Each subclass (Fish, Frog, and Bird) implements its own version of the move method, defining how each animal type moves.

### Simulating Animal Movements:

- The program maintains an array of Animal references that can hold objects of different animal types (Fish, Frog, or Bird).
- To simulate the animals' movements, the program sends each object the same message (method call) called move once per second.
- Each specific animal object responds to the move message in its own way, implementing the movement appropriate for that type of animal.
- For example, a Fish might swim three feet, a Frog might jump five feet, and a Bird might fly ten feet.
- Each animal object knows how to modify its x-y coordinates accordingly based on its specific type of movement.

### Polymorphism and Message Handling:

- Polymorphism revolves around the concept that each object knows how to respond to the same message (method call) in its own way.
- Regardless of the specific animal type, the program sends the move message to each object.
- The appropriate response to the move message is determined by the specific implementation of the move method in each subclass.
- This ability to rely on each object to perform the appropriate action based on its type is the essence of polymorphism.
- The same message (move) sent to different objects results in different forms of behavior, demonstrating polymorphism.

polymorphism involves a shared superclass and subclasses where each subclass implements its own version of methods. In the context of simulating animal movements, polymorphism allows objects of different animal types to respond to the same message (move) in their own unique ways. Each object's specific implementation of the method determines its behavior. This flexibility and adaptability of objects to handle the same message differently is the essence of polymorphism.

#### Example :-

```
java Copy code

class Animal {
    public void makeNoise() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Bark");
    }
}

class Cat extends Animal {
    public void makeNoise() {
        System.out.println("Meawoo");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal a1 = new Cat();
        a1.makeNoise(); // Prints "Meawoo"

        Animal a2 = new Dog();
        a2.makeNoise(); // Prints "Bark"

        Animal a3 = new Animal();
        a3.makeNoise(); // Prints "Some sound"
    }
}
```

#### Class Hierarchy:

- There is a superclass called Animal that defines a method makeNoise().
- There are two subclasses, Dog and Cat, which extend the Animal class and override the makeNoise() method with their own specific implementations.

#### Polymorphic Reference:

- In the Demo class, polymorphism is demonstrated by creating instances of the Animal class and its subclasses and storing them in variables of type Animal.

- This allows the variables a1, a2, and a3 to hold references to different types of objects (Cat, Dog, and Animal) but treated as instances of the Animal class.

#### Method Invocation:

- The makeNoise() method is invoked on each variable of type Animal.
- Due to polymorphism, the actual implementation of the method called depends on the type of the object being referred to by the variable.

#### Polymorphic Behavior:

- When a1.makeNoise() is called, it prints "Meawoo" because a1 holds a reference to an object of the Cat class, which overrides the makeNoise() method.
- When a2.makeNoise() is called, it prints "Bark" because a2 holds a reference to an object of the Dog class, which also overrides the makeNoise() method.
- When a3.makeNoise() is called, it prints "Some sound" because a3 holds a reference to an object of the Animal class, which uses the default implementation of the makeNoise() method.

In summary, the code demonstrates polymorphism by allowing different objects (Cat and Dog) to be referred to using a common superclass type (Animal). The actual behavior of the makeNoise() method depends on the type of the object being referenced. This flexibility and dynamic behavior of method invocation based on the object's type is the essence of polymorphism.

#### IMPLEMENTING FOR EXTENSIBILITY

- **Extensibility and Polymorphism:**
  - Polymorphism allows for the design and implementation of systems that are easily extensible.
  - Extensibility refers to the ability to add new classes or functionality to an existing system without requiring



significant modifications to the existing codebase.

- With polymorphism, new classes can be added to the system by following the inheritance hierarchy already defined, with minimal changes to the general portions of the program.

- **Plug and Play:**

- New classes that are part of the existing inheritance hierarchy can "plug right in" to the program.
- The existing code, which processes objects generically based on their superclass type, does not need to be modified.
- The new classes inherit the common behaviors and properties defined in the superclass, allowing them to be processed in a generic manner.

- **Minimizing Modifications:**

- The parts of the program that require direct knowledge of the new classes being added to the hierarchy need to be altered.
- For example, if a class Tortoise is added to extend the existing Animal class, and it responds to a move message by crawling one inch, only the Tortoise class and the specific parts of the program instantiating Tortoise objects need to be written or modified.
- The portions of the program that work with the Animal superclass and invoke generic behaviors, such as calling the move method, can remain unchanged.

- **Is-a Relationship and Casting:**

- The is-a relationship holds between a subclass and its superclasses, allowing a subclass object to be treated as an instance of its superclass.

- However, treating a superclass object as a subclass object is not possible because a superclass object is not an object of any of its subclasses.
- To invoke subclass-specific methods on a superclass object, the superclass reference can be explicitly cast to the subclass type using a technique called downcasting.
- Down casting allows the program to access and invoke the methods specific to the subclass that are not present in the superclass.
- It is important to note that attempting to invoke subclass-specific methods through a superclass reference without proper casting results in compilation errors.

Implementing polymorphism for extensibility allows for easy addition of new classes to a system without significant modifications to the existing code base. Polymorphism enables the generic processing of objects based on their superclass type, allowing new classes to plug into the program seamlessly. Only the parts of the program that require direct knowledge of the new classes need to be modified. The is-a relationship between subclasses and superclasses allows for treating subclass objects as instances of their superclass. Proper casting (downcasting) is necessary to invoke subclass-specific methods on superclass objects.

### **SUPER CLASS SUB CLASS VARIABLES**

There are three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straightforward, we assign a superclass reference to a superclass variable, and a subclass reference to a subclass variable. Then we demonstrate the relationship between subclasses and superclasses (i.e., the is-a relationship) by assigning a subclass reference to a superclass variable. The Java compiler allows this "crossover" because an object of a subclass is an object of its superclass (but not vice versa). When the compiler encounters a method call made through a variable, the

compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. **At execution time, the type of the object to which the variable refers determines the actual method to use. This process, called dynamic binding.**

#### Assigning a superclass reference to a superclass variable:

- This is a straightforward assignment where we create an object of the superclass and assign it to a variable of the same type.
- The variable can access and invoke methods defined in the superclass.
- Since the variable is of the superclass type, it cannot access subclass-specific methods or attributes.

```
java Copy code

class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

Animal animal = new Animal(); // Create an Animal object
animal.eat(); // Invoke the eat() method defined in the Animal class
```

#### Assigning a subclass reference to a subclass variable:

- Similarly, this is a straightforward assignment where we create an object of the subclass and assign it to a variable of the same type (subclass type).
- The variable can access and invoke methods defined in both the superclass and the subclass.
- This assignment allows full access to all methods and attributes specific to the subclass.

```
java Copy code

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking.");
    }
}

Dog dog = new Dog(); // Create a Dog object
dog.eat(); // Invoke the eat() method inherited from the Animal class
dog.bark(); // Invoke the bark() method defined in the Dog class
```

#### Assigning a subclass reference to a superclass variable (is-a relationship):

- This demonstrates the relationship between subclasses and superclasses, known as the "is-a" relationship.
- We create an object of the subclass and assign it to a variable of the superclass type.
- The Java compiler allows this assignment because an object of a subclass is also an object of its superclass.
- The variable can access and invoke methods defined in the superclass, but it cannot access methods specific to the subclass (unless proper casting is performed).
- At runtime, the actual type of the object to which the variable refers determines which specific method implementation is used. This is known as dynamic binding or late binding.

```
java Copy code

class Animal {
    public void makeNoise() {
        System.out.println("Animal is making a noise.");
    }
}

class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Dog is barking.");
    }
}

class Cat extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Cat is meowing.");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal animal1 = new Dog(); // Create a Dog object and assign it to
        Animal animal2 = new Cat(); // Create a Cat object and assign it to

        animal1.makeNoise(); // Prints "Dog is barking." - Dynamic binding
        animal2.makeNoise(); // Prints "Cat is meowing." - Dynamic binding
    }
}
```

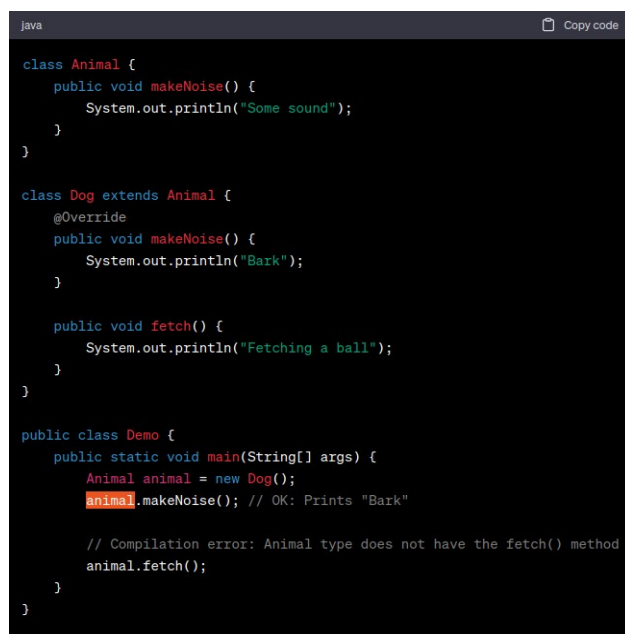
In this example, we have a superclass `Animal` and two subclasses `Dog` and `Cat`, each with their own implementation of the `makeNoise()` method.

Inside the main() method, we create objects of the Dog and Cat classes and assign them to variables of the Animal type. The variables animal1 and animal2 are declared as Animal type, but they actually reference objects of the Dog and Cat classes, respectively.

When we invoke the makeNoise() method on animal1 and animal2, the actual implementation of the method that is executed depends on the type of the object being referenced at runtime. This is determined dynamically based on the actual type of the object, rather than the declared type of the reference variable.

Therefore, at runtime, the animal1 reference points to a Dog object, so the overridden makeNoise() method in the Dog class is invoked, and it prints "Dog is barking." Similarly, the animal2 reference points to a Cat object, so the overridden makeNoise() method in the Cat class is invoked, and it prints "Cat is meowing."

It is important to note that attempting to invoke subclass-specific methods through a superclass reference without proper casting results in compilation errors.



```
java Copy code
class Animal {
    public void makeNoise() {
        System.out.println("Some sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeNoise() {
        System.out.println("Bark");
    }

    public void fetch() {
        System.out.println("Fetching a ball");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal animal = new Dog();
        animal.makeNoise(); // OK: Prints "Bark"

        // Compilation error: Animal type does not have the fetch() method
        animal.fetch();
    }
}
```

In this example, we have a superclass Animal and a subclass Dog. The Dog class extends the Animal class and overrides the makeNoise() method with

its own implementation. Additionally, the Dog class introduces a new method called fetch().

In the Demo class, we create an instance of Dog and assign it to a variable of type Animal named animal. Since Dog is a subclass of Animal, this assignment is allowed.

When we invoke the makeNoise() method on animal, it executes the makeNoise() implementation from the Dog class, which prints "Bark". This works fine because the makeNoise() method is inherited from the superclass.

However, when we attempt to invoke the fetch() method on animal, which is a subclass-specific method, a compilation error occurs. The reason is that the variable animal is of type Animal, and the Animal class does not have a fetch() method. The compiler detects this mismatch and throws a compilation error.

## UPCASTING AND DOWNCASTING

Casting in Java refers to the process of converting a value from one data type to another. It allows you to treat an object or value as another type temporarily, either to perform operations or to assign it to a different type of variable.

### Upcasting:

- Upcasting refers to the casting of a reference from a subclass type to a superclass type.
- It is done implicitly and does not require an explicit cast. (Animal animal = new Dog ())
- Upcasting is safe because a subclass object is also an instance of its superclass.
- Upcasting allows you to treat an object of the subclass as an object of the superclass, providing a more generalized view.

**Example of up casting :-**

```

java
Copy code

class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking.");
    }
}

public class Demo {
    public static void main(String[] args) {
        Dog dog = new Dog(); // Create a Dog object
        Animal animal = dog; // Upcasting: Assigning Dog object to an Animal

        animal.eat(); // Invoke the eat() method from Animal class (inherited)
        // Compilation error: The bark() method is not accessible from the Animal class
        animal.bark();
    }
}

```

In this example, we create a Dog object and assign it to a Dog reference variable named dog. Then, we perform upcasting by assigning the dog reference to an Animal reference variable named animal. This upcasting allows us to treat the dog object as an Animal object.

When we invoke the eat() method on the animal reference, it calls the eat() method from the Animal class. However, when we try to invoke the bark() method on the animal reference, a compilation error occurs because the bark() method is specific to the Dog class and is not accessible through the Animal reference.

## Downcasting:

- Downcasting refers to the casting of a reference from a superclass type to a subclass type.
- It is done explicitly by specifying the desired subclass type in parentheses before the reference variable.
- Downcasting is required when you want to access subclass-specific methods or attributes that are not present in the superclass.

```

java
Copy code

class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking.");
    }
}

public class Demo {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Create a Dog object and assign it to a Animal reference

        // Downcasting: Casting the Animal reference to a Dog reference
        Dog dog = (Dog) animal;

        dog.eat(); // Invoke the eat() method from Animal class (inherited)
        dog.bark(); // Invoke the bark() method from Dog class (specific to Dog)
    }
}

```

- Downcasting should be used with caution because it may result in a ClassCastException at runtime if the reference variable does not actually refer to an object of the specified subclass type.

## Example of down casting :-

In this example, we create an Animal object and assign it to an Animal reference variable named animal. Then, we perform downcasting by explicitly casting the animal reference to a Dog reference variable named dog. This allows us to treat the animal object as a Dog object.

After downcasting, we can invoke both the inherited eat() method from the Animal class and the specific bark() method from the Dog class using the dog reference.

However, it's important to note that downcasting should only be done when you are certain that the referenced object is actually of the desired subclass type. Otherwise, it may result in a ClassCastException at runtime.

## PACKAGES IN JAVA

In Java, packages are used to organize classes and interfaces into modular and hierarchical structures. A package provides a way to group related classes, making it easier to manage and maintain large-scale applications. It helps in avoiding naming conflicts between classes and provides better code organization and reusability.

Here are some key points about packages in Java:

### Package Declaration:

- A package is declared at the beginning of a Java source file using the package keyword followed by the package name.
- The package declaration should be the first non-comment line in the source file.
- The package name should follow the Java naming conventions and usually follows a hierarchical structure using dot notation.

### Package Structure:

- Packages form a hierarchical structure, similar to folders in a file system.
- The package hierarchy starts with the top-level package, also known as the root package. All other packages are placed under the root package.
- Packages can contain sub-packages, forming a nested structure.
- The package structure is reflected in the directory structure of the file system, where each package corresponds to a directory.

### Package Import:

- To use classes from another package in your code, you need to import them using the import statement.
- The import statement is placed after the package declaration and before the class declaration.
- It allows you to refer to the imported classes directly by their simple names, without specifying the full package name every time.

### Access Control:

- Packages also provide a way to control the access to classes and members.
- Java provides access modifiers like public, private, protected, and default (no modifier) to specify the accessibility of classes and members within a package.
- Classes and members declared with default access are accessible only within the same package.

## PACKAGE DECLARATION AND PACKAGE IMPORT

Package Declaration and Package Import are two distinct concepts in Java. Here are the differences between them:

### Package Declaration:

- The package declaration is used to define the package to which a Java source file belongs.

- It is placed at the top of the source file and is the first non-comment line.
- **Syntax:** `package packageName;`
- It specifies the package name and associates the source file with that package.
- The package name follows the Java naming conventions and is typically a hierarchical structure using dot notation.
- Example: `package com.example.util;`

### Package Import:

- Package import is used to import classes or entire packages from another package into the current Java source file.
- It allows you to refer to the imported classes directly by their simple names, without specifying the full package name every time.
- **Syntax:** `import packageName.ClassName;` or `import packageName.*;`
- It is placed after the package declaration and before the class declaration.
- It specifies the classes or packages to be imported for use in the current source file.
- Importing specific classes from a package provides more control over which classes are imported, whereas importing the entire package using `*` imports all classes in that package.
- Example: `import java.util.ArrayList;` or `import java.util.*;`

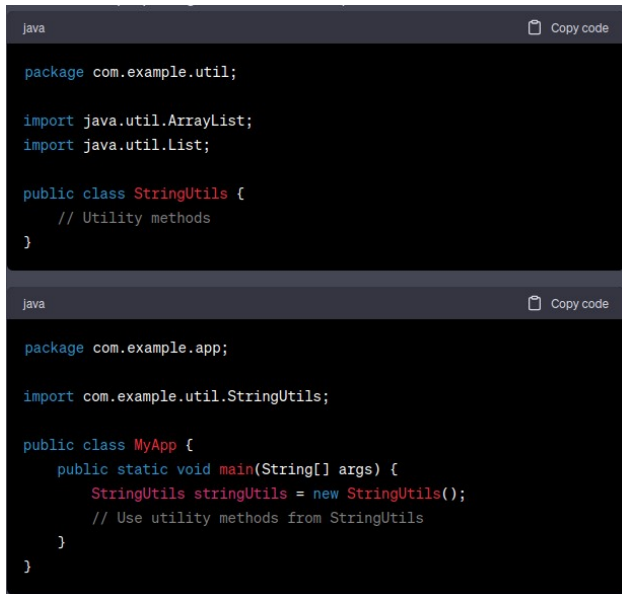
### Package Examples:

- Let's consider an example where you have two packages: `com.example.util` and `com.example.app`.
- The `com.example.util` package can contain utility classes and helper methods that can be shared across different parts of the application.



- The com.example.app package can contain classes specific to the application logic.

Here's an example package declaration and import statement:



```
java Copy code

package com.example.util;

import java.util.ArrayList;
import java.util.List;

public class StringUtils {
    // Utility methods
}

java Copy code

package com.example.app;

import com.example.util.StringUtils;

public class MyApp {
    public static void main(String[] args) {
        StringUtils stringUtils = new StringUtils();
        // Use utility methods from StringUtils
    }
}
```

In the above example, the StringUtils class in the com.example.util package is imported into the MyApp class in the com.example.app package using the import statement.

The classes in the com.example.app package can access the StringUtils class directly by its simple name without specifying the full package name.

Packages play a crucial role in organizing and managing code in Java applications. They promote modularity, code reusability, and reduce naming conflicts. They also enable access control and provide a clear structure for better code organization.

## CHAPTER FOUR EXCEPTION HANDLING

Exception handling in Java is a mechanism that allows programmers to gracefully handle and recover from runtime errors or exceptional situations that may occur during the execution of a program. It helps prevent the program from crashing or terminating abruptly when an error

occurs, and provides a way to handle such errors in a controlled manner.

In Java, exceptions are represented by classes that inherit from the `java.lang.Exception` class. When an exceptional situation occurs, such as division by zero, array index out of bounds, or file not found, an exception object is created and thrown. The program then looks for an appropriate exception handler to catch and handle the exception.

The basic structure of exception handling in Java involves the following keywords:

1. **try:** This block encloses the code that may potentially throw an exception. It is followed by one or more catch blocks or a finally block.
2. **catch:** This block is used to catch and handle specific exceptions that occur within the corresponding try block. It specifies the type of exception to catch and provides code to handle the exception. Multiple catch blocks can be used to handle different types of exceptions.
3. **finally:** This block is optional and is used to specify code that should be executed regardless of whether an exception occurs or not. It is often used to release resources or perform cleanup operations.

When an exception is thrown, the Java runtime system searches for a suitable catch block that can handle the exception based on the type of exception thrown. If a matching catch block is found, the corresponding code is executed. If no suitable catch block is found, the exception is propagated up the call stack until it is caught or the program terminates.

Here's a basic example that demonstrates exception handling in Java :-

In the above example, an `ArithmeticException` is thrown when dividing by zero. The exception is caught in the catch block, where the error message is printed. Finally, the finally block is executed regardless of whether an exception occurred or not.

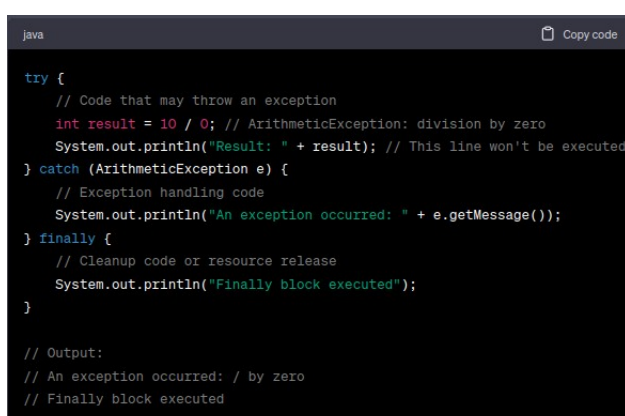
By effectively using exception handling, developers can make their programs more robust, handle unexpected situations gracefully, and provide appropriate feedback to users or take corrective actions when errors occur.

## TYPES OF ERRORS

**Syntax errors:** These errors occur when the rules of the programming language are not followed. It's like using incorrect grammar in a sentence. These errors are detected by the compiler, which is a program that checks the code for mistakes before it is executed. When a syntax error is found, the compiler shows an error message indicating the line or area where the error occurred. To fix these errors, you need to correct the syntax by following the rules of the language.

**Runtime errors:** These errors happen while the program is running. They occur when the program tries to do something that is not possible or allowed. It's like encountering a roadblock or obstacle during a journey. These errors are detected by the environment (such as the Java Virtual Machine) while executing the program. For example, if you try to divide a number by zero or access an array element that doesn't exist, a runtime error occurs. When a runtime error happens, the program may crash or terminate abruptly. To handle runtime errors gracefully, we use exception handling.

**Logic errors:** These errors occur when the program doesn't work as intended, even though it may be free of syntax and runtime errors. It's like writing a sentence that conveys the wrong meaning. Logic errors are harder to detect because they don't produce error messages or



```
java
Copy code

try {
    // Code that may throw an exception
    int result = 10 / 0; // ArithmeticException: division by zero
    System.out.println("Result: " + result); // This line won't be executed
} catch (ArithmeticException e) {
    // Exception handling code
    System.out.println("An exception occurred: " + e.getMessage());
} finally {
    // Cleanup code or resource release
    System.out.println("Finally block executed");
}

// Output:
// An exception occurred: / by zero
// Finally block executed
```

crash the program. Instead, they cause the program to produce incorrect or unexpected results. These errors can happen if the program's logic or reasoning is flawed. Fixing logic errors requires carefully reviewing the program's code and identifying and correcting the flawed logic.

In the given context, the focus is on using exception handling to deal with runtime errors specifically. Exception handling is a mechanism that allows programmers to handle and recover from runtime errors in a controlled manner. Instead of letting the program crash or terminate abruptly, exception handling provides a way to catch and handle these errors. By using exception handling, programmers can write code to handle specific types of runtime errors and provide appropriate feedback or take corrective actions when such errors occur during program execution. This helps make the program more robust and prevents unexpected crashes, improving the overall reliability of the software.

**Here's a simpler explanation of exceptions and the reasons they can occur:**

An exception is like a problem that happens while a program is running. It's something unexpected that causes the program to encounter a roadblock or difficulty.

Exceptions can occur for different reasons. Here are a couple of examples:

1. **Attempting to divide by zero:** Imagine you have a calculation in your program where you need to divide a number by zero. This is not possible because dividing by zero doesn't have a valid mathematical result. So, when the program tries to do this division, it encounters an exception called an "arithmetic exception." It's like trying to divide a pizza into zero slices—it just doesn't make sense!
2. **Reading a decimal value when an integer is expected:** Let's say your program expects the user to enter a whole number, but instead, the user enters a number with a decimal point. This can cause an exception called a "number format

exception" because the program was expecting an integer (a whole number) but received something different. It's like expecting to receive an apple, but someone hands you a banana instead—it doesn't match what you were expecting!

In both cases, these exceptions occur because the program encounters a situation that it doesn't know how to handle or proceed with. Exceptions provide a way for the program to communicate this problem to the developer or handle it gracefully, rather than crashing or producing incorrect results.

By understanding the specific reasons for exceptions and using exception handling techniques, programmers can write code that can handle these unexpected situations and ensure that the program continues to run smoothly and provide appropriate feedback when such problems occur.

#### **SOME OF THESE EXCEPTIONS CAUSED BY**

**User error:** These exceptions are caused by mistakes made by the person using the program. For example, if a user enters incorrect input or performs an action that the program doesn't expect, it can lead to an exception. It's like making a mistake while following instructions or not using a tool correctly.

**Programmer error:** These exceptions occur when the programmer makes mistakes or errors in the code. It could be a bug or oversight in the program logic or a failure to handle certain situations properly. Programmer errors can cause unexpected behavior or exceptions during program execution. It's like a mistake made by the person who wrote the instructions or built something incorrectly.

**Physical resources:** Some exceptions can occur due to limitations or problems with the physical resources that the program interacts with. For example, if a file the program needs to read doesn't exist or if there is a problem with the network connection, it can result in an exception. It's like encountering an obstacle or unavailability of a resource required to complete a task.

A well-designed program includes code that takes into account these potential errors and exceptional conditions. It includes mechanisms to guard against such problems and handle them appropriately. Exception handling is a preferred approach in Java for dealing with these conditions. It's like breaking down the problem into smaller parts and dealing with them separately. Exception handling allows the program to handle errors and exceptional situations gracefully, separating the regular program code from the code specifically designed to handle errors. This approach helps in maintaining code clarity and ensures that the program can continue running and provide appropriate feedback even when errors or exceptional conditions arise.

## EXCEPTION HANDLING

Exception handling is a technique used to catch and handle exceptions that may occur during the execution of a program at runtime. It allows programmers to gracefully handle errors and exceptional situations rather than letting them cause the program to crash or terminate abruptly.

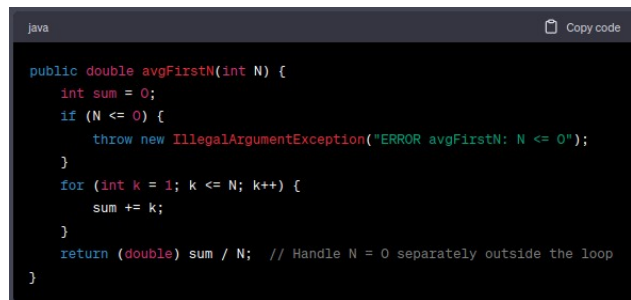
In Java, exceptions are represented by classes that inherit from the `java.lang.Exception` class. When an exceptional situation occurs, such as an error or an unexpected condition, an exception object is created and thrown. The program then looks for an appropriate exception handler to catch and handle the exception.

Java provides two primary ways to handle exceptions:

### 1. Java's Default Exception Handling

**Mechanism:** Java has a default exception handling mechanism that automatically catches and handles certain types of exceptions. If an exception is not explicitly caught and handled by the programmer, the default mechanism takes over. It typically involves displaying an error message and a stack trace (which shows the sequence of method calls that led to the exception) and then terminating the program. This default behavior helps

in identifying and debugging the issues but may not be suitable for all scenarios.



```
java Copy code

public double avgFirstN(int N) {
    int sum = 0;
    if (N <= 0) {
        throw new IllegalArgumentException("ERROR avgFirstN: N <= 0");
    }
    for (int k = 1; k <= N; k++) {
        sum += k;
    }
    return (double) sum / N; // Handle N = 0 separately outside the loop
}
```

### 2. Using the Exception class and Exception Handling Techniques:

In addition to the default exception handling, Java provides a more flexible approach to handle exceptions using the `Exception` class defined in the Java API. This allows programmers to explicitly catch and handle exceptions using try-catch blocks. The try block encloses the code that may potentially throw an exception. If an exception occurs within the try block, it is caught by a corresponding catch block that specifies the type of exception to catch and provides code to handle the exception. Multiple catch blocks can be used to handle different types of exceptions. Additionally, the finally block (if present) is executed regardless of whether an exception occurs or not, allowing for cleanup operations or resource release.

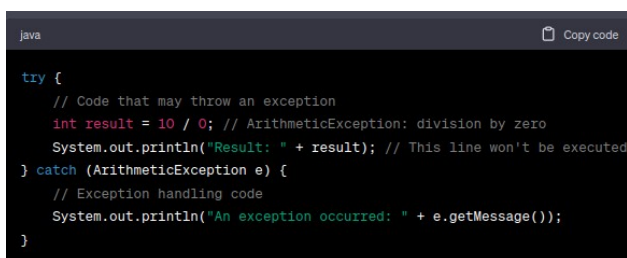
By using the `Exception` class and exception handling techniques, programmers can take more control over how exceptions are handled in their programs. They can catch specific exceptions, provide customized error handling, and take appropriate actions based on the exceptional situations encountered. This helps in making programs more robust, preventing crashes, and improving the overall user experience.

Exception handling is an essential aspect of writing reliable and maintainable code, as it allows developers to anticipate and handle errors effectively, ensuring that programs handle exceptional situations gracefully and continue to execute without unexpected disruptions.

Exception handling in Java is done using five keywords: try, catch, throw, throws, and finally. The try-catch mechanism is commonly used for handling exceptions. Here's a step-by-step explanation of how it works:

1. The code that may potentially throw an exception is enclosed in a block of code starting with the "try" keyword. This block is called the try block.
2. Immediately after the try block, a block of code starting with the "catch" keyword is placed. This block is called the catch block. It specifies the type of exception to catch and provides code to handle the exception.
3. If an exception occurs within the try block, the control flow jumps to the corresponding catch block. The catch block handles the exception by executing the code specified within it. Multiple catch blocks can be used to handle different types of exceptions.
4. If the code within the try block doesn't throw any exception, the catch block is skipped, and the program continues executing the code following the try-catch block.

Here's an example to illustrate the try-catch mechanism:

A screenshot of a code editor with a dark theme. The editor shows a Java code snippet. At the top left, the word 'java' is in the tab. At the top right, there is a 'Copy code' button. The code is as follows:

```
try {  
    // Code that may throw an exception  
    int result = 10 / 0; // ArithmeticException: division by zero  
    System.out.println("Result: " + result); // This line won't be executed  
} catch (ArithmeticException e) {  
    // Exception handling code  
    System.out.println("An exception occurred: " + e.getMessage());  
}
```

In the above example, the try block contains code that performs a division operation. Since

dividing by zero is not allowed, an `ArithmeticException` is thrown. The catch block immediately following the try block catches this exception and executes the code within it, printing an error message.

The try-catch mechanism allows you to handle exceptions and provide appropriate error handling or recovery logic. It helps prevent the program from crashing and allows for controlled handling of exceptional situations. By catching and handling exceptions, you can ensure that your program continues to execute properly even when unexpected errors occur.

### Finally Block

**Finally block:** In addition to the try and catch blocks, you can include an optional finally block. The finally block contains code that will always be executed, regardless of whether an exception occurred or not. It is typically used for performing cleanup operations, such as closing a file or releasing system resources.

1. **Protected code:** The code within the try block is considered protected because it is shielded by the try-catch mechanism. If an exception occurs within the try block, the catch block is responsible for handling it and providing appropriate error handling or recovery logic.
2. **Purpose of finally block:** The finally block ensures that certain operations, specified within it, are always executed, regardless of whether an exception occurred or not. This guarantees that critical tasks or cleanup operations are performed, even in the presence of exceptions or abnormal situations.

In summary, the try-catch-finally mechanism provides a way to handle exceptions and ensure the execution of critical operations. The try block contains the protected code, the catch block handles exceptions, and the finally block executes code that must always be executed. By using these blocks, you can control the flow of your program, handle exceptions gracefully, and perform necessary cleanup tasks.



```

java
Copy code

import java.io.*;

public class FileOperations {
    public static void main(String[] args) {
        BufferedReader reader = null;
        try {
            // Opening a file for reading
            File file = new File("example.txt");
            reader = new BufferedReader(new FileReader(file));

            // Read and print the contents of the file
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (FileNotFoundException e) {
            // Exception handling for file not found
            System.out.println("File not found: " + e.getMessage());
        } catch (IOException e) {
            // Exception handling for IO error
            System.out.println("Error reading the file: " + e.getMessage());
        } finally {
            // Cleanup operations to ensure resources are released
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                System.out.println("Error closing the file: " + e.getMessage());
            }
        }
    }
}

```

In this example, we are opening a file for reading and printing its contents. Let's break down the usage of try-catch-finally:

1. The try block contains the code where we attempt to open the file, read its contents, and print them. The potential exceptions that may occur are handled in separate catch blocks:
  - The catch block for `FileNotFoundException` handles the case when the file does not exist or cannot be opened.
  - The catch block for `IOException` handles any IO errors that may occur during reading the file.
2. The finally block is used for cleanup operations. Here, we ensure that the `BufferedReader` object `reader` is closed properly to release system resources, even if an exception occurs or not. The `close()` method is called within a nested try-catch block to handle any potential IO errors during the closing of the file.

By using the try-catch-finally blocks, we can handle exceptions that may occur while

performing file operations and ensure that necessary cleanup tasks, such as closing the file, are always executed. This helps maintain the integrity of the program and ensures that system resources are released properly.

## CATEGORIES OF EXCEPTION

### Checked Exceptions:

- Checked exceptions are types of exceptions that the Java compiler specifically checks for and requires the programmer to handle or declare. They are considered "checked" because the compiler checks for their presence and enforces handling or declaration.
- These exceptions typically represent conditions that are outside the control of the program but can be anticipated and handled by the programmer.
- Examples of checked exceptions include `IOException`, `SQLException`, `ClassNotFoundException`, and `FileNotFoundException`.
- When a method throws a checked exception, the calling code must either catch and handle the exception using a try-catch block or declare the exception using the `throws` keyword in the method signature.

### Unchecked Exceptions (Runtime Exceptions):

- Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked by the compiler during compilation. The compiler does not force the programmer to handle or declare them explicitly.
- These exceptions are typically caused by programming errors or conditions that are not expected to occur under normal circumstances.
- Examples of unchecked exceptions include `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `IllegalArgumentException`.
- Since unchecked exceptions are not checked by the compiler, it is not

mandatory to catch or declare them. However, it is generally good practice to handle them when possible to prevent unexpected program behavior or crashes.

In summary, checked exceptions are checked by the compiler and must be handled or declared, representing conditions that can be anticipated and handled. Unchecked exceptions, on the other hand, are not checked by the compiler and are typically caused by programming errors or unexpected conditions. While it is not mandatory to handle or declare unchecked exceptions, it is generally recommended to handle them appropriately to ensure the stability and reliability of the program.

## KNOWN EXCEPTIONS

### Checked Exceptions:

- Checked exceptions are errors that the Java compiler specifically checks for and requires the programmer to handle or declare.
- They usually represent external conditions that the program can anticipate and handle.
- Examples:
  - `IOException`: Problems with input/output operations, like reading or writing to files.
  - `SQLException`: Errors related to accessing or manipulating databases.
  - `ClassNotFoundException`: When a specified class is not found during runtime.
  - `InterruptedException`: Occurs when a thread is interrupted while it is sleeping or waiting.
  - `FileNotFoundException`: Trying to access a file that does not exist or cannot be opened.
  - `ParseException`: Errors when converting a string into another data type, like a date or number.

- `MalformedURLException`: Problems with malformed URLs.

### Unchecked Exceptions (Runtime Exceptions):

- Unchecked exceptions are errors that the compiler does not specifically check for, and the programmer is not required to handle or declare them.
- They are often caused by programming mistakes or unexpected conditions.
- Examples:
  - `ArithmeticException`: Errors during arithmetic operations, like dividing by zero.
  - `NullPointerException`: When trying to use or access a null reference.
  - `ArrayIndexOutOfBoundsException`: Accessing an invalid index in an array.
  - `IllegalArgumentException`: Passing an invalid argument to a method.
  - `ClassCastException`: Attempting to cast an object to a type it is not compatible with.
  - `IllegalStateException`: When the state of an object is not suitable for a particular operation.
  - `NumberFormatException`: Errors when converting an invalid string into a numeric type.

In simpler terms, checked exceptions are checked by the compiler and require handling or declaration, while unchecked exceptions are not specifically checked by the compiler and often indicate programming errors or unexpected situations. It's important to handle exceptions appropriately to ensure program stability and reliability.

## GETTING INFORMATION FROM EXCEPTION

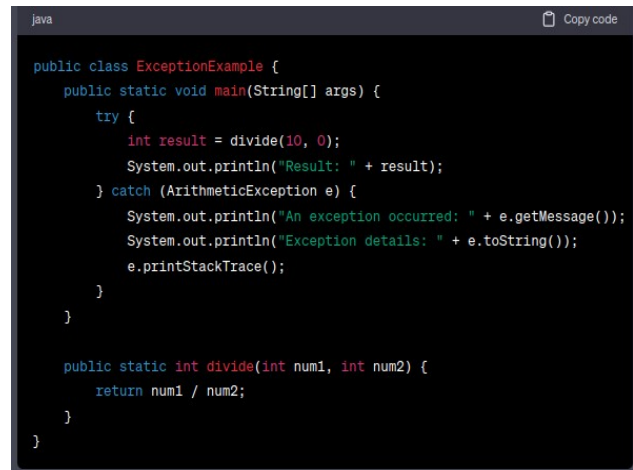
When an exception occurs in a Java program, it provides valuable information about the nature of the error or exceptional situation that has occurred. This information is encapsulated within an exception object. By accessing and examining this object, developers can gain insights into the

cause of the exception, which aids in understanding and resolving the problem.

Java provides several methods in the Throwable class (from which all exceptions are derived) that allow us to extract information from an exception object. These methods include:

1. `getMessage()`: This method returns a String containing a descriptive message associated with the exception. It provides additional details about the exception and what went wrong.
2. `toString()`: This method returns a String representation of the exception object. It combines the class name, the message obtained from `getMessage()`, and additional information about the exception. It provides a summarized view of the exception.
3. `printStackTrace()`: This method outputs the exception and its call stack trace to the console. The stack trace shows the sequence of method calls that led to the exception, helping identify the exact location where the exception occurred. It is particularly useful for debugging purposes.

By utilizing these methods, developers can extract relevant information from exceptions to understand the nature and cause of the error. This information can aid in logging the error, displaying meaningful error messages to users, or assisting in debugging and troubleshooting issues in the program.

A screenshot of a Java code editor with a dark theme. The code defines a class `ExceptionExample` with a `main` method. Inside `main`, a `try` block calls `divide(10, 0)`. A `catch` block for `ArithmeticException` prints the exception's message, details, and stack trace. A separate `divide` method is also shown, which simply returns the result of integer division.

```
java Copy code

public class ExceptionExample {
    public static void main(String[] args) {
        try {
            int result = divide(10, 0);
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("An exception occurred: " + e.getMessage());
            System.out.println("Exception details: " + e.toString());
            e.printStackTrace();
        }
    }

    public static int divide(int num1, int num2) {
        return num1 / num2;
    }
}
```

In this example, we have a method `divide()` that performs integer division. However, we divide by zero intentionally to generate an `ArithmeticException`. Here's how the code retrieves information from the exception:

1. Inside the `try` block, the `divide()` method is called with arguments 10 and 0, which triggers an `ArithmeticException`.
2. In the `catch` block, we catch the `ArithmeticException` and retrieve information from the exception object `e`.
  - `e.getMessage()` retrieves the exception's message, which is typically a brief description of the error. It helps to understand what went wrong in the division operation.
  - `e.toString()` returns a string representation of the exception. It includes the class name, the exception message, and additional details about the exception.
  - `e.printStackTrace()` prints the exception and its stack trace to the console. It shows the method calls leading up to the exception, helping identify the exact location where the exception occurred.

By utilizing these methods (`getMessage()`, `toString()`, and `printStackTrace()`), we can gather information about the exception, such as the

error message, the class name, and the stack trace. This information is valuable for understanding the cause of the exception and assisting in debugging and troubleshooting the program.