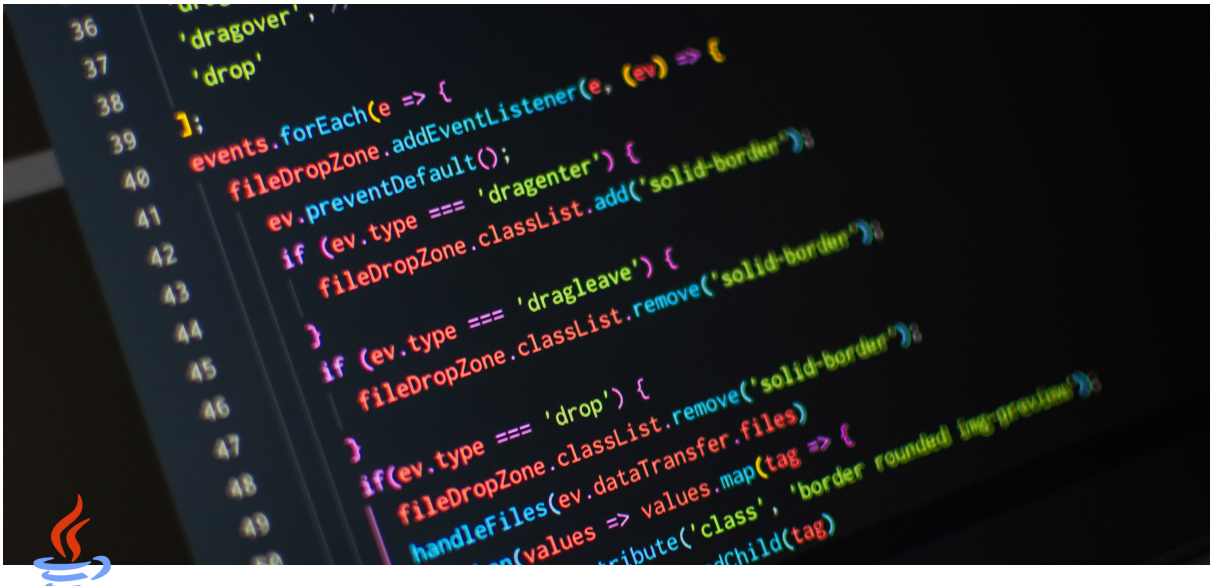

Java OOP

Through Maya Mnaizel

Object Oriented Programming in Java is a robust paradigm that revolves around the principles of objects, encapsulation, inheritance, polymorphism, and abstraction.

Classes and Objects

- Class: a blueprint or template for creating objects. It defines properties (attributes or fields) and behaviors (methods) that objects of the class have
- Object: an instance of a class. It represents a real-world entity and encapsulates data and behavior

```
public class Car{
  //Attributes
  String model;
  int year;

  //Constructor
  public Car(String model, int year){
    this.model = model;
    this.year = year;
  }
```

```
  //Method
  public void displayInfo(){
    System.out.println("Model: " + model)
    System.out.println("Year: " + year)
}

Public static void main(String[]args){
  //creating an object of the Car class
    Car myCar = new Car ("Toyota", 2022);

  //Using the object to call a method
  myCar.displayInfo();
  }
}
```

Encapsulation

The bundling of data and methods that operate on that data within a single unit. It hides the internal state of an object and restricts access to certain components, promoting information hiding and reducing complexity

```
public class BankAccount{
  //Private attributes
  private double balance;

  //Public method to access balance
  public double getBalance(){
    return balance;
  }

  //Public method to deposit money
  public void deposit(double amount){
    if(amount > 0){
      balance += amount;
      System.out.println("Deposit successful, Balance: " + balance);
    } else {
          System.out.println("Invalid deposit amount");
    }
  }
  public static void main(String[]args){
    BankAccont account = new BankAccount();
    account.deposit(1000);
    System.out.println("Current balance: " + account.getBalance());
```

Inheritance

Allows a class (subclass or derived class) to inherit the properties and behaviors of another class (superclass or base class). It promotes code reuse, extending functionality, and creating a hierarchy of classes

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

//Dog class inherits from Animal
public class Dog extends Animal{
  public void bark(){
    System.out.println("Dog is barking");
  }
  public static void main(String[]args){
    Dog myDog = new Dog();
    myDog.eat(); //inherited method
    myDog.bark(); //Dog class method
  }
}
```

Polymorphism

Allows objects of different types to be treated as objects of a common type. There are two types:

- Compile-time (Static) Polymorphism
  - This is achieved through method overloading, where multiple methods have the same name but different parameters
- Runtime (Dynamic) Polymorphism
  - This is achieved through method overriding, where a subclass provides a specific implementation for a method already defined in its superclass

```
Public class Shape{
  public void draw(){
        System.out.println("Drawing a shape");
  }
}

//Circle class overrides the draw method
public class Circle extends Shape{
  @Override
  public void draw(){
      System.out.println("Drawing a circle")'
  }
  public static void main(String[]args){
    Shape myShape = new Circle();
    myShape.draw();//calls out circle method
  }
}
```

Abstraction

Involves simplifying complex systems by modeling classes based on the essential characteristics they share, while ignoring unnecessary details. Abstract classes and interfaces are used to define abstract types

```
//Abstract class
abstract class Shape{
  //Abstract method with no implementation
  public abstract void draw();

  //Concrete method
  public void displayInfo(){
    System.out.println("This is a shape");
  }
}

//Implementing the abstract class
class Circle extends Shape{
  @Override
  public void draw(){
    System.out.println("Drawing a circle");
  }
}
```

```
public class AbstractionExample {
    public static void main(String[] args) {
        Circle myCircle = new Circle();
        myCircle.draw();
        myCircle.displayInfo();
    }
}
```

Interfaces

A collection of abstract methods. Classes implement interfaces, ensuring that they provide concrete implementations for all methods defined in the interface. Interfaces allow for multiple inheritance enabling a class to implement multiple interfaces

```
//Interface Method
interface Playable{
  void play();
}

//Class implementing the interface
class Guitaer implements Playable{
  @Override
  public void play(){
    System.out.println("Playing the guitar");
  }
}
public class InterfaceExample {
    public static void main(String[] args) {
        Guitar myGuitar = new Guitar();
        myGuitar.play();
    }
}
```

Static Member

The ‘*static*’ keyword in Java is used to create class-level members that are shared among all instances of the class.

Static members are accessed using the class name rather than an instance

```
public class MathUtils {
    // Static variable
    public static final double PI = 3.14159;

    // Static method
    public static int add(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println("PI: " + MathUtils.PI);
        System.out.println("Sum: " + MathUtils.add(5, 7));
    }
}
```

Constructors

Special methods with the same name as the class, are used for initializing objects when they are created. They can take parameters and set the initial state of the object

```
public class Car {
    // Instance variables
    String model;
    int year;

    // Parameterized constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    // Method to display car information
    public void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }

    public static void main(String[] args) {
        // Creating an object of the Car class using the constructor
```

```
        Car myCar = new Car("Toyota Camry", 2022);

        // Using the object to call a method
        myCar.displayInfo();
    }
}
```

Composition

Involves creating objects of other classes within a class. It allows for the creation of complex objects by combining simpler ones, promoting code reuse and flexibility

```
//Engine class used in composition
class Engine{
  public void start(){
    System.out.println("Engine started");
  }
}

//Car class using composition
class Car
  private Engine engine;

  //Constructor with composition
  public Car(Engine engine){
    this.engine = engine;
  }

  public void startCar(){
    engine.start();
    System.out.println("Car started");
  }
}
public class CompositionExample {
    public static void main(String[] args) {
        // Creating an Engine
        Engine myEngine = new Engine();

        // Creating a Car with the Engine
        Car myCar = new Car(myEngine);

        // Starting the Car
        myCar.startCar();
    }
}
```