Encapsulation, Inheritance and Polymorphism

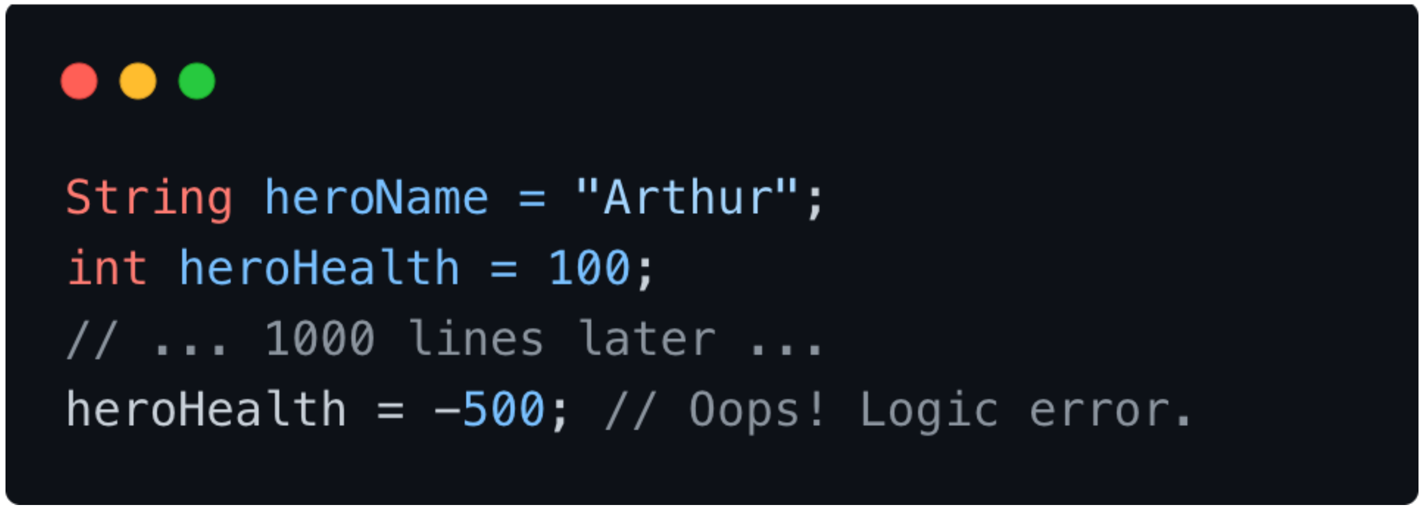
ENG23 2032

Object–Oriented Technology

Dr. Nuntawut Kaoungku
Assistant Professor of Computer Engineering

Why do we need OOP?

- **Scenario**: You are building a Game Character.
- **The "Old" Way**:



```
String heroName = "Arthur";
int heroHealth = 100;
// ... 1000 lines later ...
heroHealth = -500; // Oops! Logic error.
```

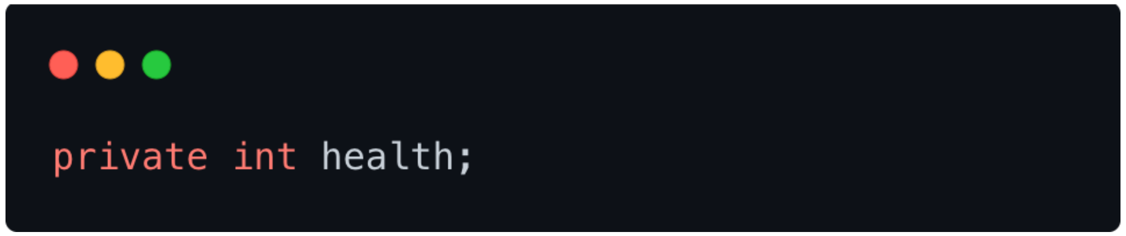
- **The Problem:** Data is global or unprotected. Anyone can break the state of your application.
- **The Solution:** Bundle the data and the logic together -> Objects.

Encapsulation (The Shield)

- **Definition:** Bundling data (variables) and methods (functions) into a single unit (Class) and restricting direct access.

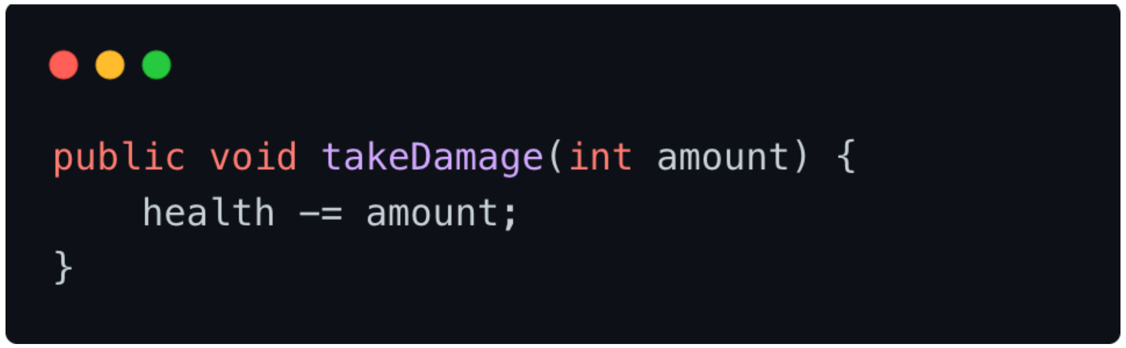
- **The Golden Rule:**

  - Variables should usually be Private.



```
private int health;
```

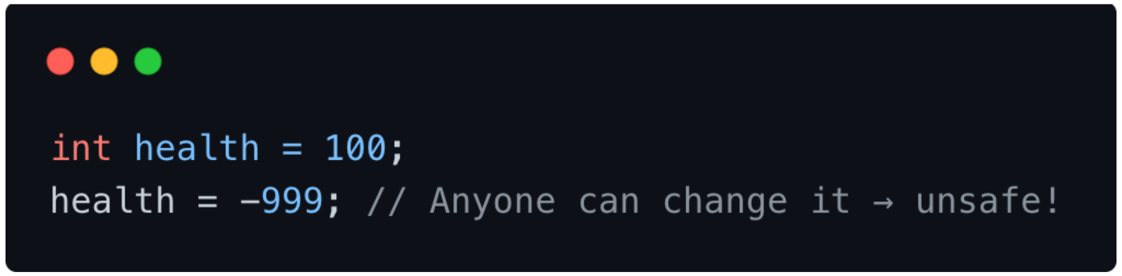
  - Methods can be Public (to provide controlled access).



```
public void takeDamage(int amount) {
    health -= amount;
}
```

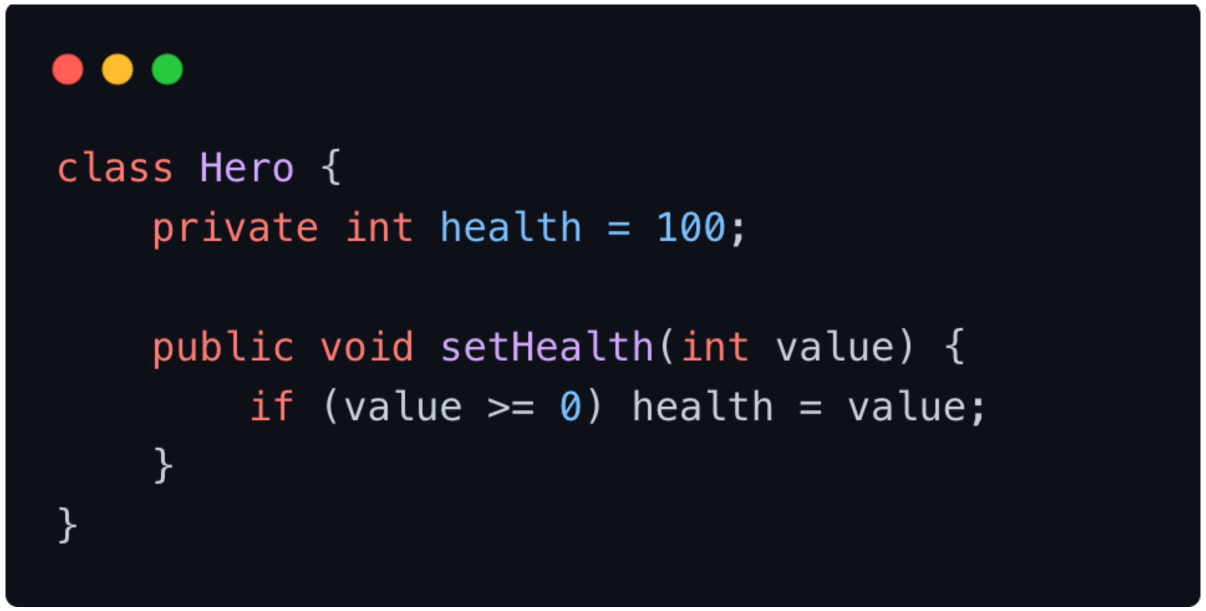
Example - Encapsulation

❌ Without Encapsulation



```java
int health = 100;
health = -999; // Anyone can change it → unsafe!
```

✓ With Encapsulation



```java
class Hero {
    private int health = 100;

    public void setHealth(int value) {
        if (value >= 0) health = value;
    }
}
```

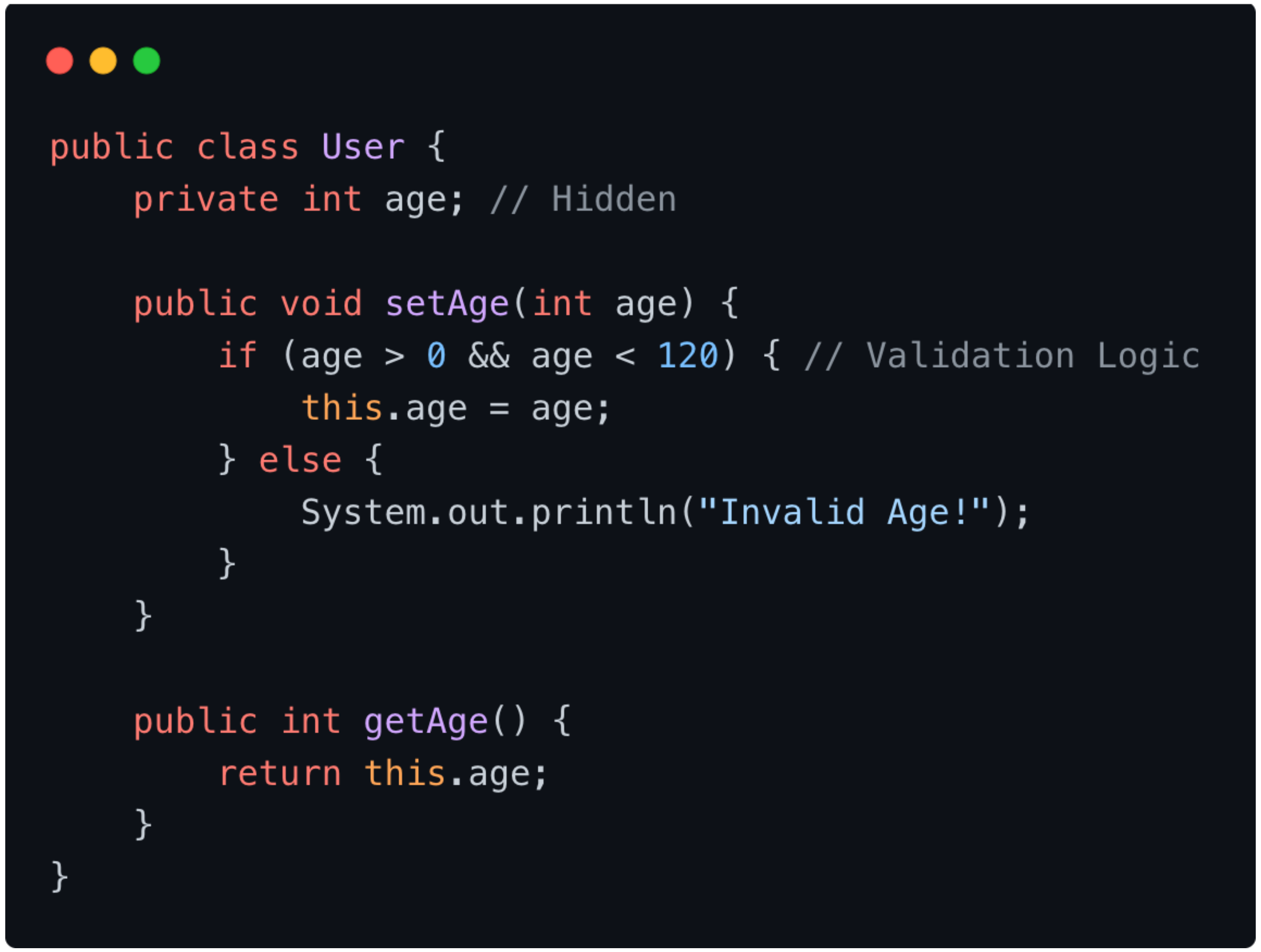
Controlling Visibility

Modifier	Class	Package	Subclass	World	Role
`public`	✅	✅	✅	✅	Open to everyone
`protected`	✅	✅	✅	❌	For family (inheritance)
`default`	✅	✅	❌	❌	Package neighbors only
`private`	✅	❌	❌	❌	**Strictly internal**

Getters and Setters

- Instead of accessing *user.age* directly, we use methods.

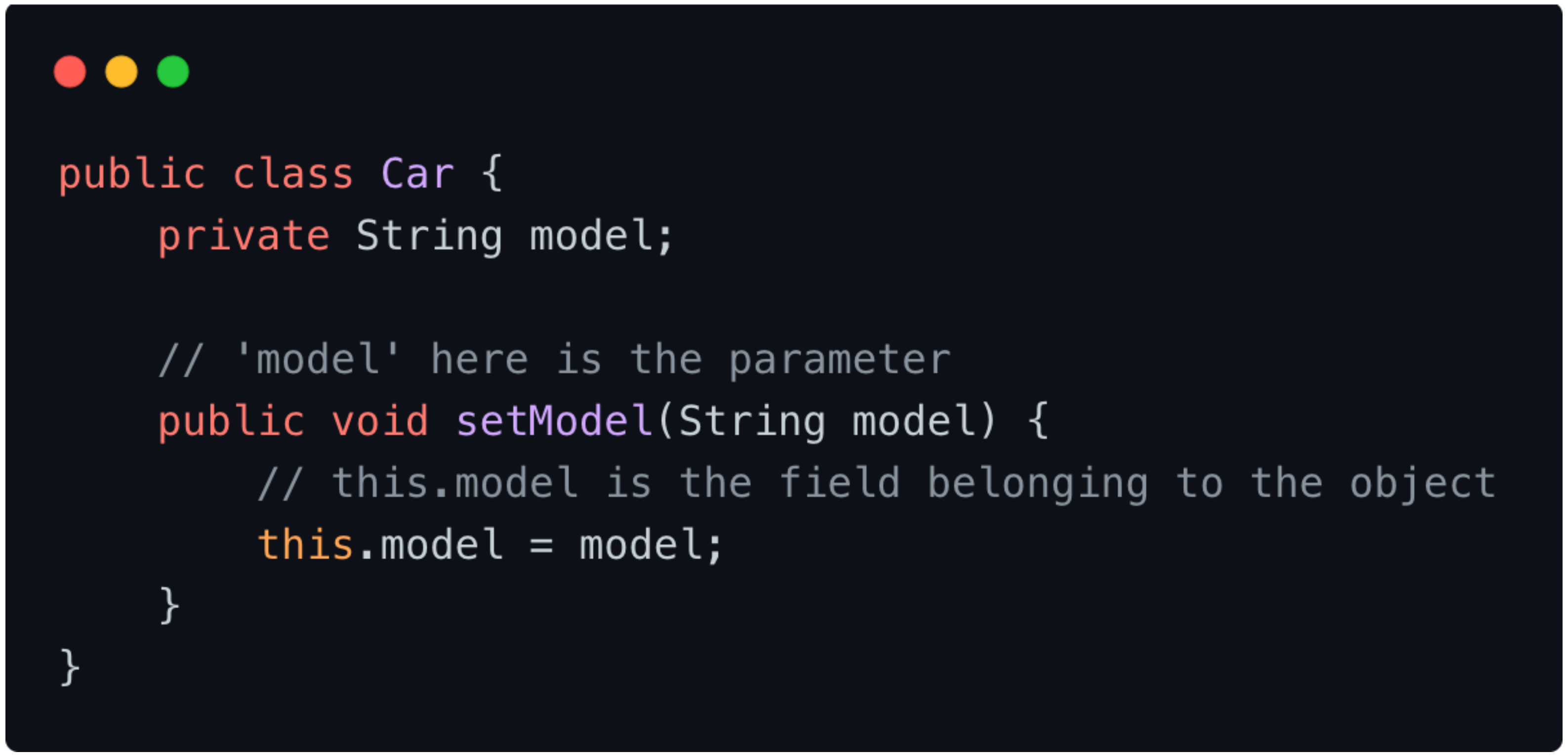
- Why? Validation!



```
public class User {
    private int age; // Hidden

    public void setAge(int age) {
        if (age > 0 && age < 120) { // Validation Logic
            this.age = age;
        } else {
            System.out.println("Invalid Age!");
        }
    }

    public int getAge() {
        return this.age;
    }
}
```

The *this* Keyword (Who am I?)

- **Problem:** Naming conflicts between Parameters and Fields.
- **Solution:** this refers to the current object instance.
- *Code Example*:

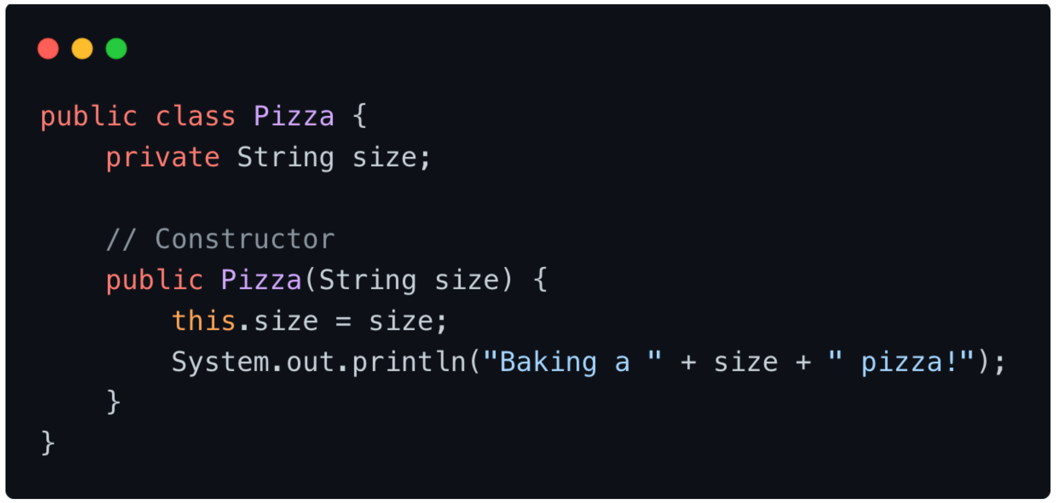


```java
public class Car {
    private String model;

    // 'model' here is the parameter
    public void setModel(String model) {
        // this.model is the field belonging to the object
        this.model = model;
    }
}
```

Constructors

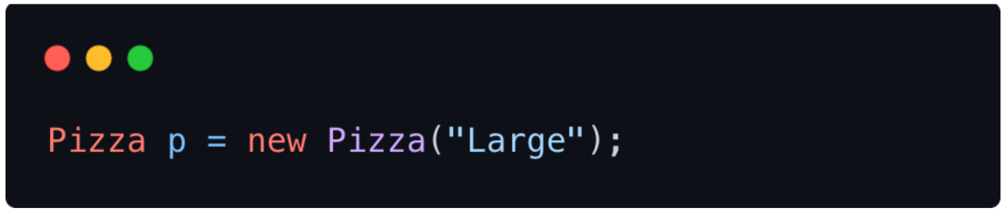
- A special method called when an object is instantiated (new).
- Rules:
  - Must have the same name as the Class.
  - Must not have a return type (not even void).

Code Example:



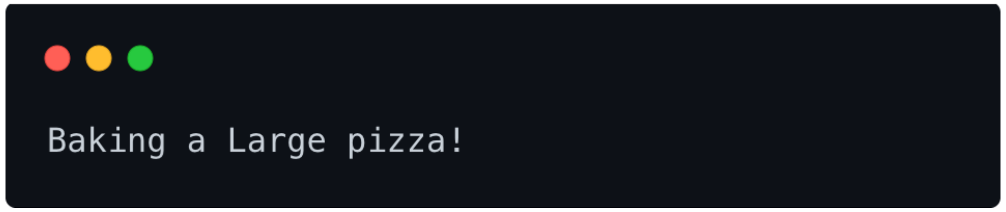
```
public class Pizza {
    private String size;

    // Constructor
    public Pizza(String size) {
        this.size = size;
        System.out.println("Baking a " + size + " pizza!");
    }
}
```

Usage:



```
Pizza p = new Pizza("Large");
```

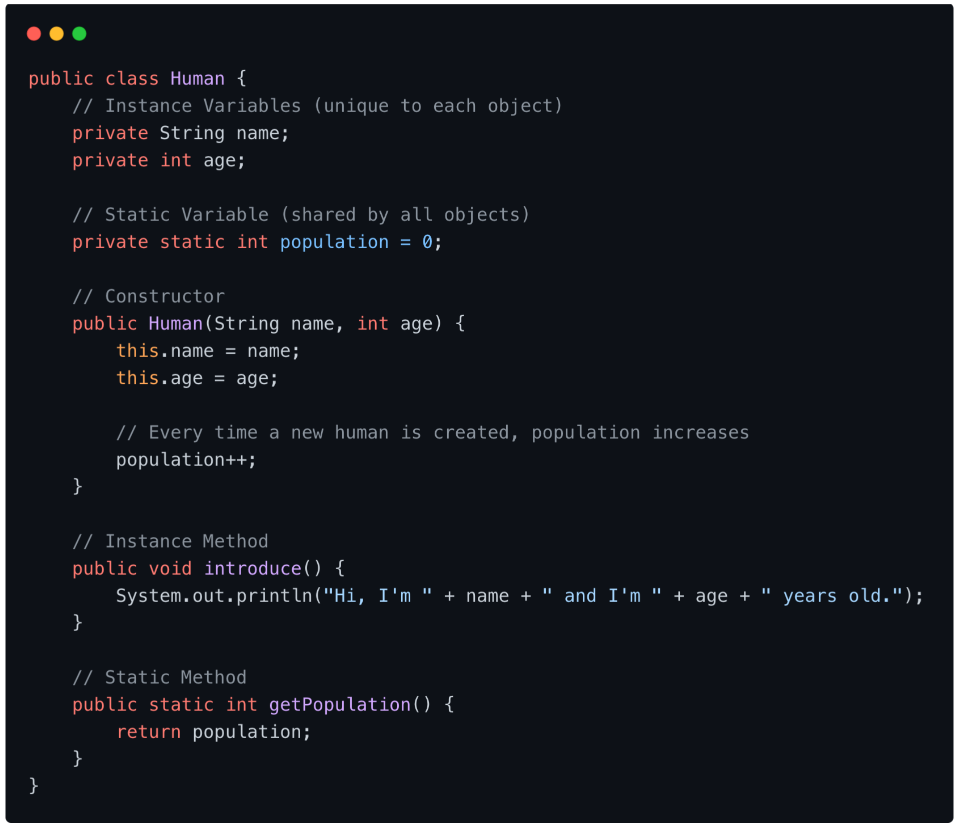
Result:



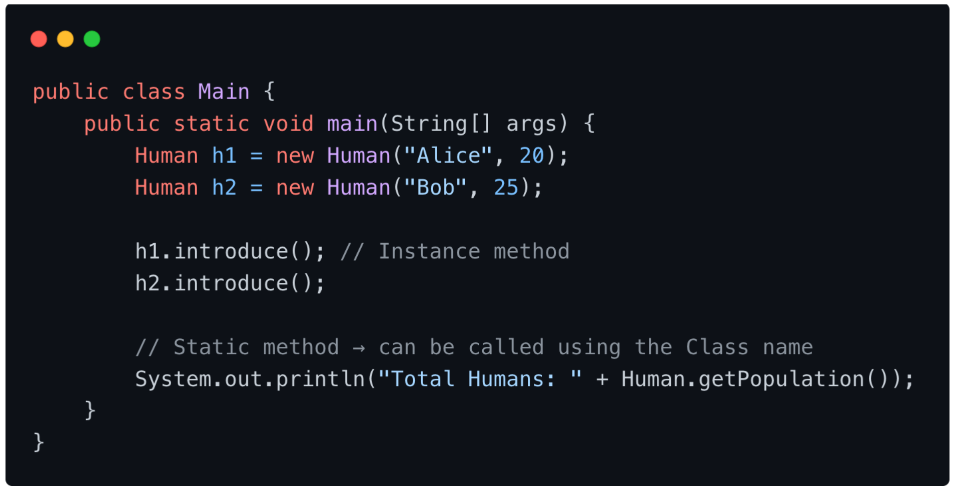
```
Baking a Large pizza!
```

Static vs Instance

- **Instance Variable (Non-static)**:
    - Belongs to a specific object.
    - Example: *myEyeColor, myHeight*.
- **Static Variable**:
    - Belongs to the Class. Shared by all objects.
    - Example: *HumanPopulation, Math.PI.*



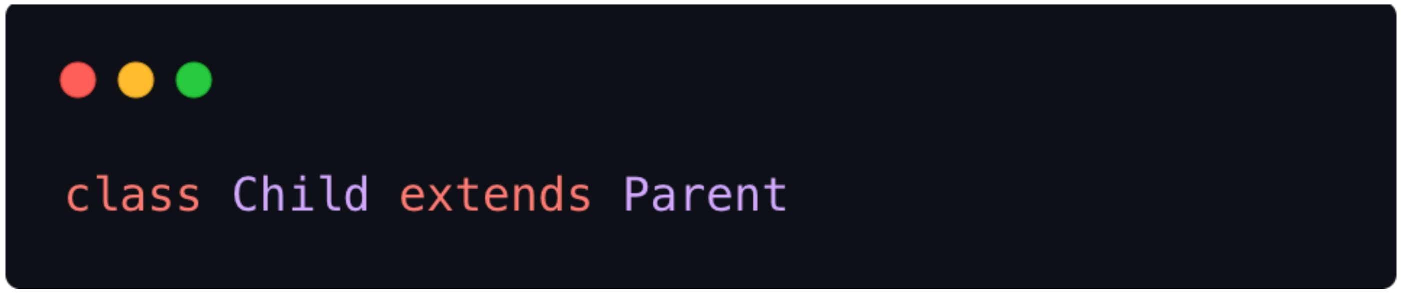
```java
public class Human {
    // Instance Variables (unique to each object)
    private String name;
    private int age;

    // Static Variable (shared by all objects)
    private static int population = 0;

    // Constructor
    public Human(String name, int age) {
        this.name = name;
        this.age = age;

        // Every time a new human is created, population increases
        population++;
    }

    // Instance Method
    public void introduce() {
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years old.");
    }

    // Static Method
    public static int getPopulation() {
        return population;
    }
}
```



```java
public class Main {
    public static void main(String[] args) {
        Human h1 = new Human("Alice", 20);
        Human h2 = new Human("Bob", 25);

        h1.introduce(); // Instance method
        h2.introduce();

        // Static method → can be called using the Class name
        System.out.println("Total Humans: " + Human.getPopulation());
    }
}
```

Inheritance (The Family Tree)

- **Definition:** A mechanism where a new class acquires the properties and behaviors of an existing class.
- **Terminology**:
  - Parent (Superclass / Base Class)
  - Child (Subclass / Derived Class)
- **Syntax**:

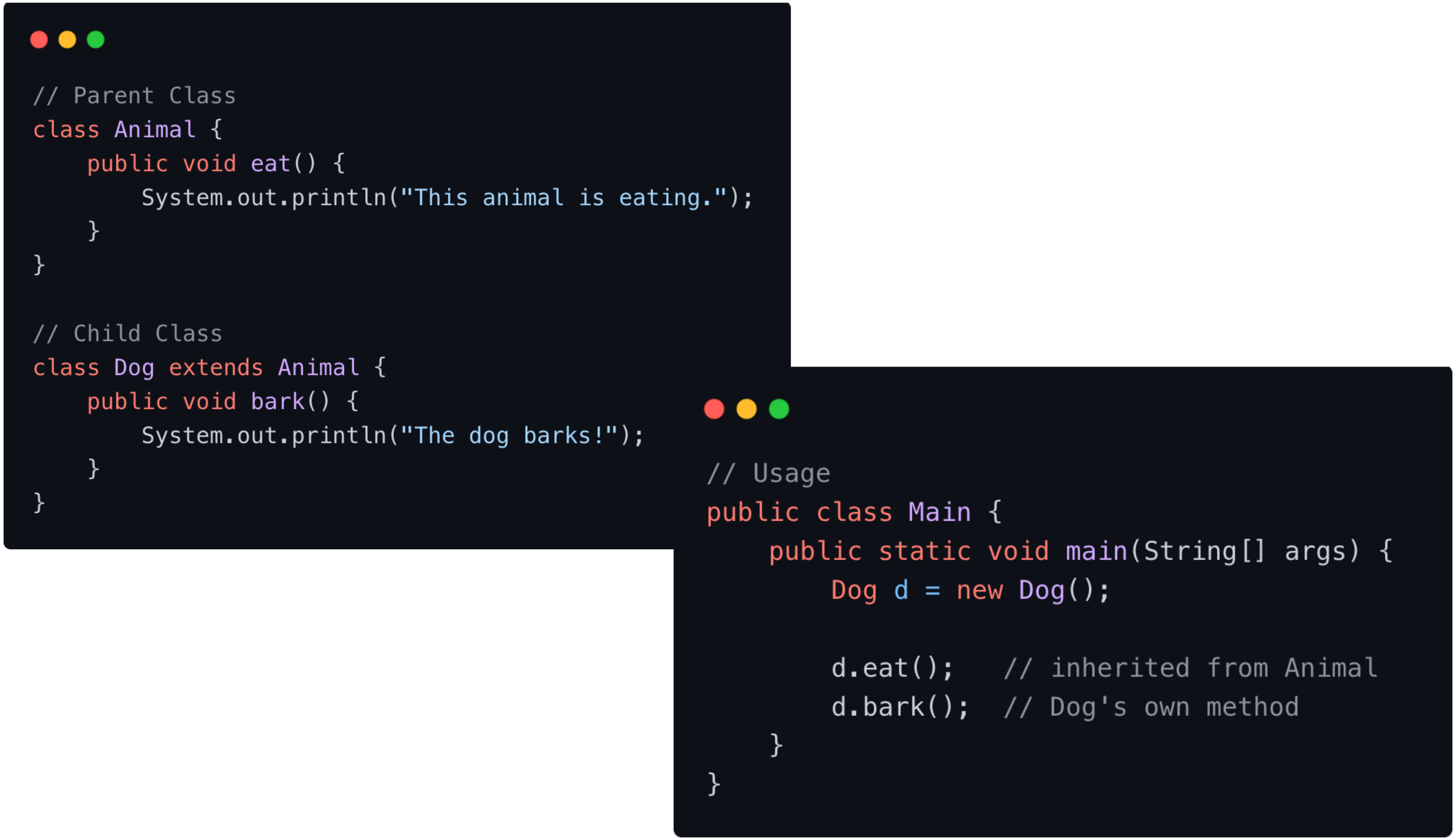


```
class Child extends Parent
```

When to use Inheritance?

- Use Inheritance only if the relationship is "Is-A".
  - A Cat is an Animal? -> ✅ Yes.
  - A Manager is an Employee? -> ✅ Yes.
  - A Car is a Wheel? -> ❌ No. (This is "Has-A" / Composition).

Basic Example : Inheritance in Action



```java
// Parent Class
class Animal {
    public void eat() {
        System.out.println("This animal is eating.");
    }
}

// Child Class
class Dog extends Animal {
    public void bark() {
        System.out.println("The dog barks!");
    }
}
```

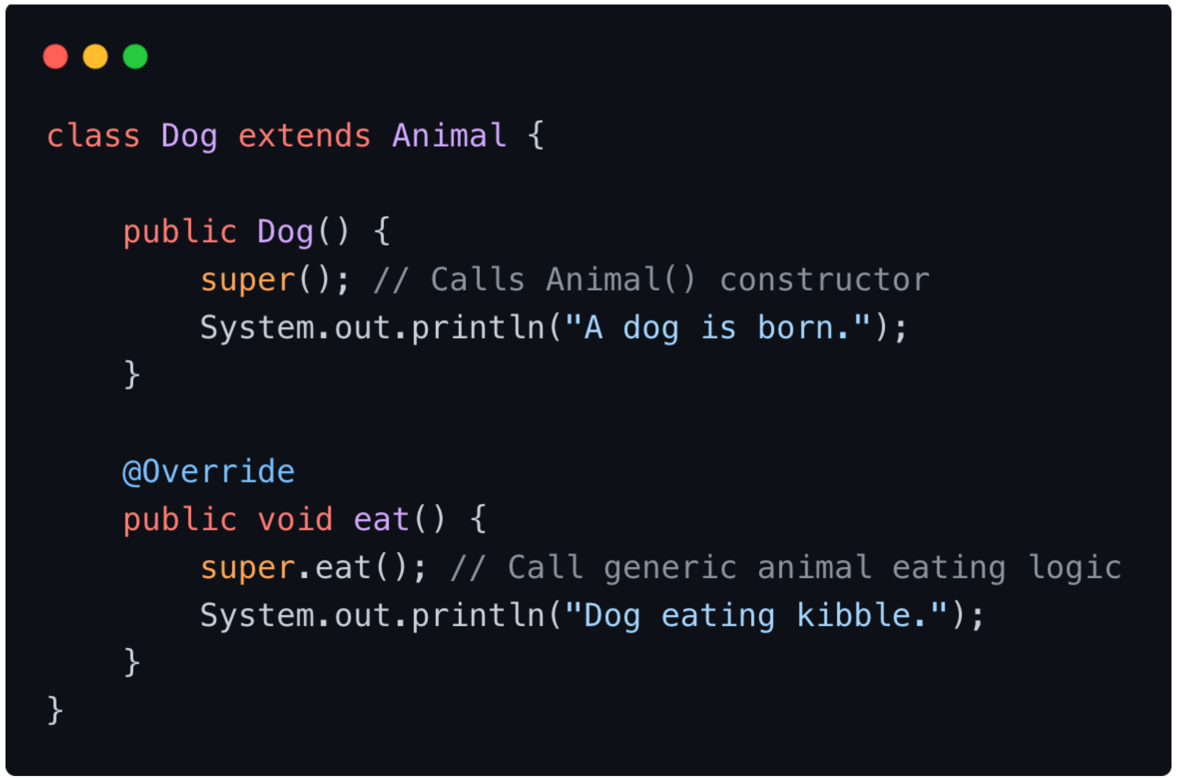
```java
// Usage
public class Main {
    public static void main(String[] args) {
        Dog d = new Dog();

        d.eat();   // inherited from Animal
        d.bark();  // Dog's own method
    }
}
```

The *super* Keyword (Talking to Parents)

- *super* is used to access the Parent class members.

- **Usage 1**: Calling Parent Constructor (*Must be the first line!*).

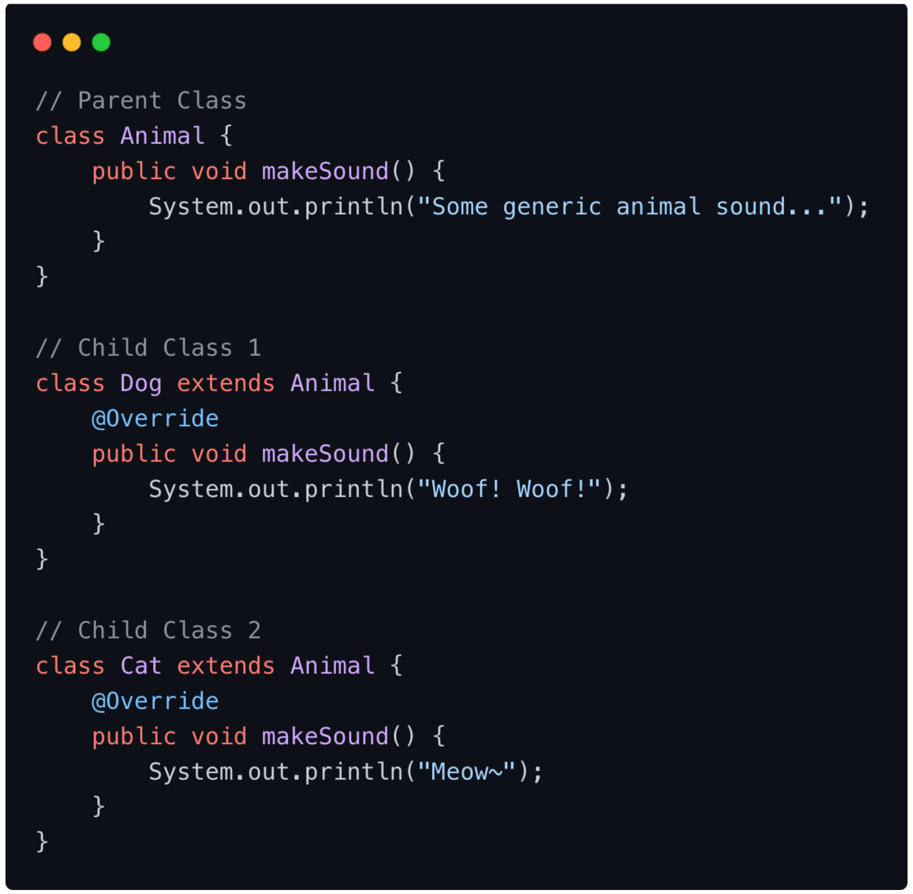
- **Usage 2**: Calling Parent Methods.



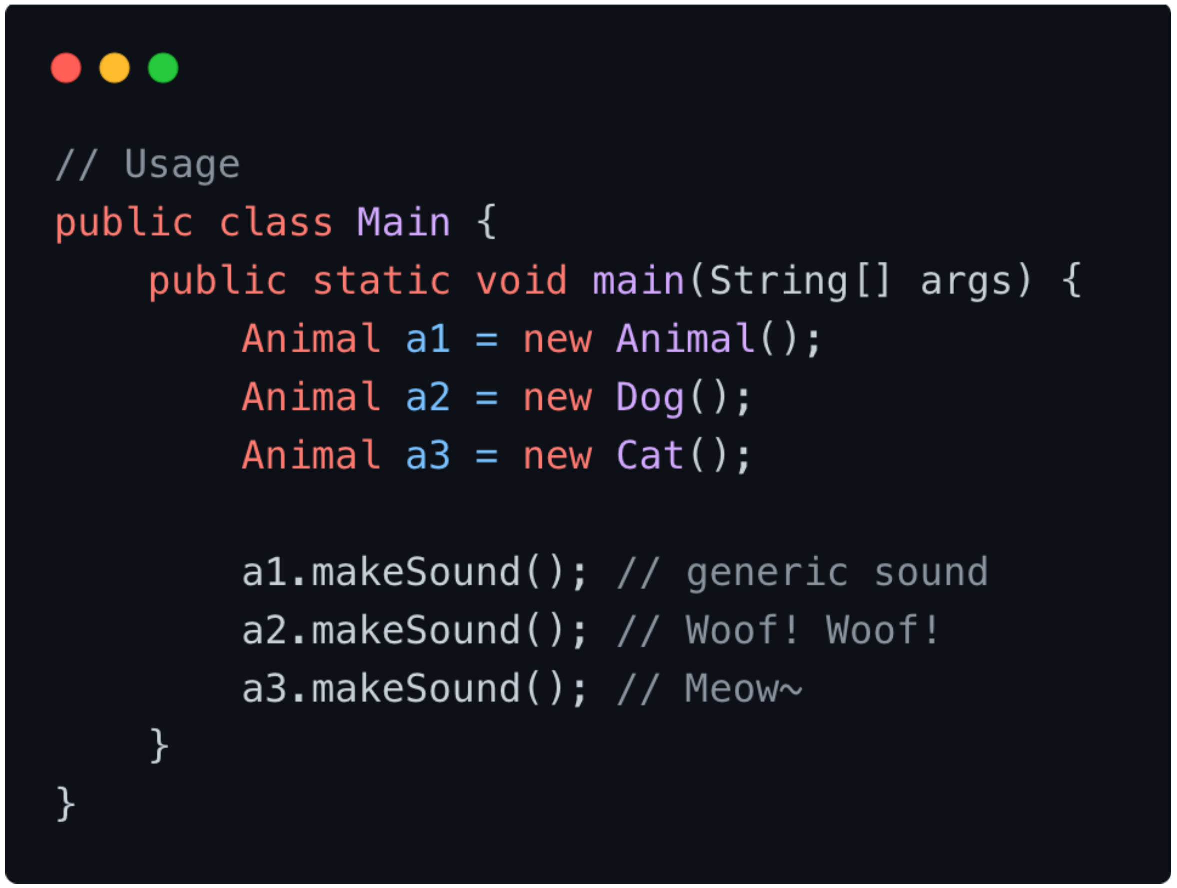
```java
class Dog extends Animal {

    public Dog() {
        super(); // Calls Animal() constructor
        System.out.println("A dog is born.");
    }

    @Override
    public void eat() {
        super.eat(); // Call generic animal eating logic
        System.out.println("Dog eating kibble.");
    }
}
```

Method Overriding (Changing Behavior)

- Definition: When a Child class provides a specific implementation of a method already defined in the Parent.
- Annotation: Always use @Override to prevent typos.



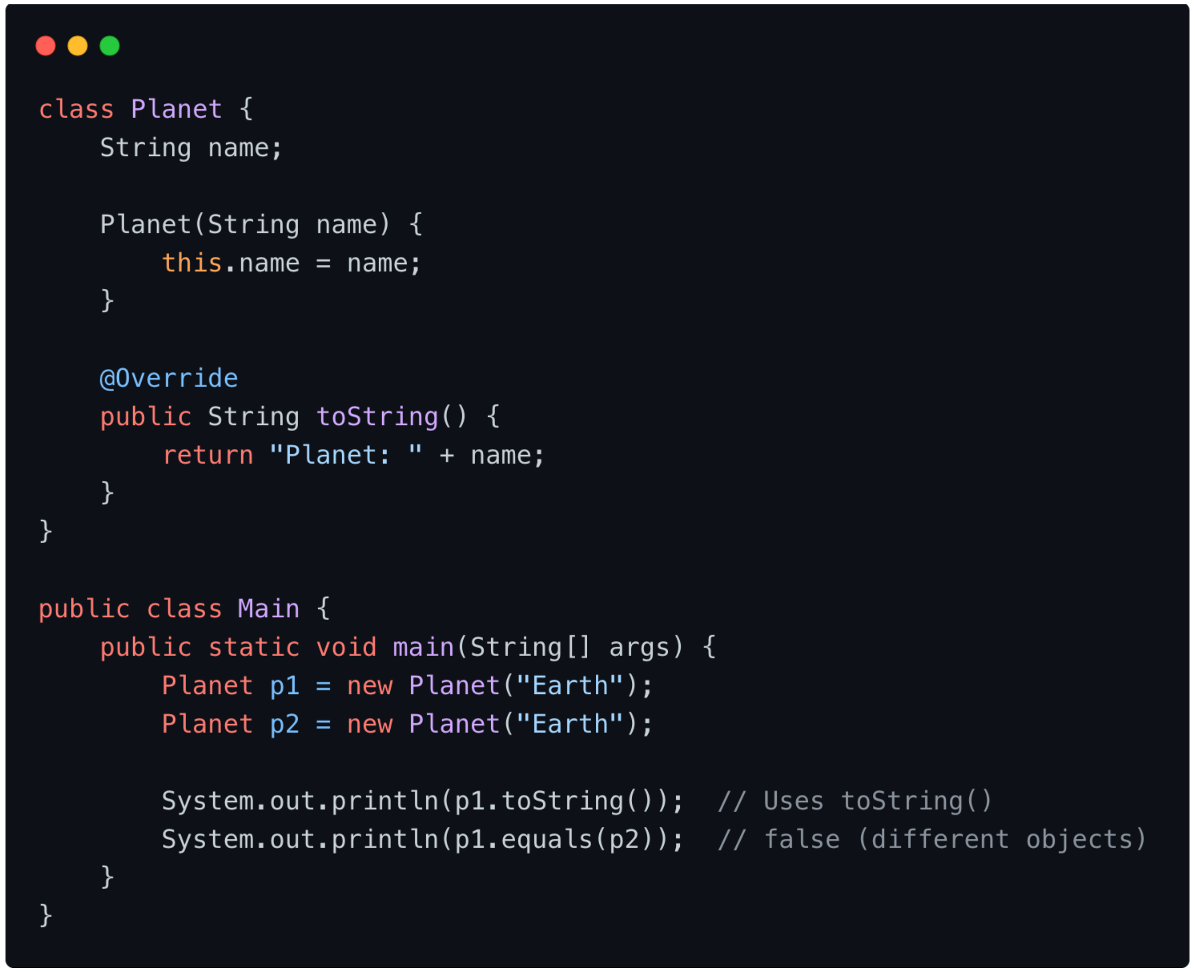
```java
// Parent Class
class Animal {
    public void makeSound() {
        System.out.println("Some generic animal sound...");
    }
}

// Child Class 1
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Woof! Woof!");
    }
}

// Child Class 2
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow~");
    }
}
```



```java
// Usage
public class Main {
    public static void main(String[] args) {
        Animal a1 = new Animal();
        Animal a2 = new Dog();
        Animal a3 = new Cat();

        a1.makeSound(); // generic sound
        a2.makeSound(); // Woof! Woof!
        a3.makeSound(); // Meow~
    }
}
```

The Cosmic Superclass

- Every class in Java implies extends Object.
- Common methods you inherit automatically:
    - *toString()*: Converts object to String (usually memory address unless overridden).
    - equals(Object o): Compares if two objects are the same instance.



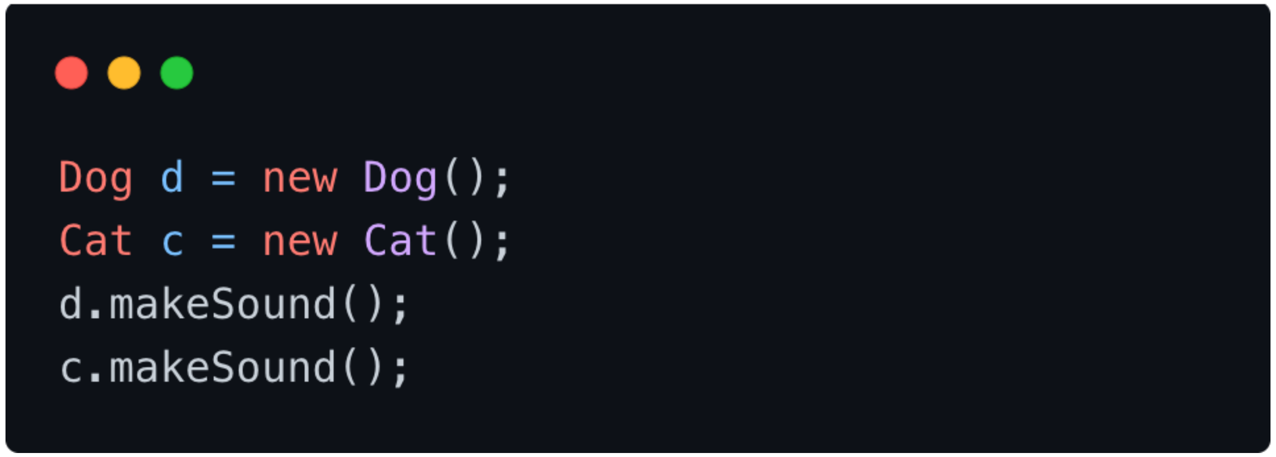
```
class Planet {
    String name;

    Planet(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Planet: " + name;
    }
}

public class Main {
    public static void main(String[] args) {
        Planet p1 = new Planet("Earth");
        Planet p2 = new Planet("Earth");

        System.out.println(p1.toString());  // Uses toString()
        System.out.println(p1.equals(p2));  // false (different objects)
    }
}
```

Polymorphism (Many Forms)

- Poly = Many, Morph = Forms.
- Concept: One interface, many implementations.
- It allows us to treat different specific objects (Cat, Dog, Cow) as a single general type (Animal).

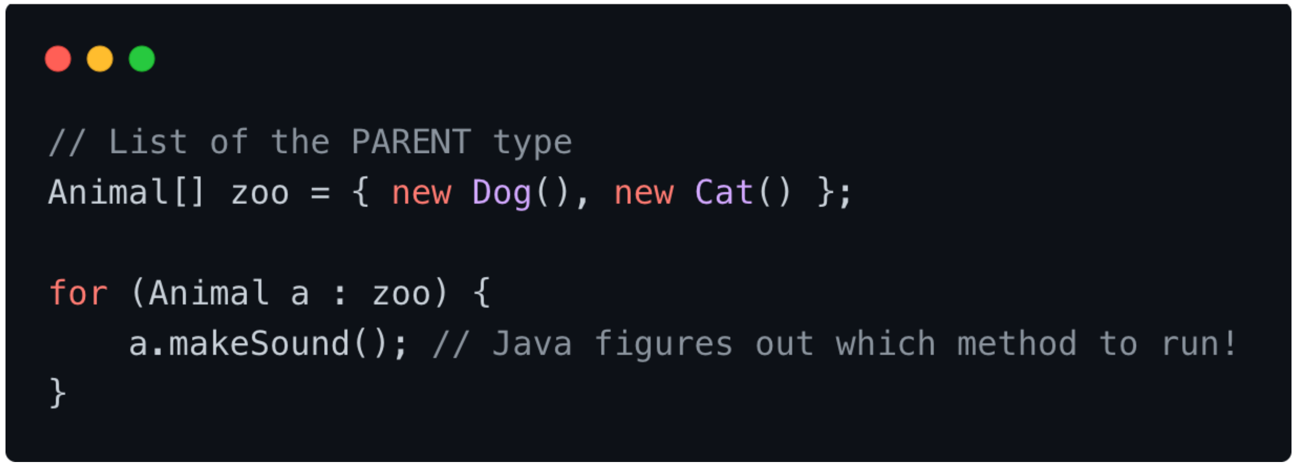
Polymorphism in Action

Without Polymorphism



```
Dog d = new Dog();
Cat c = new Cat();
d.makeSound();
c.makeSound();
```

With Polymorphism:



```
// List of the PARENT type
Animal[] zoo = { new Dog(), new Cat() };

for (Animal a : zoo) {
    a.makeSound(); // Java figures out which method to run!
}
```

Abstract Classes

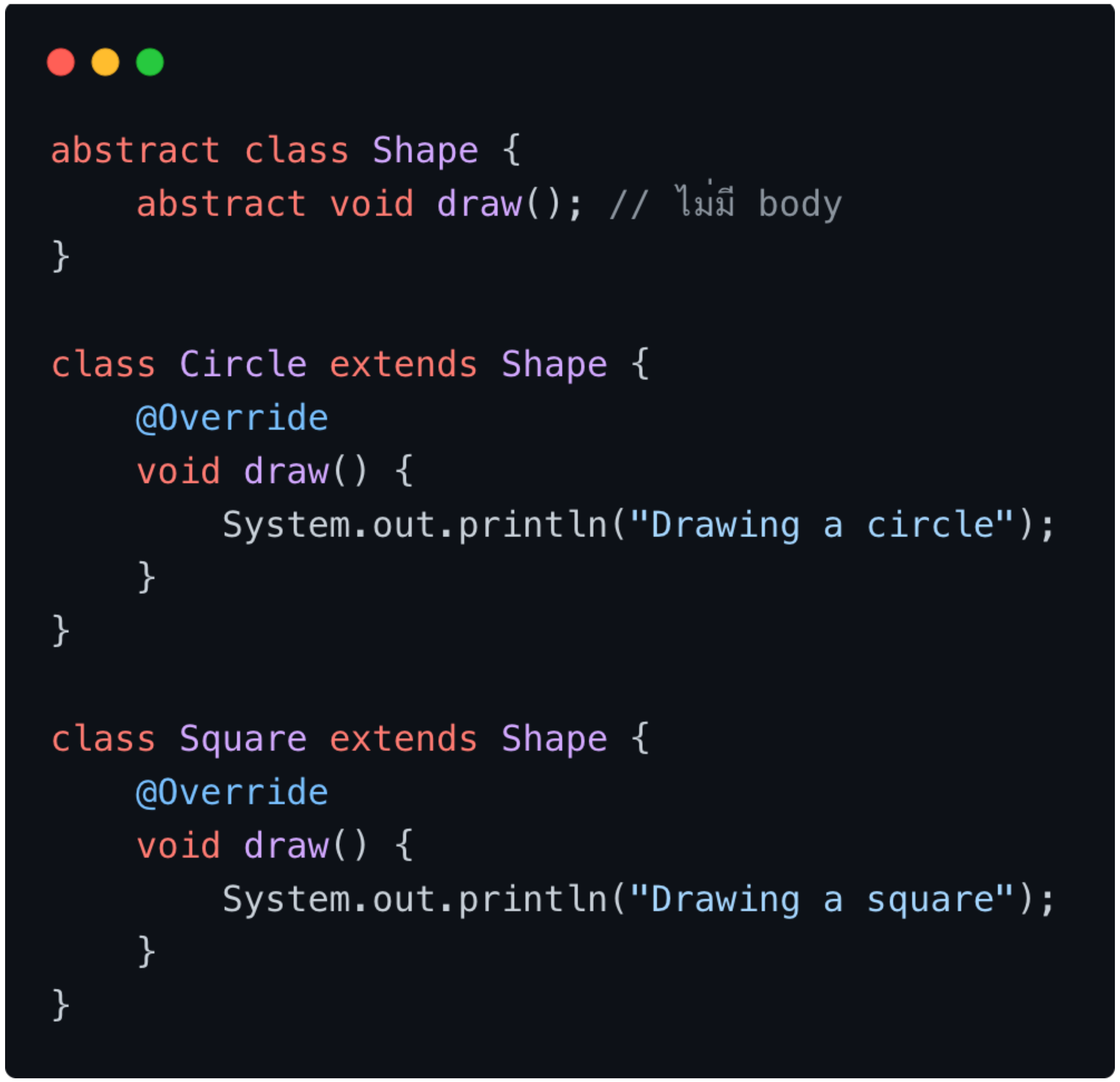
- Sometimes, a parent class is just a concept.

- **Example**: You cannot have just a "Shape". It must be a Circle or a Square.

- **Keyword**: abstract

  - Cannot be instantiated (new Shape() is illegal).

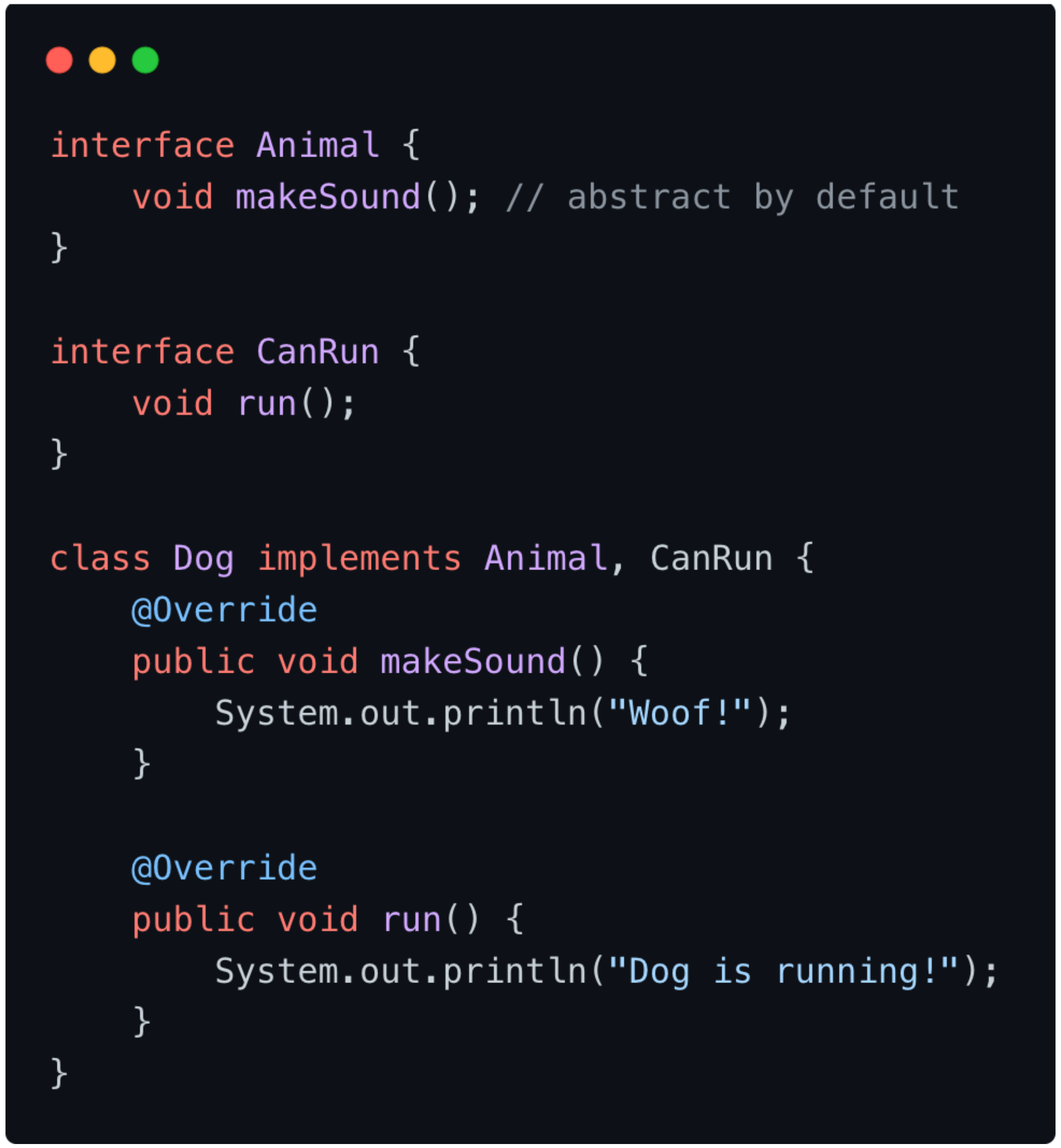
  - Can have abstract methods (methods without body).



```
abstract class Shape {
    abstract void draw(); // ไม่มี body
}

class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle");
    }
}

class Square extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a square");
    }
}
```

Interfaces

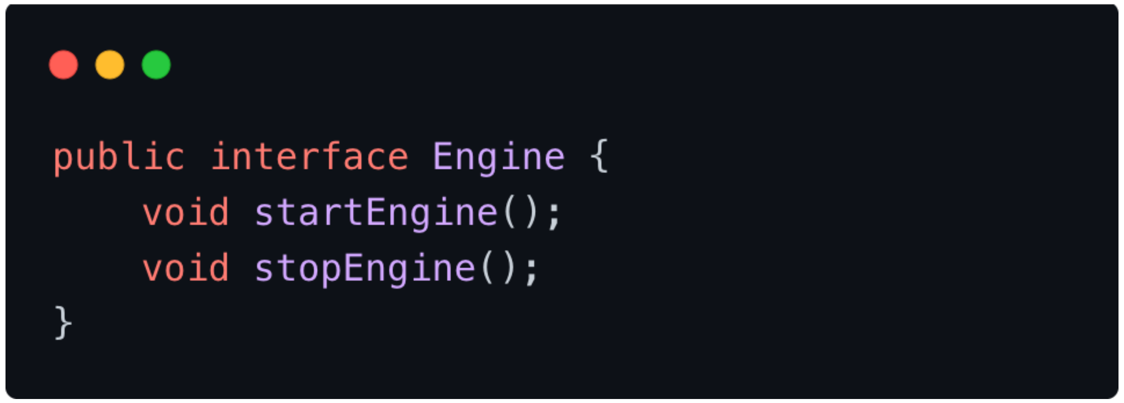
- An interface is a completely abstract class (mostly).
- It acts as a Contract.
- A class uses implements (not extends).
- Power: Java allows implementing multiple interfaces!



```
interface Animal {
    void makeSound(); // abstract by default
}

interface CanRun {
    void run();
}

class Dog implements Animal, CanRun {
    @Override
    public void makeSound() {
        System.out.println("Woof!");
    }

    @Override
    public void run() {
        System.out.println("Dog is running!");
    }
}
```

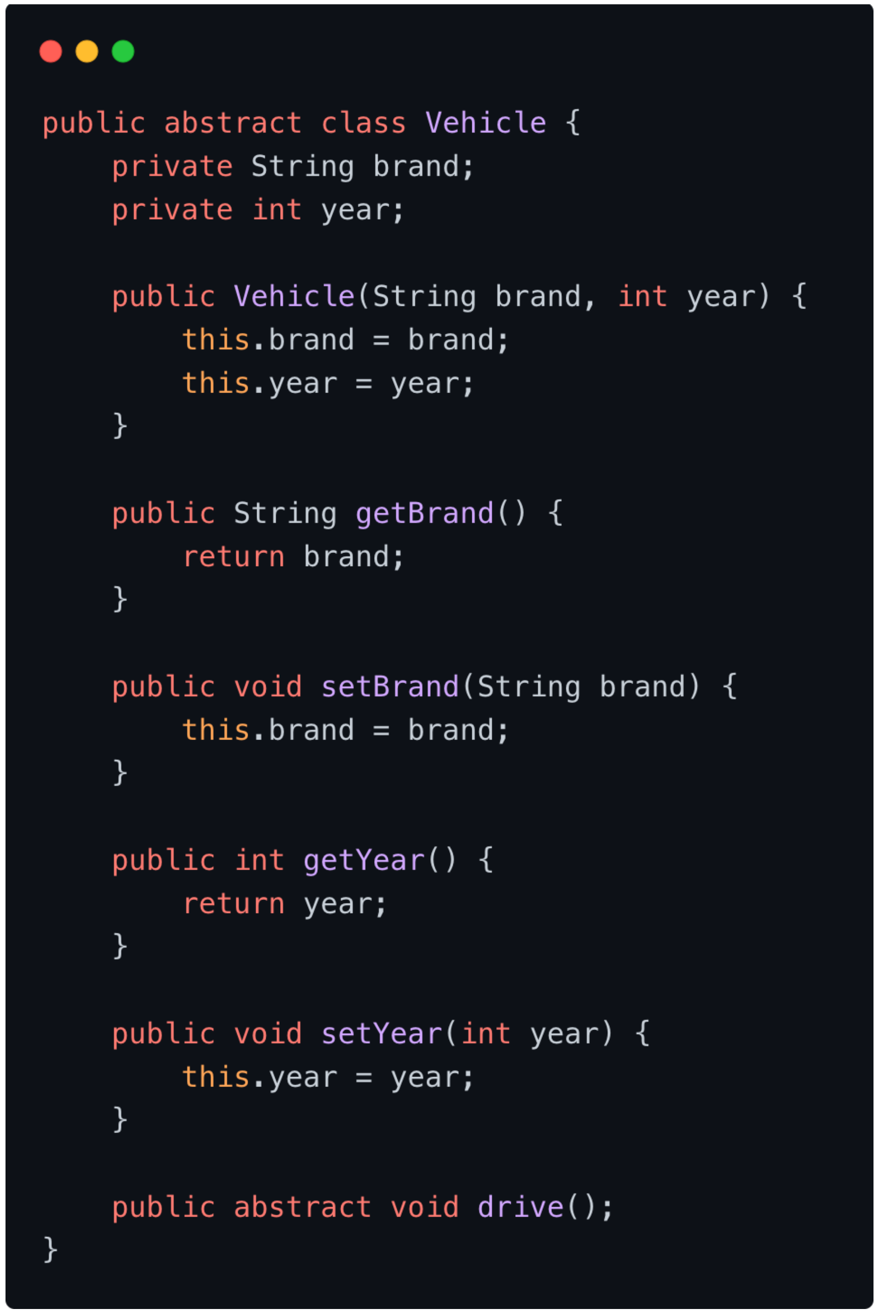
Example

Engine.java (Interface)



```java
public interface Engine {
    void startEngine();
    void stopEngine();
}
```

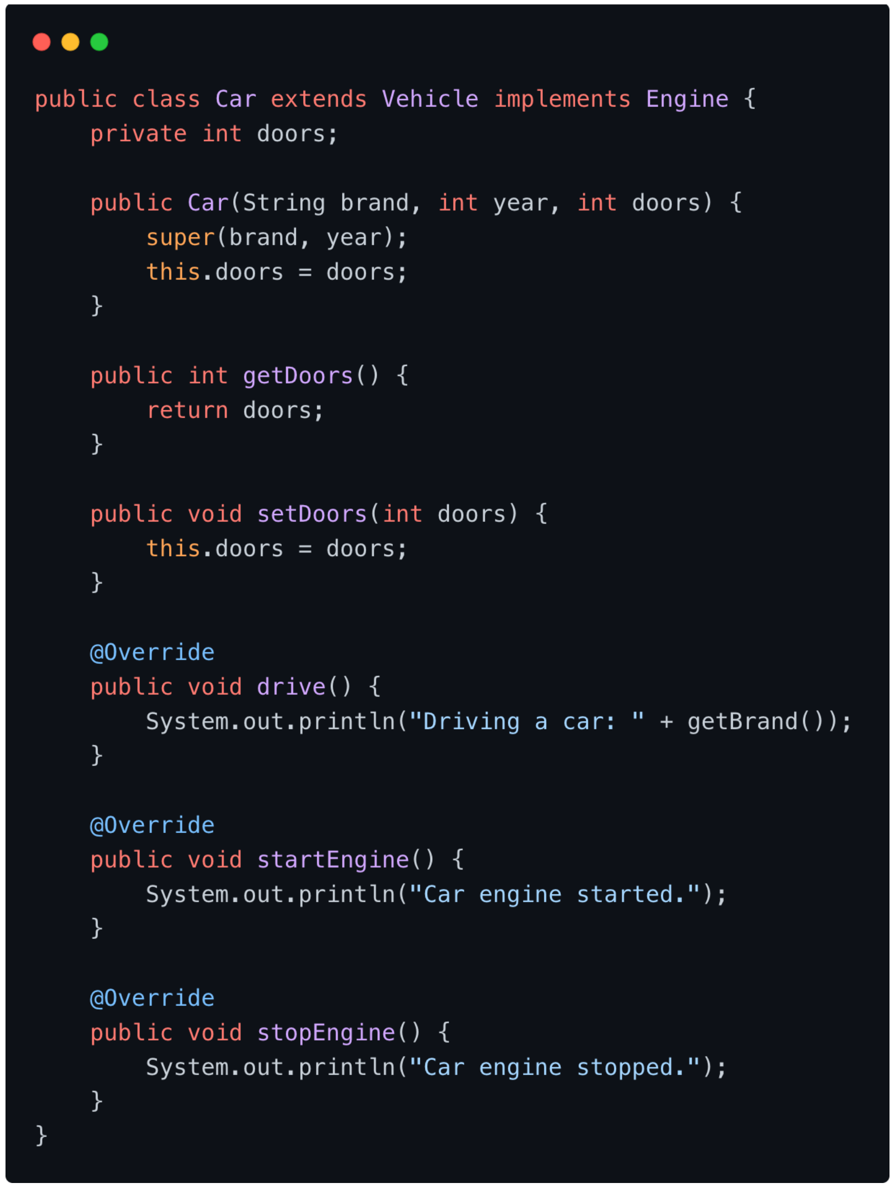
Vehicle.java (Abstract Class)



```java
public abstract class Vehicle {
    private String brand;
    private int year;

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

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }

    public abstract void drive();
}
```

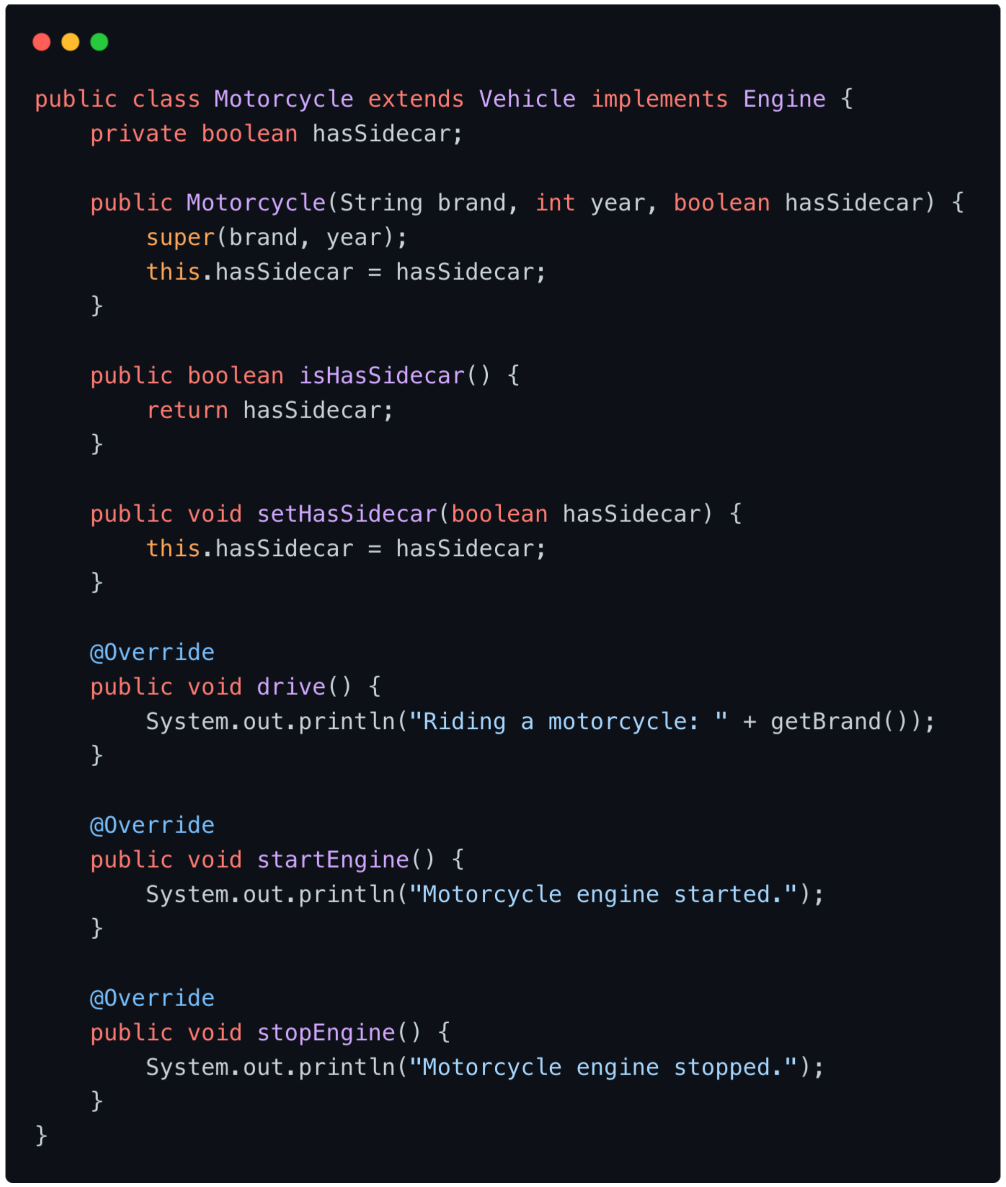
Example

Car.java (Subclass)



```java
public class Car extends Vehicle implements Engine {
    private int doors;

    public Car(String brand, int year, int doors) {
        super(brand, year);
        this.doors = doors;
    }

    public int getDoors() {
        return doors;
    }

    public void setDoors(int doors) {
        this.doors = doors;
    }

    @Override
    public void drive() {
        System.out.println("Driving a car: " + getBrand());
    }

    @Override
    public void startEngine() {
        System.out.println("Car engine started.");
    }

    @Override
    public void stopEngine() {
        System.out.println("Car engine stopped.");
    }
}
```

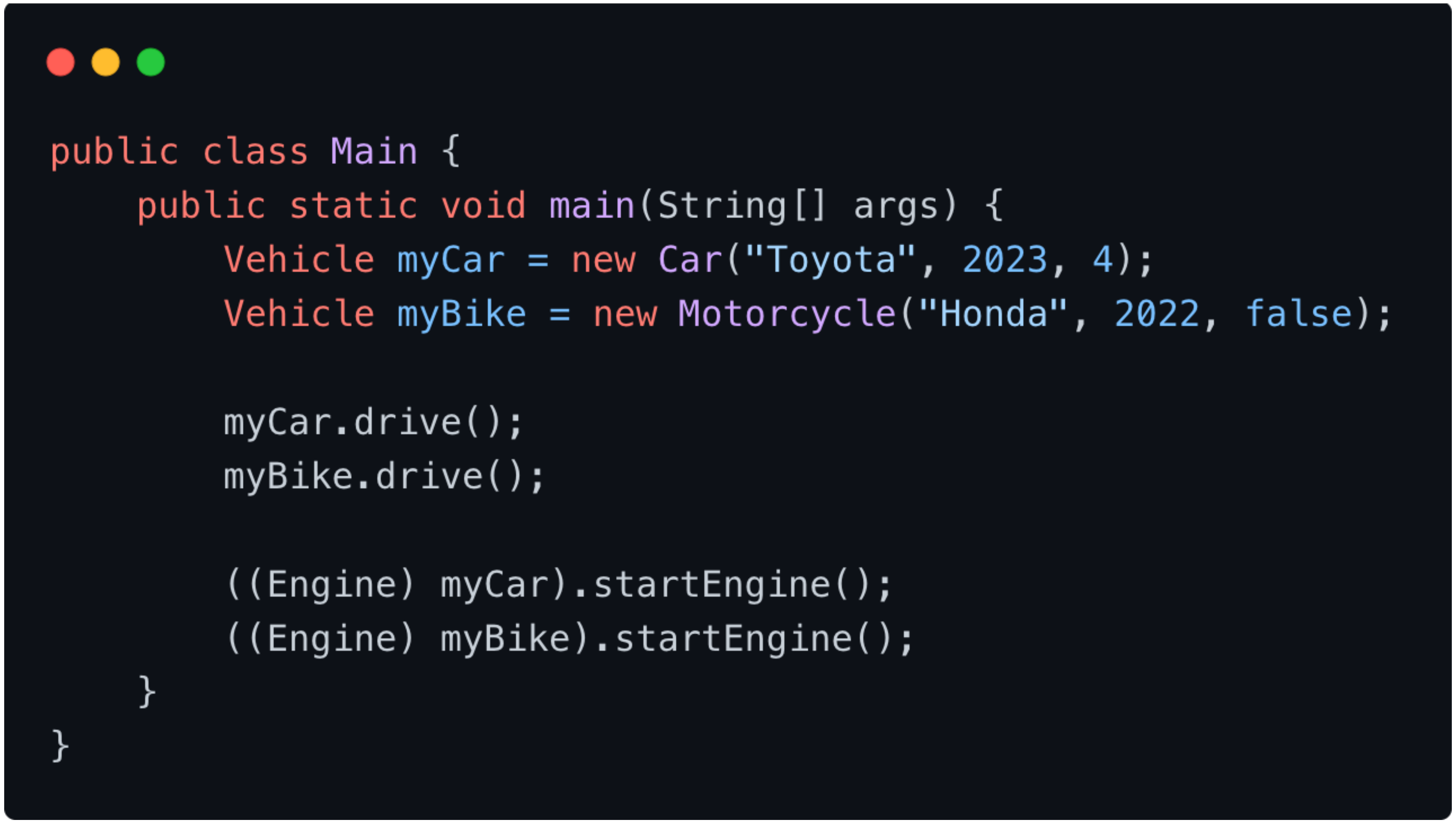
Motorcycle.java (Subclass)



```java
public class Motorcycle extends Vehicle implements Engine {
    private boolean hasSidecar;

    public Motorcycle(String brand, int year, boolean hasSidecar) {
        super(brand, year);
        this.hasSidecar = hasSidecar;
    }

    public boolean isHasSidecar() {
        return hasSidecar;
    }

    public void setHasSidecar(boolean hasSidecar) {
        this.hasSidecar = hasSidecar;
    }

    @Override
    public void drive() {
        System.out.println("Riding a motorcycle: " + getBrand());
    }

    @Override
    public void startEngine() {
        System.out.println("Motorcycle engine started.");
    }

    @Override
    public void stopEngine() {
        System.out.println("Motorcycle engine stopped.");
    }
}
```

Example

Main.java (Main Class)



```
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car("Toyota", 2023, 4);
        Vehicle myBike = new Motorcycle("Honda", 2022, false);

        myCar.drive();
        myBike.drive();

        ((Engine) myCar).startEngine();
        ((Engine) myBike).startEngine();
    }
}
```

Summary

- Encapsulation: Protect data using private and Getters/ Setters.

- Inheritance: Reuse code using extends and super.

- Polymorphism: Write flexible code using Overriding, Abstract classes, and Interfaces.