⋆˚₀⋆✧˚ Chapter 2: Core concepts of OOP ˚✧⋆₀˚⋆

The Object-Oriented Programming (OOP) paradigm shifts focus from linear logic to modular, real-world modeling. By organizing code into self-contained entities, OOP enables the creation of scalable and maintainable software that can adapt to complex requirements.

In this chapter, we explore the essential principles of this approach. We will define the relationship between **Classes** and **Objects**, examine how data and behavior are encapsulated, and study the mechanisms through which objects communicate via **Messages**. This conceptual shift is the first step toward mastering professional Java development and architectural design.

1. Object

In the physical world, objects are everywhere: a cat, a desk, or a bicycle. Every real-world object is defined by two fundamental characteristics: **State** and **Behavior**.

- **State:** The attributes or data an object possesses (e.g., a cat has a name, color, and breed).
- **Behavior:** The actions an object can perform (e.g., a cat can meow, hunt, or stretch).

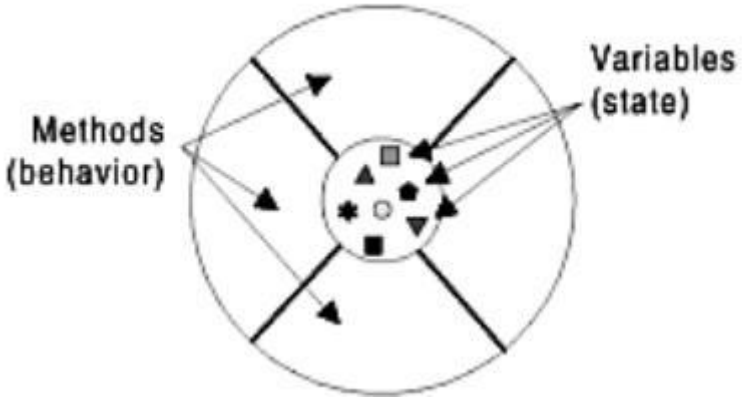

Figure 6. Anatomy of a Software Object

Software objects are modeled after these real-world counterparts. In a Java program, an object is a self-contained unit that bundles data and functionality together:

- **State (Variables):** The data maintained by the object. For a bicycle object, variables might store the current speed (10 mph) or the current gear (5th).
- **Behavior (Methods):** The functions or subroutines that define what the object can do. A bicycle object would include methods to brake, accelerate, or change gears.

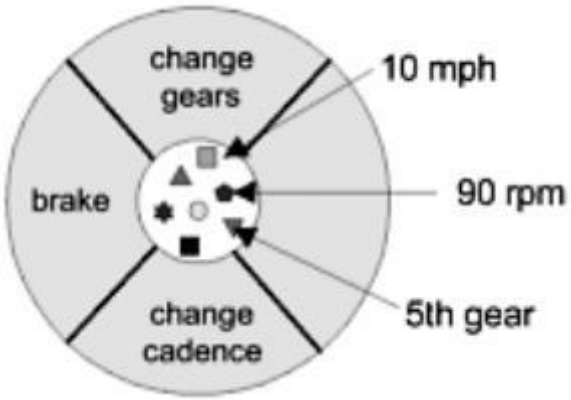

Figure 7. The real object bicycle modeled by a code object

By structuring programs around objects, Java provides two critical software engineering benefits:

- **Modularity:** Objects are independent units. You can modify the source code of one object without affecting others, making the system easier to build, test, and reuse.
- **Information Protection:** Objects control what information they reveal. By hiding complex internal logic and exposing only necessary methods, an object protects its data and simplifies how other parts of the program interact with it. For example, you can shift gears on a bicycle without needing to understand the mechanical complexity of the chain and derailleurs.

2. Communication via messages

In a real-world scenario, a single object is rarely useful in isolation. For example, a bicycle hanging in a garage is just a collection of metal and rubber; it only fulfills its purpose when another object (a rider) interacts with it. Similarly, a Java program consists of multiple objects that interact to achieve complex functionality.

Software objects communicate by sending **messages**. When Object A needs Object B to perform a specific action, Object A invokes one of Object B's methods. This request is known as "sending a message."

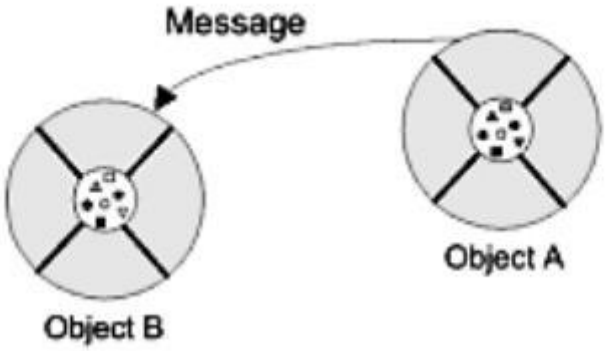

Figure 8. Object interaction using messages.

Often, the receiving object requires specific details to execute a task correctly. For instance, telling a bicycle to "change gears" is not enough; we must specify *which* gear. This extra data is passed along with the message as **parameters**.

A standard message consists of three essential components:

- **The Recipient:** The specific object to which the message is sent (e.g., myBicycle).
- **The Method:** The name of the action to be performed (e.g., changeGears).
- **The Parameters:** Any additional information required by the method (e.g., lowerGear).

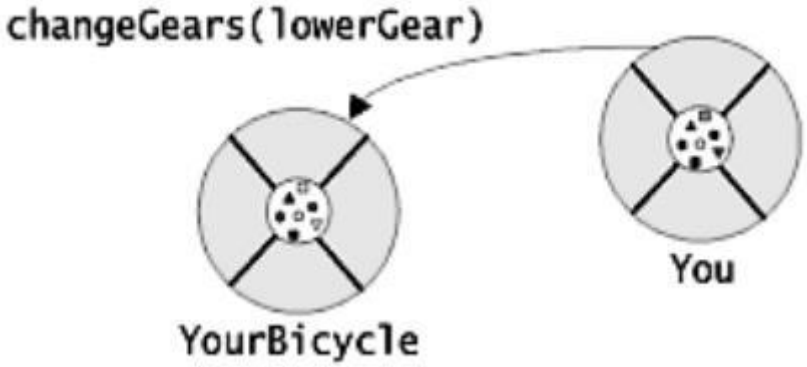

Figure 9. Example of a message in Java.

By utilizing these message components, objects can work together seamlessly while remaining distinct, modular entities.

3. The concept of classes

In the real world, we often encounter many objects of the same kind. Your bicycle is not unique in its basic design; it is one of millions that share the same fundamental structure. In Object-Oriented Programming, we say that your bicycle is an **instance** of a general category known as a **Class**.

We can see a class as a **prototype** or a **blueprint**. It would be highly inefficient for a factory to design a new set of plans for every single bicycle they build. Instead, they create one master blueprint and use it to manufacture thousands of identical bikes.

- **The Class:** The blueprint that defines what variables and methods all objects of that type will have.
- **The Object:** The actual, physical bicycle created from that blueprint.

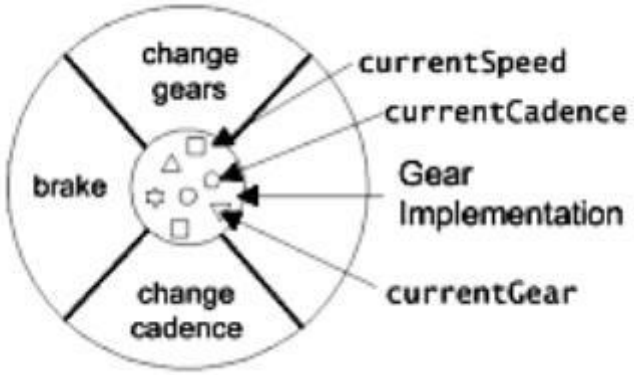

Figure 10. The Bicycle class acting as a blueprint for multiple objects.

When we create objects from a class, they all possess the same variables, but their internal data can differ.

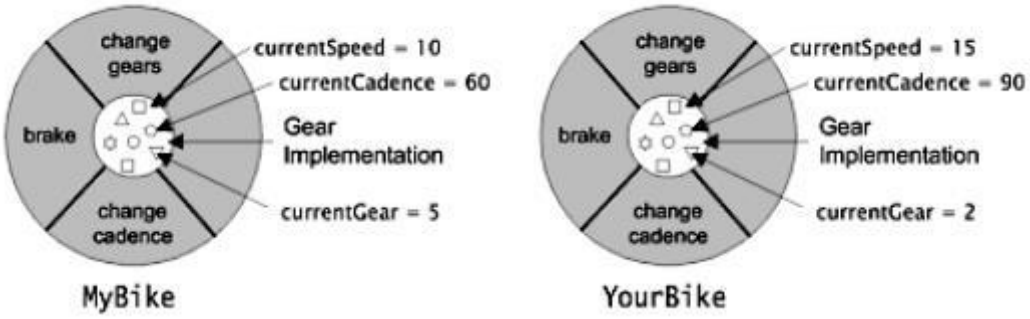

Figure 11. Two bicycle object instances sharing a class variable.

To distinguish between individual data and shared data, Java categorizes variables into two types:

- **Instance Variables:** These belong to the specific object. Your bicycle might be in **3rd gear** while your friend's bicycle is in **5th gear**. Each "instance" maintains its own state.
- **Class Variables:** These contain information shared by *every* object created from that class. For example, if every bicycle model produced by a factory is designed with exactly **18 gears**, that value is stored in a class variable. Every individual bike instance then references this shared value.

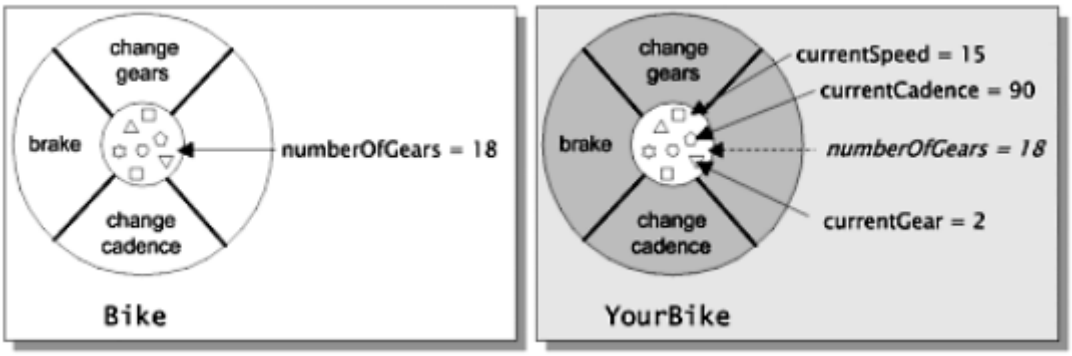

Figure 12. Relationship between a shared Class Variable and specific Instance Variables

While objects provide **modularity** and **protection**, classes provide **reusability**. Just as a manufacturer reuses a blueprint to save time and resources, a programmer reuses a class to create thousands of objects without rewriting the code. By defining the logic once in a class, you can deploy it infinitely across your application.

4. The concept of inheritance

In Object-Oriented Programming, classes can be organized into hierarchies. Often, different types of objects share many characteristics but remain distinct in their specific functions. For example, **Mountain Bikes**, **Racing Bikes**, and **Tandems** are all different, yet they are all fundamentally **Bicycles**.

Inheritance allows one class to derive its state and behavior from another.

- **Superclass (Parent):** The general class that provides common features (e.g., Bicycle).
- **Subclass (Child):** The specialized class that inherits from the superclass (e.g., MountainBike).

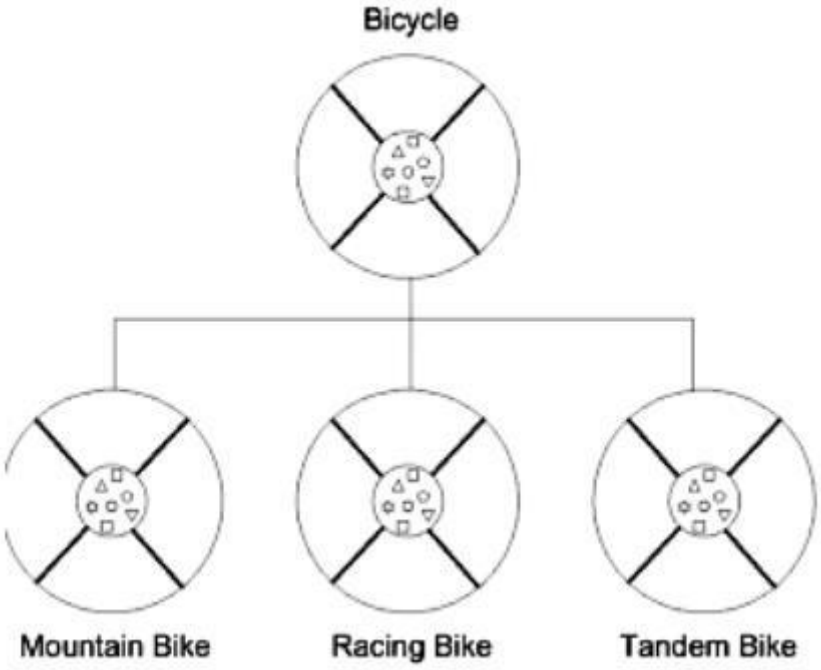

Figure 7: Hierarchy of the Bicycle superclass and its specialized subclasses.

Through inheritance, the relationship between a parent and child class is defined by three primary mechanisms:

- **Shared State and Behavior:** Every subclass automatically inherits the variables (cadence, speed) and methods (braking, shifting) of the superclass. This avoids redundant code.
- **Specialization:** Subclasses can define their own unique features. For instance, a TandemBike might add a variable for secondSeat, while a MountainBike might include an additional suspension method.
- **Method Overriding:** A subclass can modify an inherited behavior to fit its specific needs. If a MountainBike has an extra set of gears, it can **override** the standard changeGears method to include logic for the lower gear ratio.

Inheritance promotes **reusability** and **extensibility**. Instead of writing the same "braking" logic for every type of bike, a programmer writes it once in the Bicycle superclass. Subclasses then simply inherit that logic and add only what makes them unique, creating an efficient and organized "inheritance tree."

5. The concept of interface

In everyday life, an **interface** is a bridge that allows unrelated entities to interact. For example, a remote control serves as the interface between a human and a television, and a common language acts as an interface between two people.

In Java, an interface is a functional contract that allows unrelated objects (objects from different class hierarchies) to communicate. While inheritance links related objects (like different types of bikes), an interface links objects that might have nothing in common except for a specific capability.

It is important to note that an interface is **not a class**. It follows a distinct set of rules:

- **No State:** Unlike a class, an interface does not contain instance variables to store data.
- **Method Signatures:** It only contains constants and "generic" methods (declarations without the actual code).
- **Implementation:** The classes that use the interface are responsible for providing the actual implementation of those methods.

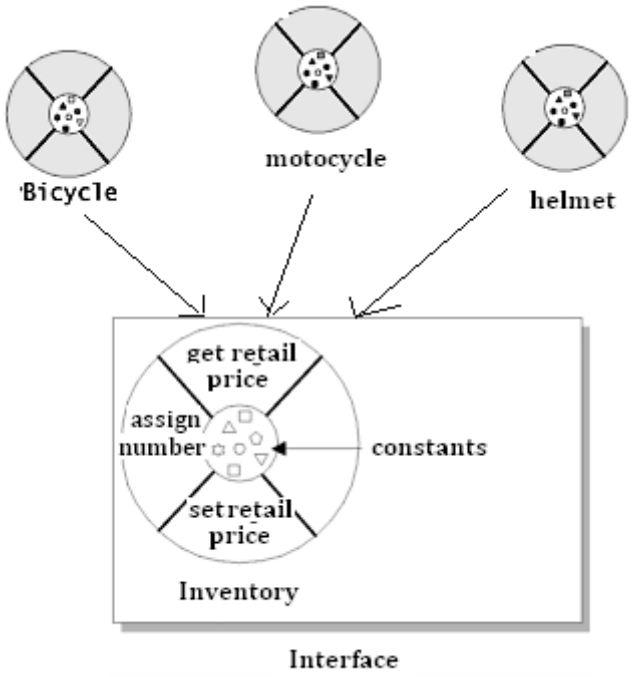

Figure 13: Unrelated objects (Bicycle, Motorcycle, Helmet) utilizing a common interface to manage pricing and tracking.

Imagine a store that sells bicycles, protective helmets, and motorcycles. the **Inventory Interface** acts as a professional contract. It doesn't care *what* an object is (a vehicle or a

piece of clothing); it only cares that the object can perform specific inventory tasks. The two methods shown are "requirements" that every object must fulfill:

- **get retail price / set retail price:** This allows the system to handle money. The **Bicycle** might calculate its price based on parts, while the **Motorcycle** includes taxes and registration. Because of the interface, the cash register doesn't need to know these details—it just asks for the "Price," and the object provides it.
- **assign number:** This is the "Tracking" requirement. Every item in a store needs a unique ID (like a barcode). By implementing this method, the **Helmet**, **Bicycle**, and **Motorcycle** all agree to accept a tracking number so they can be stored in the same database.

6. Polymorphism

In biology, polymorphism refers to the ability of an organism or species to exist in many different forms. In Object-Oriented Programming, **polymorphism** is the principle that objects belonging to the same class hierarchy can exhibit different behaviors. Essentially, it means "one interface, many implementations."

Polymorphism allows objects to perform the same action in different ways. For example, consider a general Bicycle class that includes a brake method. Even though two objects are both "bicycles," how they execute that brake command can differ:

- **MyBike:** Uses a simple braking system. When the brake method is called, it requires no extra information—it just slows down.
- **YourBike:** Features a more complex double-brake system. When its brake method is called, it requires a **parameter** (such as "front" or "rear") to know which specific brake to apply.

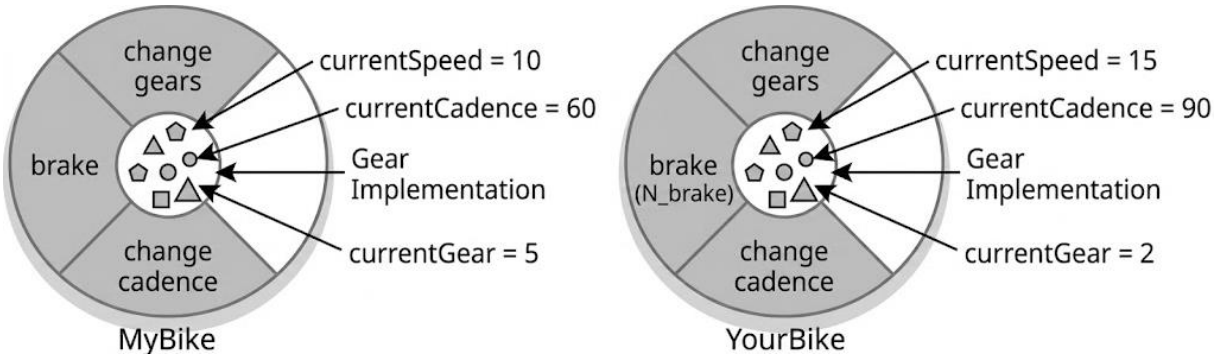

Figure 14. Two object instances (MyBike and YourBike) of the same Bicycle class exhibiting different behaviors.

Polymorphism allows a programmer to treat different objects as if they were the same type while letting each object handle its own specific logic.

Imagine you have a list of various bikes (Mountain Bikes, Racing Bikes, etc.). You can tell the entire group to brake(). Because of polymorphism:

- The **Mountain Bike** might use heavy-duty disc brakes.
- The **Racing Bike** might use lightweight rim brakes.
- The **Tandem** might engage brakes on both sets of wheels.

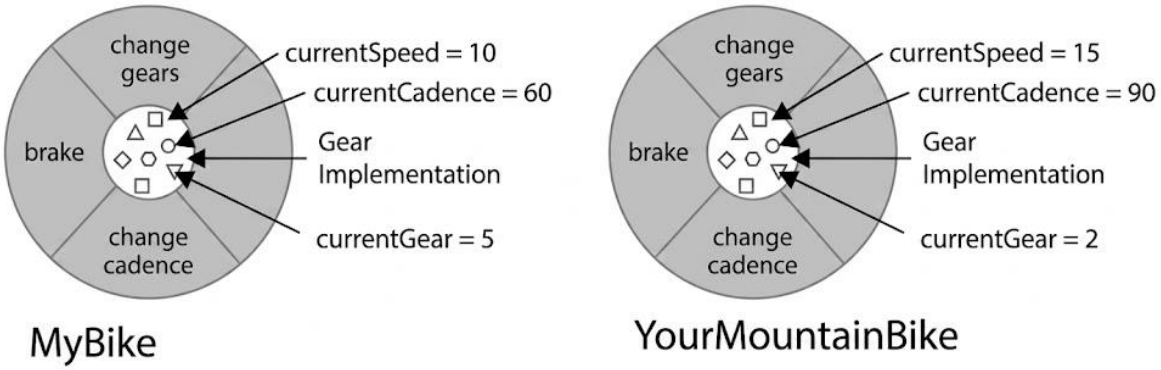

Figure 15: Polymorphism in action across different bicycle types in the same hierarchy.

Benefits of Polymorphism

- **Flexibility:** You can add new types of objects (like an ElectricBike) to your program without changing the code that manages the bikes. You simply define how that new bike "brakes."

- **Simplicity:** It reduces complexity by allowing you to use the same method name (brake) for different actions, rather than having to remember unique names like mountainBikeBrake() or racingBikeBrake().

7. Abstract classes

In some cases, you may want to design a class that represents a general concept rather than a specific, finished product. This is where **Abstract Classes** come into play.

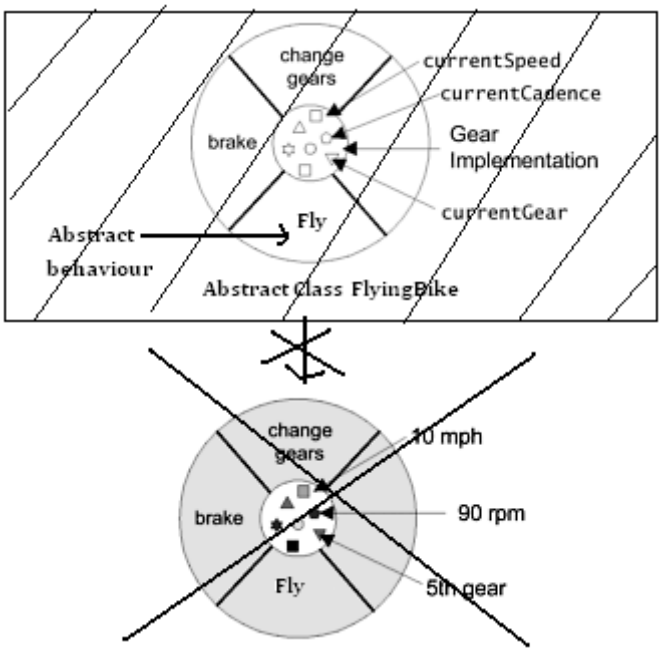

Figure 16. Attempted instantiation of the Abstract Class FlyingBike and why it is prohibited.

Imagine you want to invent a futuristic "Flying Bicycle." You know it should have the ability to fly, but at the design stage, you haven't yet figured out the mechanical details of *how* it stays airborne (e.g., propellers vs. jet engines).

In programming, you would declare the fly() behavior without writing the code for it. This is called an **abstract method**. A class that contains at least one abstract method is an **Abstract Class**.

To ensure these classes are used correctly within a program, they are governed by three fundamental rules:

- **No Direct Objects:** You cannot create an object from an abstract class. Since the class is "incomplete" (like a blueprint missing the engine schematics), the computer wouldn't know what to do if you tried to use it.
- **The Concrete Requirement:** To actually use an abstract class, you must first create a **subclass** that inherits from it. This subclass must provide the "concrete" implementation (the actual code) for all the abstract methods.
- **Purpose:** Abstract classes act as a strict template. they ensure that all future specialized versions of the class follow a specific structure while allowing each version to handle the "how" differently.

To visualize how this concept moves from an idea to a functional object, consider this step-by-step progression:

- **Abstract Superclass:** FlyingBicycle (defines that it *must* fly, but not how).
- **Concrete Subclass:** PropellerBike (inherits from FlyingBicycle and implements the fly() method using propeller logic).
- **Result:** You can now create an object of PropellerBike because it is a finished, "concrete" version of the abstract idea.

8. Encapsulation

Encapsulation is the principle of shielding an object's internal state by wrapping its variables within protective methods. In simpler terms, it is the practice of "hiding" the data so that it can only be accessed or modified through authorized behaviors.

Imagine the currentSpeed variable of a bicycle. In a well-encapsulated system, you cannot simply reach into the object and manually type a new number for the speed. Instead, the speed changes only as a result of a specific action, such as invoking the changeGears or applyBrakes methods.

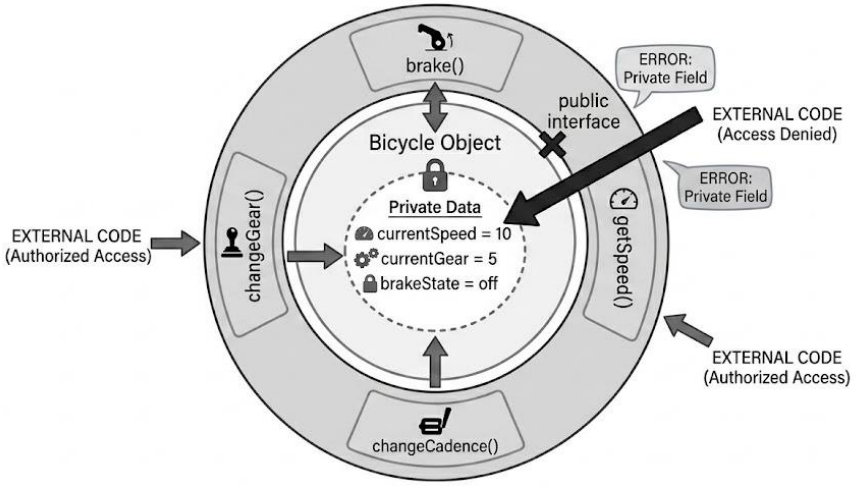

Figure 17: Encapsulation acting as a protective barrier around an object's data.

By implementing these protective boundaries, developers gain several critical technical advantages:

- **Information Protection:** This keeps the internal "secrets" of how an object works hidden. By only showing what is necessary, you prevent outside parts of the program from messing with the object's internal logic.
- **Data Integrity:** This acts like a quality control check. It ensures that the information inside an object always "makes sense" because it can only be changed through methods that follow the rules of the program.
- **Prevents Corruption:** By restricting direct access, you prevent external code from accidentally setting a variable to an impossible value (like setting a bicycle's speed to -50 mph).
- **Simplified Interaction:** Other parts of the program don't need to know exactly how a value is calculated; they just need to know which method to call to get the result they want.
- **Ease of Maintenance:** If you want to change how the speed is calculated later, you only have to fix the code inside that one method instead of searching through your entire project for every time the speed variable was mentioned.

9. From principle to implementation

After understanding the core theories of Object-Oriented Programming (OOP), we will now implement them using **JAVA**. In this course, Java serves as our primary tool to bring these concepts to life.

➢ Class Definition and Visibility

To define a class in Java, we use the class keyword along with **Access Modifiers** to control data security:

- **public**: Visible to all classes.
- **protected**: Visible to the class and its subclasses.
- **private**: Visible only within the class (The basis of **Encapsulation**).

```
class Bicycle {

    public int cadence = 0;   // Public attribute

    protected int gear = 1;   // Protected attribute

    private int speed = 0;   // Private attribute (Encapsulated)

    void changeCadence(int newValue) { cadence = newValue; }

    void changeGear(int newValue) { gear = newValue; }

    void speedUp(int increment) {

        speed = speed + increment;

    }

    void applyBrakes(int decrement) {

        speed = speed - decrement;

    }

    void printStates() {

        // System.out.println is equivalent to printf in C

        System.out.println("cadence:" + cadence + " speed:" +
speed + " gear:" + gear);
```

```
    }
}
```

- **Object Creation (Instantiation)**

To create an object, we use the **new** keyword. This allocates a block in the **RAM (Heap)** for the object's attributes and methods.

Syntax: `Bicycle bike1 = new Bicycle();`

- **Inheritance**

Inheritance allows a **Subclass** to acquire the properties of a **Superclass** using the keyword **extends**.

```
class MountainBike extends Bicycle {
    public int length;
    void print() {
        System.out.println("length:" + length);
    }
}
```

To instantiate a Subclass object: `MountainBike obj = new MountainBike();`

obj has access to all methods in Bicycle AND MountainBike.

To instantiate a Superclass object: `Bicycle objs = new MountainBike();`

objs can **only** access methods defined in the Bicycle class.

- **Interfaces and Abstract Classes**

An interface is a contract. Classes use the keyword **implements** and must define all listed methods.

```
interface Bicycle {
    void speedUp(int increment);
}
```

```
class implementedBicycle implements Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

void speedUp(int increment) { speed = speed + increment; }

}
```

A class declared with **abstract** cannot be instantiated. It can contain **abstract methods** (methods without a body) that subclasses must fill in.

```
abstract class FlyBicycle {

    abstract void fly();

}
```

If we want to implements the abstract method:

```
class Bicycle extends FlyBicycle {

void Fly() { // code here

}

}
```

- **Polymorphism**

Java supports three main types of polymorphism:

- **Method Overloading:** Same method name, different parameters (within the same class or class hierarchy).

```
class Bicycle {

// …… int speed = 0;

void applyBrakes() { speed = speed - 1;

}

applyBrakes(int decrement) {
```

```
speed = speed - decrement; }

// ……

}
```

- **Method Overriding:** A subclass redefines a method from the superclass to change its behavior.

```
class MountainBike extends Bicycle {

// this class inherit from Bicycle class

void applyBrakes() {

speed = speed – 2;} // override applyBrakes() of Bicycle

}

}
```

- **Constructor Polymorphism:** Multiple ways to initialize an object.

A Constructor is a special type of method used to initialize an object. It sets the initial state (attributes) of the object at the moment of creation. It must have the exact same name as the Class and it never has a return type (no void, int, etc.). It is called automatically when we use the **new** keyword.

We distinguish two type of constructor:

A. Default Constructor

If you do not write any constructor, Java automatically provides a "default" one. It initializes numerical variables to 0 and objects/strings to null.

```
class Bicycle {

    int cadence = 0;

    int speed = 0;

    int gear = 1;

    void printStates() {
```

```
        System.out.println("cadence:" + cadence + " speed:" +
speed + " gear:" + gear);
    }
}
class BicycleDemo {
    public static void main(String[] args) {
      // Calling the system-defined default constructor
      Bicycle bike1 = new Bicycle();
      bike1.printStates(); // Output: cadence:0 speed:0 gear:1
    }
}
```

B. User-Defined Constructors

When you want specific initial values, you define your own constructors. There are three main variations:

a. Non-parameterized Constructor

Used to set fixed default values for every new object.

```
class Bicycle {
    int cadence, speed, gear;
    public Bicycle() {
        cadence = 0;
        speed = 0;
        gear = 0;
    }
    // Note: You can have a method with the same name if it
has a return type
```

```
    void Bicycle() {
        System.out.println("This is a regular method, not a
constructor!");
    }
}
```

b. Parameterized Constructor

Allows you to pass specific values for some attributes when creating the object.

```
class Bicycle {
    int cadence, speed, gear = 1;
    public Bicycle(int c, int s) {
        cadence = c;
        speed = s;
    }
}
```

c. Immediate (Full) Constructor

Used when you want to initialize all attributes at once.

```
class Bicycle {
    int cadence, speed, gear;
    public Bicycle(int c, int s, int g) {
        cadence = c;
        speed = s;
        gear = g;
    }
}
```

Note: If you define any user-defined constructor, Java will no longer provide the automatic default constructor. You must define a no-argument constructor manually if you still wish to use it.

In Java, you can have multiple constructors in the same class as long as they have different parameter lists. This is known as Constructor Overloading. Java decides which one to use based on the arguments you provide.

```
class Bicycle {

    int cadence, speed, gear;

    // Constructor 1: Full initialization

    public Bicycle(int c, int s, int g) {

        cadence = c;

        speed = s;

        gear = g;

    }

    // Constructor 2: Default fixed values

    public Bicycle() {

        cadence = 20;

        speed = 5;

        gear = 1;

    }

    // Constructor 3: Partial initialization

    public Bicycle(int c, int s) {

        cadence = c;

        speed = s;

        gear = 0; // Default if not provided

    }
```

```
    void printStates() {
        System.out.println("cadence:" + cadence + " speed:" +
speed + " gear:" + gear);
    }
}
class BicycleDemo {
    public static void main(String[] args) {
        Bicycle bike1 = new Bicycle();            // Uses
Constructor 2
        Bicycle bike2 = new Bicycle(50, 10, 2);   // Uses
Constructor 1
        Bicycle bike3 = new Bicycle(63, 12);      // Uses
Constructor 3
        bike1.printStates(); // cadence:20 speed:5 gear:1
        bike2.printStates(); // cadence:50 speed:10 gear:2
        bike3.printStates(); // cadence:63 speed:12 gear:0
    }
}
```

Unlike languages like other language like C++, Java does not use destructors. Memory management is handled by the Garbage Collector. This is an automatic tool that identifies objects that are no longer being used by the program and destroys them to free up memory space automatically.

- **Encapsulation**

Encapsulation protects data integrity by hiding variables behind a private barrier. Users must use public methods to interact with the data.

```
class Bicycle {
    private int cadence = 0; // Encapsulated
```

```
    public int gear = 1;     // Not encapsulated
    public int getcadence() { // encapsulation
          return cadence;  }
}
// …..
// accessing stats
bike1.gear = 3;      // Works
bike1.cadence = 10;  // ERROR: cadence is private!
int x = bike1.getcadence();  // Works
```