In previous chapters, we mastered procedural programming by controlling the flow of instructions using decision structures (Chapter 3) and repetition (Chapter 4), and organizing reusable code into methods (Chapter 5). This process, known as **functional decomposition**, helped us break down large problems into smaller, manageable sub-problems.

Now, we shift our focus entirely to **Object-Oriented Programming (OOP)**, which is the defining approach of the Java language. OOP revolves around organizing data (fields) and operations (methods) together into self-contained software components called **objects**.

This chapter provides the blueprint for building those objects by introducing the concept of a **Class**.

What you will learn

In this chapter, you will learn how to:

- Understand the relationship between **classes (blueprints)** and **objects (instances)**.
- Differentiate between executable classes and object classes.
- Define a class structure using **instance fields** and **instance methods**.
- Implement **data hiding** using the `private` access specifier.
- Create public **accessor (getter)** and **mutator (setter)** methods to manage private data.
- Differentiate between **static** (class) methods and **instance** (object) methods.
- Define and use **constructors** (including no-arg and parameterized) to initialize objects.
- Use method and constructor **overloading** to provide flexibility.
- Pass objects as arguments to methods and understand how **references** are handled.
- Override the default `toString()` and `equals()` methods for custom object behavior.
- Understand object design principles, including packages and scope. # 6.1 Classes are Blueprints for Data

A **class** is a blueprint or template that defines the properties (**fields**) and behaviors (**methods**) that objects created from it will possess.

We are familiar with Java Classes such as `Random`, `File`, `PrintWriter`, `Scanner`, and `String`, which we have used to create objects.

An **object** is a specific instance of a class, created in memory based on that blueprint. Here are some examples of how to create objects from classes:

```java
// Create an instance of the Random class called randomNumber.
Random randomNumber = new Random();
```

```java
// Create an instance of the File class called quotesFile.
// It takes one input parameter, a filename.
File quotesFile = new File("quotes.txt");

// Create an instance of the Scanner class called kbd.
// It takes one input parameter, the System input (keyboard).
Scanner kbd = new Scanner(System.in);

// Create an instance of the Scanner class called inputFile.
// It takes one input parameter, the File object called "quotesFile".
Scanner inputFile = new Scanner(quotesFile);

// Create an instance of the String class called name.
// String is the only Java Class that doesn't require a constructor.
// It takes one input parameter, a name.
String name = "Nancy";
```

A class specifies the **fields** and **methods** for a particular type of object.

Primitive Variables vs. Object Reference Variables

Variables in Java are categorized by what they store:

Variable Type	Data Storage	Example
Primitive Variables	Stores the **actual data value** directly in memory (e.g., `int`).	`int rating = 5;`
Object Reference Variables	Stores the **memory address** (reference) where the object's data is located.	`String name = "Zoe";`

Two Types of Classes

For practical development, we categorize classes based on their purpose:

1. **Object Classes (Model Classes):** These are the **blueprints** (e.g., `Car.java`, `Song.java`). They define the structure and behavior (fields and methods) of a custom object. They do not contain a `main` method.
2. **Executable Classes (Test Classes):** These contain the `public static void main(String[] args)` method. They serve as the program's starting point and are used to **instantiate** (create) and utilize objects defined by Object Classes.

To create an object from a class, we use the `new` keyword, which calls a constructor.

Example: The Car Blueprint and the Test Drive

We define a `Car` class and then create instances of it in a separate executable class.

```java
// Car.java (Object Class/Blueprint)
public class Car {
    // Fields (Data associated with this Car object)
    String make;
    String model;

    // Method (Behavior this Car object can perform)
    public void startEngine() {
        System.out.println(make + " " + model + " engine starting up.");
    }
}

// TestCar.java (Executable Class)
public class TestCar {
    public static void main(String[] args) {
        // Instantiate the first Car object
        Car sedan = new Car();

        // Access and set the object's fields directly (not recommended, see 6.2)
        sedan.make = "Honda";
        sedan.model = "Civic";

        System.out.println("Object 1 created: " + sedan.make);
        sedan.startEngine();

        // Instantiate a second, independent Car object
        Car truck = new Car();
        truck.make = "Ford";
        truck.model = "F-150";
        truck.startEngine();
    }
}
```

Output:

```
Object 1 created: Honda
Honda Civic engine starting up.
Ford F-150 engine starting up.
```

Check Your Understanding (6.1)

1. What is the key functional difference between an executable class and an object class?

2. Why is the variable `sedan` in `Car sedan = new Car();` considered a reference variable, and what does it store?
3. A class specifies the **fields** and **methods** for a particular type of object. What do these two terms refer to in practical programming?

Practice Problems (6.1)

1. **Conceptual Design (Music):** Imagine a class called `Playlist`. Name three data points (fields) a `Playlist` object should store and one action (method) it should be able to perform.
2. **Instantiation:** Write the single Java statement required to declare and instantiate a new object named `myCar` from the `Car` class (assuming no custom constructor exists). # 6.2 Writing an Object Class: Fields, Accessors, and Data Hiding

A high-quality Object Class prioritizes **data hiding** to ensure data integrity. This is achieved by making the data fields private and only allowing manipulation through public methods.

Data Hiding and the `private` Keyword

Instance fields are the data components of an object. Good object-oriented programming practice dictates making all of a class's fields **private**.

The **`private`** keyword is an access modifier that restricts access to the field only to methods **within the same class**. This prevents external classes (like our `TestCar.java` from Section 6.1) from directly accessing or corrupting the object's internal state.

Accessor (Getter) and Mutator (Setter) Methods

Since the fields are private, we need controlled pathways for other code to interact with the data:

- **Accessor Method (Getter):** A public, **value-returning** instance method used to **retrieve** (get) the value of a private field.
- **Mutator Method (Setter):** A public, typically **`void`** instance method used to **modify** (set) the value of a private field. Mutator methods are often used to implement input validation.

Example 6.2: Designing a Song Class with Data Hiding

```java
// Song.java (Object Class/Blueprint demonstrating data control)
public class Song {
    // Private Instance Fields (Data Hiding)
    private String title;
    // Volume level must stay between 0 (silent) and 100 (max).
    private int volumeLevel;
    // Duration of the song in seconds. Note: No setter method.
```

```java
private int durationSeconds;

// --- MUTATOR (SETTER) METHODS ---

// Setter for title. Note the use of 'this' to avoid shadowing.
public void setTitle(String title) {
    this.title = title;
}

// Setter for volumeLevel, including validation logic using if-else-if.
public void setVolumeLevel(int level) {
    // Check if level is too low
    if (level < 0) {
        this.volumeLevel = 0;
    }
    // Check if level is too high
    else if (level > 100) {
        this.volumeLevel = 100;
    }
    // If valid, set the level
    else {
        this.volumeLevel = level;
    }
}

// --- ACCESSOR (GETTER) METHODS ---
public String getTitle() {
    return title;
}

public int getVolumeLevel() {
    return volumeLevel;
}

public int getDurationSeconds() {
    return durationSeconds;
}

// Instance Method (Behavior that modifies the object's state)
public void increaseVolume() {
    System.out.println(title + " volume boosted by 15!");
    // We use the setter to update the state, which automatically ensures
    // the new level does not exceed the maximum of 100.
    setVolumeLevel(this.volumeLevel + 15);
    System.out.println("New volume level: " + this.volumeLevel);
}
```

```
}
```

The this Reference

When a method parameter has the same name as an instance field (e.g., `setTitle(String title)`), this causes **shadowing**. The local parameter takes precedence.

The **`this` reference variable** explicitly refers to the object's instance field, resolving this ambiguity. Example: `this.title = title;`

Avoiding Stale Data

You should avoid storing calculated data (e.g., area, age, total price) in a field if that data depends on other fields that can change. If the source data changes, the calculated value becomes **stale data**.

To prevent this, perform the calculation within a public instance method every time the value is requested.

```java
// Calculated Method Example for a Song (Calculates duration in minutes)
public double getDurationMinutes() {
    // Assuming private field int durationSeconds exists
    return (double)durationSeconds / 60.0;
}
```

Check Your Understanding (6.2)

1. What is the fundamental goal of declaring instance fields as `private`?
2. Which type of method, accessor or mutator, is usually responsible for validating input data?
3. Why should a `Rectangle` object have a `getArea()` method that calculates the area dynamically, rather than storing the area in a private field?

Practice Problems (6.2)

1. **Setter Implementation (Music):** A `Song` class has a private field `int playCount`. Write a public mutator method `incrementPlayCount()` that increases `playCount` by exactly one. (No parameter needed.)
2. **Getter Implementation (Car):** A `Car` class has a private field `boolean isAvailable`. Write the corresponding public accessor method `isAvailable()`. # 6.3 Static vs. Instance: Class-Level vs. Object-Level

In Chapter 5, we were introduced to **static** methods. Now that we are writing custom classes, understanding the distinction between static and instance members is critical.

The core difference lies in their **ownership** and **context**:

Feature	**Instance Members (Non-Static)**	**Static Members (Class Members)**
Ownership	Belongs to a specific **object** (instance) created via `new`.	Belongs to the **class itself**.
Access/Call	Must be called on an **object reference** (e.g., `myCar.startEngine()`).	Called directly on the **class name** (e.g., `Math.random()`).
Context	Operates on the object's specific, unique data (instance fields).	Cannot access instance fields or methods of the class, as there is no specific object context.
`this` keyword	Available, refers to the current object executing the method.	Prohibited, as there is no instance object to refer to.

Methods that operate on an object's specific private fields (like getters and setters) are always **instance methods**.

Example 6.3: Tracking Total Cars Built (Static Field)

If we wanted to track the total number of `Car` objects ever created, this count should be shared among *all* instances, making it a **static field** belonging to the class itself.

```java
public class Car {
    // Instance fields (unique to each object)
    private String make;

    // Static field (shared by all Car objects)
    private static int totalCarsBuilt = 0;

    // Constructor (omitted for brevity)

    // Static method (operates on the static field)
    public static int getTotalCarsBuilt() {
        return totalCarsBuilt;
    }

    // Instance method (operates on the instance field)
    public String getMake() {
        return this.make; // Uses 'this' reference
    }

    // Instance method
    public void buildCar() {
        // Instance method can update the static field
        totalCarsBuilt++;
        System.out.println("Car built. Total: " + totalCarsBuilt);
    }
}
// In TestCar:
// Car.getTotalCarsBuilt(); // Called on Class (static)
// Car mySedan = new Car();
// mySedan.buildCar(); // Called on Object (instance)
```

Check Your Understanding (6.3)

1. What modifier must be used if a method is intended to belong to the class rather than any specific object instance?
2. Why is the `this` keyword illegal to use inside a static method?
3. If you have a `Pet` object named `myDog`, write the syntax to call an instance method named `bark()` and the syntax to call a static method named `getSpeciesCount()` located in the `Pet` class?

Practice Problems (6.3)

1. **Design Principle:** Should the method `getHungerLevel()` (from Section 6.2, which accesses a private instance field) be declared as `static` or non-static (instance)? Explain why.
2. **Music Example:** If a `Song` class contains a `static` field storing the name of the record label that owns all songs, how would another class access this record label name? # 6.4 Constructors: Building and Initializing Objects

A **constructor** is a special type of instance method that is automatically called when an object is **instantiated** using the `new` keyword. Its primary role is to initialize the object's fields.

Constructor Properties

Constructors have unique characteristics that separate them from standard methods:

1. **Naming:** Must have the exact same name as the class (e.g., `public Car(...)`).
2. **Return Type:** They have **no return type** (not even `void`).
3. **Behavior:** They may not return any values.
4. **Access:** They are typically declared as `public`.

The Default Constructor

If you **do not** write any constructor for your class, Java automatically provides a **default constructor** (a no-argument, or no-arg, constructor). This constructor initializes instance fields to their default values:

- Numeric types (int, double) are 0 or 0.0.
- Boolean types are `false`.
- Reference types (String, custom objects) are `null`.

Example: Default Initialization

If a `Student` class has no custom constructors and fields `String name`, `double gpa`, and `boolean enrolled`, when instantiated with `Student s = new Student();`, the fields will be initialized to `name: null`, `gpa: 0.0`, and `enrolled: false`.

Writing Your Own Constructors and Constructor Overloading

If you define **any** custom constructor (e.g., one that takes arguments), **Java no longer provides the default no-arg constructor**. If you still need a no-arg option, you must explicitly write it.

Overloading allows a class to have multiple constructors (or methods) with the same name, provided they have **different parameter lists (signatures)**. This provides flexible ways to instantiate objects.

Example 6.4b: Overloaded Song Constructors (Music)

A `Song` object needs to be initialized either with full details or as a simple placeholder.

```java
public class Song {
    private String title;
    private String artist;
    private double durationMinutes;

    // 1. No-Arg Constructor (Must be manually written to retain default instantiation)
    public Song() {
        this.title = "Untitled Track";
        this.artist = "Various Artists";
        this.durationMinutes = 0.0;
    }

    // 2. Parameterized Constructor (Requires title and artist)
    public Song(String title, String artist) {
        this.title = title;
        this.artist = artist;
        this.durationMinutes = 3.5; // Default duration
    }

    // 3. Fully Parameterized Constructor (Requires all three fields)
    public Song(String title, String artist, double durationMinutes) {
        // Use 'this' to refer to the object's instance fields
        this.title = title;
        this.artist = artist;
        this.durationMinutes = durationMinutes;
    }
}
// Instantiation examples:
// Song placeholder = new Song();                             // Calls #1
// Song duet = new Song("Perfect Symphony", "Ed Sheeran");     // Calls #2
// Song classic = new Song("Bohemian Rhapsody", "Queen", 5.9); // Calls #3
```

The compiler uses the **method signature** (name + parameter types/order) to match the call with the correct overloaded constructor.

Check Your Understanding (6.4)

1. Name two key properties that distinguish a constructor from a standard method.
2. If you write a two-argument constructor for the `Pet` class, why would `Pet myPet = new Pet();` generate a compiler error?
3. What is constructor overloading, and what constraint must all overloaded constructors satisfy?

Practice Problems (6.4)

1. **Default Value (Quiz Q29):** What is the default value of a `String` field in a Java class if no constructor initializes it?
2. **Constructor Writing (Car):** Write the header for a constructor in the `Car` class that accepts only a `String` representing the car's color.
3. **Overloading Necessity:** A `Pet` class needs a constructor that accepts a `String` name, and a second constructor that accepts a `String` name and a `double` weight. Write the signatures for these two constructors. # 6.5 Using Objects in Methods (Passing and Returning)

Custom objects can be passed as arguments to methods and returned from methods, forming complex relationships between classes.

Passing Object References

When you pass an object as an argument to a method, Java follows the "pass by value" rule, but the value passed is a copy of the **memory address (reference)** to the object.

Because the receiving method gets a reference to the original object in memory, it can use the object's public mutator methods (setters) to modify the original object's internal state.

Static Helper Methods and Object Parameters

A common pattern in executable classes is using **static helper methods** that accept custom objects as parameters. Since the method is static, it doesn't belong to any specific object, but it can operate on the object passed to it.

Example 6.5: Passing a Song Object to a Playlist Manager

Assume a static method exists in a manager class to add a song to a playlist's internal structure (for simplicity, we just print the action).

```java
public class MusicManager {
    // Static method accepts a Song object reference
    public static void printSongInfo(Song s) {
        // Accessing the instance methods (getters) of the passed object 's'
        System.out.println("Processing Song: " + s.getTitle() +
                           " by " + s.getArtist());
    }

    // Static method that modifies an object (requires setter)
    public static void markAsPlayed(Song s) {
        // Assume setPlayedStatus(true) is an instance method in Song
        s.setPlayedStatus(true);
        System.out.println(s.getTitle() + " status updated to Played.");
    }

    public static void main(String[] args) {
        Song metal = new Song("Binary Code", "CPU", 4.0);

        // Pass the object reference
        printSongInfo(metal);

        // Pass the object reference for modification
        markAsPlayed(metal);
    }
}
```

Note: This static helper method can work on ***any*** *instance of the* `Song` *class.*

Returning Objects from Methods

Methods can return **references** to objects, including custom objects. The return type in the method header must match the class type being returned (e.g., `public static Car getLuxuryCar()`).

When an object reference is returned, the calling method receives the memory address and can assign it to a compatible reference variable.

Check Your Understanding (6.5)

1. When an object reference variable is passed to a method, what specific piece of information does the parameter variable hold?
2. Why is a static helper method that accepts a custom object as an argument an effective pattern in OOP?
3. If Method A calls Method B, and Method B has the header `public static String[] getArtists()`, what exactly is being returned?

Practice Problems (6.5)

1. **Object Access (Quiz Q267):** Assume you have a `User` class instance named `a`. Write the code snippet that correctly calls its instance method `getUsername()`.
2. **Method Header:** Write the method header for a static void method called `rechargeCar` that accepts one argument of type `Car`. # 6.6 Advanced Object Behavior: `toString()` and `equals()`

All classes implicitly inherit methods from the foundational `Object` class. Two crucial methods we often **override** are `toString()` and `equals()`.

Customizing `toString()`

By default, the inherited `toString()` method returns a string showing the class name and the object's memory location (hash code, e.g., `Car@xxxxx`). This occurs whenever you print an object reference using `System.out.println()`.

To make the output useful, we typically **override** `toString()` to return a descriptive string containing the values of the object's instance fields.

Example 6.6a: Overriding `toString()` for a Pet

```java
public class Pet {
    private String name;
    private String species;
    // ... constructor, fields, etc. ...

    @Override
    public String toString() {
        return "Pet Instance: Name=" + name + ", Species=" + species;
    }
}
// In main:
// Pet poodle = new Pet("Fido", "Poodle");
// System.out.println(poodle);
// Output: Pet Instance: Name=Fido, Species=Poodle
```

Customizing `equals()` for Content Comparison

The default `equals()` method compares two object references based on their **memory locations**, just like the `==` operator. It determines if two references point to the *exact same* object in memory.

If you want to define equality based on the objects' **content** (e.g., two different `Song` objects are "equal" if they share the same title and artist), you must **override** the `equals()` method.

Example 6.6b: Defining Song Equality (Music)

```java
public class Song {
    private String title;
    private String artist;
    // ...

    // Custom method to compare contents
    public boolean equals(Song otherSong) {
        // Compare titles AND artists for equality
        if (this.title.equals(otherSong.title) &&
            this.artist.equals(otherSong.artist)) {
            return true;
        } else {
            return false;
        }
    }
}
```

Note that we use the `String.equals()` method inside our custom `Song.equals()` method because strings are non-primitive objects and must be compared by content.

Check Your Understanding (6.6)

1. When is the `toString()` method implicitly called in Java?
2. If `carA` and `carB` are two different `Car` objects created independently but initialized with the same make and model, what will the default `equals()` method return, and why?
3. In `Song.equals(Song otherSong)`, why must we use `this.title.equals(otherSong.title)` instead of `this.title == otherSong.title`?

Practice Problems (6.6)

1. **`toString()` implementation (Car):** Write a custom `toString()` method for the `Car` class that prints the car's make and year in a readable format.
2. **`equals()` Logic:** You are designing a `Pet` class where two pets are considered equal if they have the same `String species`. Write the basic logic

for the custom `equals(Pet otherPet)` method. # 6.7 Packages, Scope, and Design

Packages and the `import` Statement

Classes in the Java API are organized into collections called **packages**.

- **`java.lang` Package:** Contains fundamental classes like `String`, `System`, and `Math`. This package is **automatically imported** into every Java program.
- **Other Packages:** Require an explicit **`import` statement** to tell the compiler where to find the class. For example, `Scanner` is in `java.util`.

You can import a single class (e.g., `import java.util.Scanner;`) or use a **wildcard** (`*`) to import all classes in a package (e.g., `import java.util.*;`).

Scope of Instance Fields

The **scope** of a local variable is limited to the method in which it is declared.

In contrast, **instance fields** (the private fields defined at the class level) have **class scope**—they are visible and accessible to **all** instance methods within that class. **Shadowing** occurs when a method parameter shares the same name as an instance field; the local parameter takes precedence.

Object-Oriented Design (OOD)

OOD involves identifying necessary classes in a problem and assigning them **responsibilities**.

1. **Finding the Classes:** Look for **nouns** in the problem description (e.g., `Pet`, `Car`, `Song`).
2. **Identifying Responsibilities:** A class is responsible for two things:
    - **Knowing (Fields):** The data the object stores (e.g., a `Song` knows its `artist`).
    - **Doing (Methods):** The actions the object performs (e.g., a `Song` can `playSong()`).

Check Your Understanding (6.7)

1. If a program uses the `Random` class from `java.util`, why must it include an `import` statement, but a program using the `String` class does not?
2. How does the scope of an instance field differ from the scope of a parameter variable declared in a constructor?
3. In the context of OOD, define the two types of responsibilities a class should fulfill.

Practice Problems (6.7)

1. **Import Syntax:** Write the correct Java statement to import only the `File` class from the `java.io` package.
2. **Design Analysis:** For a fitness application, identify two nouns that should become classes and list one “knowing” and one “doing” responsibility for each.