CS1010X: Programming Methodology I
Chapter 15: Object-Oriented Programming in Java

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 15: Object-Oriented Programming in Java

Learning Objectives

By the end of this chapter, students will be able to:

- **Design** composite data types using Java classes to elevate code semantics from primitive variables to robust domain models.
- **Differentiate** between static scope (class-level) and instance scope (object-level).
- **Enforce** encapsulation and the "Tell, Don't Ask" principle by applying appropriate access modifiers (`private`, `protected`, `public`) to protect invariants and decouple implementation from usage.
- **Evaluate** design scenarios to select the appropriate abstraction mechanism, that is, abstract classes for shared state/identity ("is-a") versus interfaces for pure behavioral contracts ("can-do").
- **Construct** class hierarchies that correctly utilize single inheritance for specialization and composition for flexibility, while avoiding the pitfalls of tight coupling.
- **Predict** the execution flow of polymorphic code by tracing the two-step process of static resolution (based on compile-time types) and dynamic dispatch (based on runtime type).
- **Distinguish** between method overloading and overriding by analyzing method signatures, return types, and resolution timings (compile-time vs rimto,e).

15.1 From Primitives to Composite Data Types

This chapter is largely a review of Chapter 12, Object-Oriented Programming, but in the context of an object-oriented language like Java.

In procedural programming, we often manage data as loose collections of primitive variables. For example, to represent a circle, we might pass around three separate variables: `double x`, `double y`, and `double r`.

This approach scales poorly. As the system grows, the connection between these variables relies entirely on variable naming conventions and the programmer's discipline.

Object-Oriented Programming (OOP) solves this by allowing us to define **Composite Data Types**. A composite data type aggregates primitive types (or other reference types) into a single logical unit that models a concept in the problem domain.

In Java, the **class** is the mechanism for defining a composite data type.

A class consists of:

- State (Fields); and
- Behaviour (Methods)

15.2 Java Class Syntax

A Java class is a blueprint that defines the structure and behavior of objects. The syntax is strictly structured.

```java
// 1. Class Declaration
// 'public' means accessible everywhere.
public class BankAccount {

    // 2. Fields (State)
    // Variables declared inside the class but outside methods.
    // 'private' hides them from the outside world (Encapsulation).
    private double balance;
    private String ownerName;

    // 3. Constructor (Initialization)
    // A special method with:
    //    - The same name as the class.
    //    - NO return type (not even void).
    // Runs automatically when 'new BankAccount(...)' is called.
    public BankAccount(String ownerName, double initialBalance) {
        this.ownerName = ownerName;
        this.balance = initialBalance;
    }

    // 4. Instance Methods (Behavior)
    // Operations that act on the object's state.
    public void deposit(double amount) {
        if (amount > 0) {
            this.balance += amount;
        }
    }

    public double getBalance() {
        return this.balance;
    }
}
```

15.2.1 Key Components

- **Declaration:** `public class ClassName { ... }`. By convention, class names are `UpperCamelCase`.
- **Fields:** Store the object's data. Best practice is to mark them `private` to enforce encapsulation.
- **Constructor:** Initializes the object. If you do not define one, Java provides a default no-argument constructor.
- **Methods:** Define what the object can do. They can return values or be `void`.
- **`this`:** A keyword referring to the current instance. It resolves ambiguity between fields (`this.balance`) and parameters (`balance`).

15.3 Static vs. Instance Scope

In Java, members (fields and methods) can be associated either with individual objects (instances) or with the class itself.

15.3.1 Instance Members

- **Instance Fields:** Every time `new Class()` is called, a new object is created. Each object has its own copy of instance fields. Changing an object's field does not affect others.
- **Instance Methods:** Can only be invoked on an object reference (`obj.method()`). They can access both the specific object's state (`this`) and static members.
- **`this` Reference:** Inside an instance method, Java implicitly passes a reference to the object on which the method was called. This reference is accessible via `this`.

15.3.2 Static Members (Denoted with `static` modifier)

- **Static Fields:** Belong to the class itself. There is exactly **one copy** in memory (in the metaspace), regardless of how many objects exist (or even if none exist). They represent shared state or constants (e.g. `Math.PI`).
- **Static Methods:** Invoked on the class (`Class.method()`). They generally act as utility functions (e.g. `Math.sqrt()`).
- **No `this`:** Since static methods are not called on specific objects, `this` is undefined and illegal in static contexts. So, **static methods cannot access instance fields.**

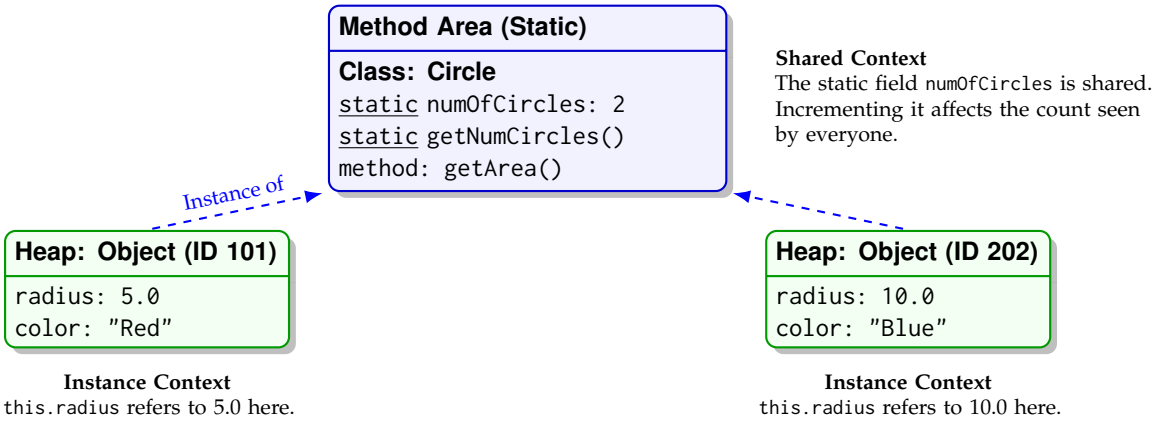

Figure 15.1: Static members exist once per class; Instance members exist once per object.

Use `static` when the logic or data does not depend on a specific instance.

```java
public class IDGenerator {
    // Shared counter across all instances
    private static int nextID = 1;
    // Logic is universal, doesn't need an object state
    public static int generate() {
        return nextID++; // Call using IDGenerator.generate();
    }
}
```

15.4 Abstraction (Implementation Hiding)

There are two main ways to achieve high-level abstraction in Java: **Abstract Classes** and **Interfaces**.

These two ways provide users the ability to define a **contract**, defining the behaviour required from the classes, but leaving the implementation up to the individual classes.

We shall look at their uses and differences before talking about abstraction in more detail.

15.4.1 Abstract Classes

An abstract class is a class that cannot be instantiated on its own. It is used to define a template for subclasses, mixing both defined behavior (concrete methods) and required behavior (abstract methods).

- Declared using the `abstract` keyword.
- Can contain **state** (instance variables/fields).
- Can contain **constructors** (to initialize state for subclasses).
- Subclasses use the `extends` keyword (Single Inheritance).

```java
public abstract class Shape {
    // 1. State: Allowed (unlike interfaces)
    protected String color;

    // 2. Constructor: Used by subclasses to init state
    public Shape(String color) {
        this.color = color;
    }

    // 3. Abstract Method: No body, forces subclass to implement
    public abstract double getArea();

    // 4. Concrete Method: Has body, shared behavior
    public void display() {
        System.out.println("This is a " + color + " shape.");
    }
}
```

15.4.2 Interfaces

An interface is a pure contract. It defines a set of behaviors that a class must implement. It represents a "capabilities" relationship (e.g. `Drawable`, `Serializable`).

- Declared using the `interface` keyword.
- **No State:** Variables are implicitly `public static final` (constants).
- **No Constructors:** Interfaces cannot hold state, so they need no initialization.
- Classes use the `implements` keyword (Multiple Implementation allowed).

```
public interface Drawable {
    // 1. Constants: Implicitly public static final
    int MAX_OPACITY = 100;

    // 2. Abstract Method: Implicitly public abstract
    void draw();

    // 3. Default Method (Java 8+): Optional behavior
    default void reset() {
        System.out.println("Resetting to default...");
    }
}

// Implementation
public class Circle extends Shape implements Drawable {
    public Circle(String color) { super(color); }
    @Override public double getArea() { return Math.PI; } // From
        Shape
    @Override public void draw() { /* logic */ }          // From
        Drawable
}
```

Feature	Abstract Class	Interface
Relationship	**"Is-A"** (Identity)	**"Can-Do"** (Capability)
State (Fields)	Can have instance variables (mutable state).	Can ONLY have constants (`static final`).
Inheritance	Single inheritance (`extends`).	Multiple implementation (`implements`).
Constructors	**Yes**, to initialize state.	**No**, cannot be instantiated.
Methods	Can have any visibility (`private`, `protected`).	Methods are `public` by default.
Best Use	When subclasses share core code and state (e.g. `Animal` → `Dog`).	To decouple unrelated classes via a common protocol (e.g. `Comparable`).

Table 15.1: Comparison of Abstract Classes and Interfaces

15.4.3 Abstraction

Abstraction focuses on hiding the implementation details and exposing only the functionality to the user. The user knows *"what it does"*, but not *"how it does it"*.

In general, even simple functions are abstractions because they provide an interface (signature) while hiding the logic body. As we have seen earlier, abstract classes and interfaces also provide a useful tool for high-level abstraction in Java.

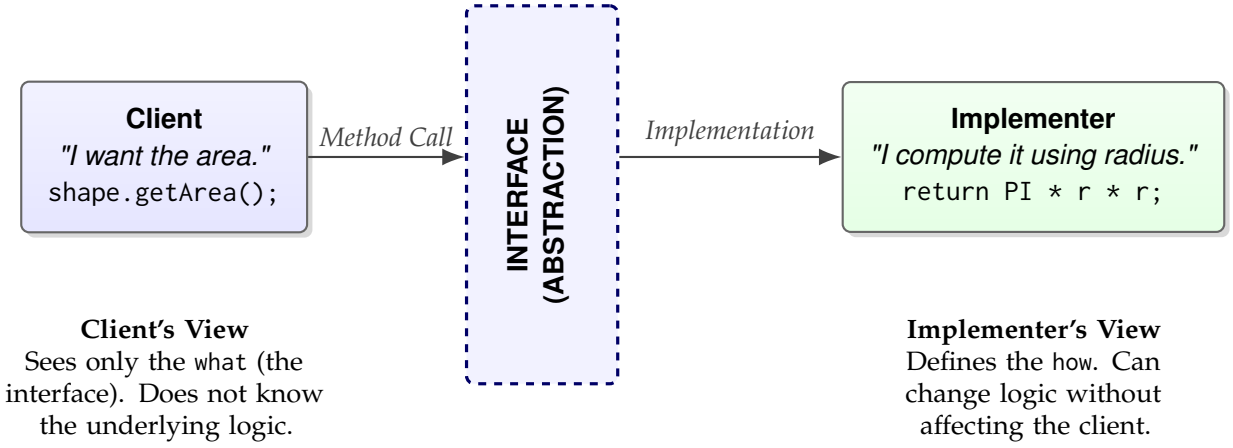

Figure 15.2: Abstraction separates the definition of behavior from its implementation.

Consider a system that needs to process payments. The core logic should not care whether the user pays via credit card or PayPal.

```
// 1. The Abstraction (The "What")
// Defines the contract. Users know payment happens,
// but don't know (or care) how the networking works.
interface PaymentProcessor {
    void pay(double amount);
}

// 2. Concrete Implementation A (The "How")
class CreditCardProcessor implements PaymentProcessor {
    private String cardNumber;

    public CreditCardProcessor(String cardNumber) {
        this.cardNumber = cardNumber;
    }

    @Override
    public void pay(double amount) {
        // Complex logic: Connect to Visa/Mastercard...
        System.out.println("Paid $" + amount + " via Card " +
            cardNumber);
    }
}

```

```java
// 3. Concrete Implementation B (The "How")
class PayPalProcessor implements PaymentProcessor {
    private String email;

    public PayPalProcessor(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        // Complex logic: Login to digital wallet...
        System.out.println("Paid $" + amount + " (PayPal) " + email);
    }
}

// 4. The Client Code
// Note: This method works with ANY PaymentProcessor.
// It is loosely coupled to the specific implementation.
public static void checkout(PaymentProcessor p, double total) {
    p.pay(total); // "Tell, don't ask"
}
```

Notice that in the above, the client does not need to know the specific details about *how* the payment processor implements the `pay` function. It just requires that payment processsors provide this functionality.

15.5 Encapsulation

Encapsulation bundles data (state) and methods (behavior) into a single unit and restricts direct access to some of an object's components.

The first major reason for encapsulation is **information hiding**. By making fields `private`, the Implementer ensures that the object never enters an invalid state (an invariant violation).

Consider a `Circle` class where the radius is `public`. There is nothing stopping a client from setting a negative radius, which is geometrically impossible.

```
public class BadCircle {  // BAD: No Access Control
    public double radius; // Accessible to everyone
}

// Client Code
BadCircle c = new BadCircle();

// DANGER: The client can break the invariant (radius >= 0).
c.radius = -100.0;
```

By marking the field `private`, we force the client to go through the constructor or methods, where we can add validation logic.

```
// GOOD: Proper Encapsulation
public class Circle {

    // 1. Hide the internal state
    private double radius;

    // 2. Enforce rules in the Constructor
    public Circle(double radius) {
        if (radius < 0) {
            throw new IllegalArgumentException("Radius cannot be
                negative");
        }
        this.radius = radius;
    }
}
```

15.5.1 Access Control Modifiers

Java provides four levels of access control to determine where members (fields and methods) can be used. This is the primary tool for enforcing the abstraction barrier.

Best Practice: Follow the "Principle of Least Privilege." Always start with `private`. Open access to `protected` or `public` only when absolutely necessary for the design.

Modifier	Same Class	Same Package	Subclass	World
`public`	**Yes**	**Yes**	**Yes**	**Yes**
`protected`	**Yes**	**Yes**	**Yes**	No
(package-private)	**Yes**	**Yes**	No	No
`private`	**Yes**	No	No	No

Table 15.2: Java Access Modifiers. Note: "Subclass" access refers to inheritance even across packages.

15.5.2 The "Tell, Don't Ask" Principle

The second major reason for encapsulation is **decoupling**. We want to prevent the client from depending on *how* the data is stored. Asking for fields and performing logic outside the class causes code to break if internal representation is changed.

```
// BAD: Client Logic depends on 'radius' field existing
Circle c = new Circle(5.0);
// If 'radius' is renamed or removed, this code breaks.
double area = 3.14159 * c.radius * c.radius;
```

Instead, we move the logic *into* the class. The client *tells* the object to compute the area.

```
// GOOD: Client relies on the Behavior (Interface)
Circle c = new Circle(5.0);
double area = c.getArea();
```

If the implementer decides to change the internal representation from **radius** to **diameter**, the "Ask" client breaks, but the "Tell" client survives.

```
public class Circle { // REFACTORED CLASS (Version 2)
    // Internal representation CHANGED, radius no longer exists!
    private double diameter;
    public Circle(double radius) {
        this.diameter = radius * 2;
    }
    // Public method signature stays the same.
    // The client code (c.getArea()) does NOT need to change.
    public double getArea() {
        double r = this.diameter / 2.0;
        return Math.PI * r * r;
    }
}
```

Avoid Anemic Models

Avoid exposing every field via trivial getters and setters (e.g. `get_x()`, `set_x()`). This breaks encapsulation and turns your class into a dumb data structure.

15.6 Inheritance and Composition

Inheritance allows a class to derive attributes and methods from another class. It is appropriate only when there is a strict **is-a** relationship (e.g. a `Ship` *is a* `MobileObject`).

Subclasses inherit all methods and properties by default, fulfilling the Don't Repeat Yourself (DRY) principle, but also can **override** them, providing extensibility.

How do we decide when to use inheritance?

1. **Identify Classes:** Look for the distinct "nouns" in your system (e.g. Ships, Torpedoes).
2. **Identify Commonality:** Do these objects share state (position, velocity) or behavior (move, update)?
3. **Refactor:** Extract this commonality into a **Base Class** (Superclass).

15.6.1 Inheritance in Java

Inheritance represents an **"is-a"** relationship. For example, a `Dog` *is an* `Animal`.

- **Syntax:** Use the `extends` keyword.
- **Single Inheritance:** Java classes can extend only **one** parent class.
- **Default Behavior:** By default, a subclass inherits all `public` and `protected` members (fields and methods) of the parent.

Constructors are **not** inherited. Since a subclass relies on the state defined in the parent, the parent must be initialized *before* the child. Java uses the `super` keyword to manage this.

1. **`super()`:** The subclass constructor **must** call the parent's constructor.
   - If you do not write a call to `super()`, Java implicitly inserts `super()` (no arguments) as the first line.
   - If you explicitly call `super(args)`, it **must be the very first statement** in the constructor.
2. **Parent Members (`super.`):** You can use `super.methodName()` to access a method in the parent class, even if it has been overridden in the child.

15.6.2 Multiple Interfaces vs. Single Classes

While Java restricts you to extending a single class (to define **identity**), it allows you to implement multiple interfaces (to define **capabilities**).

Since interfaces traditionally contain no state (fields) and no implementation logic, implementing multiple interfaces does not suffer from the "Diamond Problem" in the same way classes do.

- **Identity (extends):** An object can only "be" one thing fundamentally. A `Dog` is an `Animal`. It cannot also be a `Planet`.
- **Capabilities (implements):** An object can have many skills. A `Dog` can be `Trainable`, `Petable`, and `Serializable`.

Java enforces a strict ordering in the class declaration header. You must define the **parent class** first, followed by the **interfaces**. That is, `extends` comes before `implements`.

```java
// 1. Define the Parent (Class)
abstract class Animal { ... }
// 2. Define Capabilities (Interfaces)
interface Trainable { void sit(); }
interface Petable { void pet(); }
// 3. The Child Class
// CORRECT: 'extends' first, then 'implements'
public class Dog extends Animal implements Trainable, Petable {
    // Must implement methods from ALL interfaces
    public void sit() { ... }
    public void pet() { ... }
}
// INCORRECT: Compilation Error!
// public class Dog implements Trainable extends Animal { ... }
```

Extra: Diamond Problem with Default Methods?

Since Java 8, interfaces can have **default methods** (methods with bodies). What happens if two interfaces implement the same method?

- Java detects the conflict at compile time.
- It forces you to **override** the method in the child class and explicitly choose which version to use (e.g. `InterfaceA.super.method()`). This prevents the ambiguity found in C++ multiple inheritance.

Now, an example of simple inheritance is:

```java
class Animal {
    protected String name;
    public Animal(String name) { // Parent Constructor
        this.name = name;
        System.out.println("Animal initialized");
    }
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}
class Dog extends Animal {
    public Dog(String name) { // Child Constructor
        // 1. MUST call parent constructor first
        super(name);
        System.out.println("Dog initialized");
    }
    @Override
    public void makeSound() {
        // 2. Can call the parent implementation
        super.makeSound();
        System.out.println("Bark!");
    }
}
```

Notice the use of the decorator `@Override`. This signals to the compiler that this method is meant to override a parent function. This prevents us from attempting to override non-existent methods (which will be more useful when working with generic classes in CS2030/S). As such, it is good practice to make use of the decorator.

The Diamond Problem (Why Single Inheritance?)

Java forbids multiple class inheritance to avoid the "Diamond Problem." Imagine Class B and Class C both inherit from Class A and override a method `run()`. If Class D inherits from **both** B and C, which version of `run()` does it use?

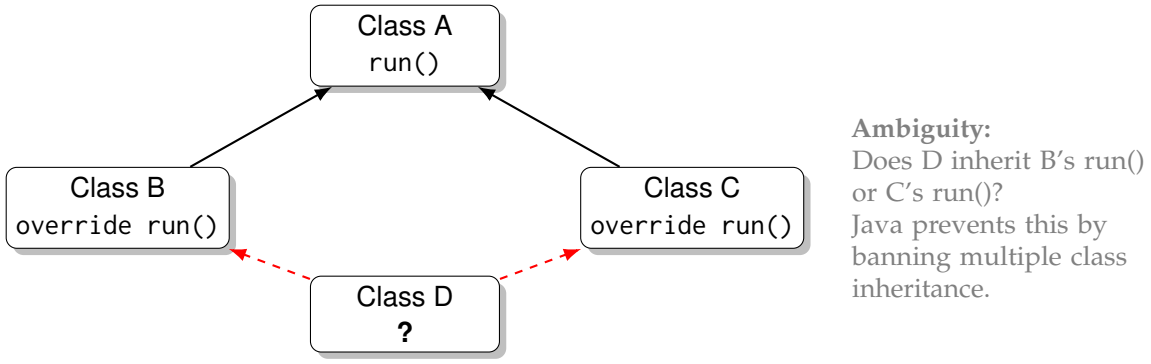

Figure 15.3: The Diamond Problem: Why Java allows only single class inheritance.

15.6.3 Inheritance vs. Composition

Once again, there is a key difference between inheritance and composition:

- **Inheritance ("Is-A"):** Use when the child class is a specialized version of the parent. It is rigid; the relationship is defined at compile time.
- **Composition ("Has-A"):** Use when a class is made up of other components. The class holds references to other objects (fields) to delegate work. It is flexible; components can often be swapped at runtime.

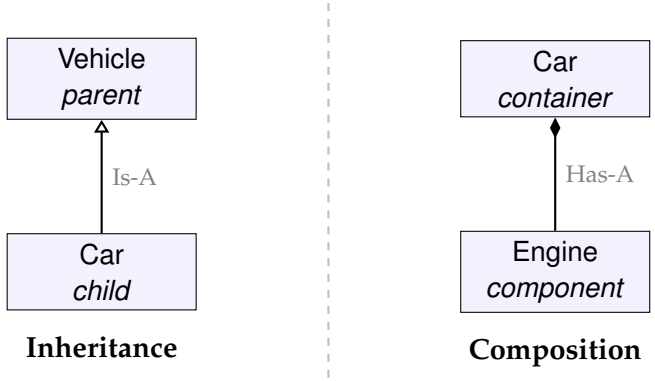

Figure 15.4: Visualizing Relationships: Inheritance is for specialization; Composition is for assembly.

For example, consider this code that describes a car and its components.

```java
// --- APPROACH 1: INHERITANCE (Is-A) ---
// Valid: A Car IS-A Vehicle.
// It inherits speed and move() logic.
public class Vehicle {
    protected int speed;
    public void move() { /* logic */ }
}

public class Car extends Vehicle {
    // Car inherits 'speed' and 'move()' automatically.
}

// --- APPROACH 2: COMPOSITION (Has-A) ---
// Valid: A Car HAS-AN Engine.
// It would be wrong to say "Car extends Engine".
class Engine {
    public void start() { System.out.println("Vroom"); }
}

public class Car {
    // Composition: The Car 'has' an Engine
    private Engine engine;

    public Car() {
        this.engine = new Engine();
    }

    public void startCar() {
        // Delegation: The Car asks the Engine to do the work
        this.engine.start();
    }
}
```

Feature	Inheritance	Composition
Relationship	**Is-A** (Dog is an Animal)	**Has-A** (Car has an Engine)
Coupling	**High.** Child breaks if Parent changes.	**Low.** Loose coupling via interfaces.
Flexibility	Static (fixed at compile time).	Dynamic (can swap components at runtime).
Code Reuse	Automatic (inherits code).	Manual (must write delegation methods).

Table 15.3: Comparison of Inheritance and Composition

15.7 The Object Class

In Java, the `java.lang.Object` class is the root of the class hierarchy. Every class has `Object` as a superclass. If a class does not explicitly extend another class, it implicitly extends `Object`.

This means that a variable of type `Object` can reference an instance of *any* class in Java.

Since all objects inherit from `Object`, they all possess the following critical methods. It is common practice to **override** these methods to provide domain-specific behavior.

- `String toString()`: Returns a string representation of the object.
  - **Default:** `ClassName@HexAddress` (e.g. `Dog@1b6d3586`).
  - **Best Practice:** Override to return human-readable state (e.g. `"Dog[name=Rex]"`).
- `boolean equals(Object obj)`: Checks if some other object is "equal to" this one.
  - **Default:** Checks **Reference Equality** (acts like ==). Returns true only if both variables point to the *same memory address*.
  - **Best Practice:** Override to check **Logical Equality** (e.g. do they have the same ID or values?).
- `int hashCode()`: Returns a hash code value for the object.
  - **Contract:** If two objects are equal according to `equals(Object)`, they **MUST** have the same `hashCode()`.
  - Used by hash-based collections like `HashMap` and `HashSet`.

The code belows show how we override `toString()` from Object.

```
class Point {
    private int x, y;

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

    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
}

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Point p2 = new Point(1, 2);

        // toString() is called automatically
        System.out.println(p1); // Prints "(1, 2)" instead of Point@
            ...
    }
}
```

15.8 Polymorphism

Polymorphism allows us to write succinct code that is future-proof. By writing code against a common superclass or interface, we can swap in new implementations without changing the client logic.

In Java, polymorphism relies on **dynamic dispatch**, which is the mechanism where the specific implementation of a method is decided at runtime based on the actual object type.

Polymorphism

CS1010X and CS2030/S differ in their definition of polymorphism. In CS1010X, polymorphism includes both:

- **Method overloading** (sometimes known as *static polymorphism*).
- **Subtype polymorphism**, enabled by method overriding.

15.8.1 Method Signature vs. Descriptor

Before understanding dispatch, we must define what identifies a method.

- **Method Signature:** The method name + the number, type, and order of parameters. *Note:* Parameter names are irrelevant. `add(int x, int y)` is the same signature as `add(int a, int b)`.
- **Method Descriptor:** The Method Signature + the **Return Type**.

15.8.2 Overloading vs. Overriding

Method overloading and overriding fundamentally differ. Method oveloading refers to have the same method name, and different method **signatures**, while method overriding refers to an instance method with the same **method descriptor** in subclasses.

Feature	Method Overloading	Method Overriding
Definition	Same method name, **different** signature.	Subclass defining an instance method with the **same** descriptor.
Resolution	Resolved at **compile time** (static).	Resolved at **runtime** (dynamic).
Return Type	Can change freely.	Must be the same or **covariant** (a subtype of the original).
Restrictions	None.	`private`, `static`, and `final` methods **cannot** be overridden.

Table 15.4: Comparison of Overloading and Overriding

15.8.3 Overloading Resolution

When multiple methods share the same name but have different parameters, the compiler must decide which one to call. This process occurs at **compile time**.

The compiler searches for the "most specific method" in three strictly ordered phases. If a candidate is found in Phase 1, it stops there. If not, it proceeds to Phase 2, and throws an error otherwise.

Phase 1: Exact Match

The compiler looks for a method where the parameter types **exactly match** the argument types.

- **Example:** Calling `f(int)` with an `int` argument.
- *Note:* Even if the method is inherited from a superclass, an exact type match is preferred over a closer class hierarchy match with different types.

Phase 2: Widening

If no exact match is found, the compiler tries to match by **widening** the argument type.

- **Primitives:** Promotes smaller types to larger types (e.g. `int` → `long` → `float` → `double`).
- **References:** Moves up the inheritance chain (e.g. `ShihTzu` → `Dog` → `Animal`).

```
class Animal {}
class Dog extends Animal {}
class ShihTzu extends Dog {}

class Handler {
    // General overload
    void treat(Animal a) { System.out.println("Treating Animal"); }
    // Specific overload
    void treat(Dog d)    { System.out.println("Treating Dog"); }
}

Handler h = new Handler();
ShihTzu puppy = new ShihTzu();

// Resolution Steps:
// 1. Exact Match treat(ShihTzu)? No.
// 2. Candidates via Widening:
//      - treat(Dog)    : Valid (ShihTzu is a Dog)
//      - treat(Animal) : Valid (ShihTzu is an Animal)
// 3. Tie-Breaker ("Most Specific"):
//    Since Dog is a subclass of Animal, treat(Dog) is more specific.
h.treat(puppy); // Prints "Treating Dog"
```

15.8.4 The Two-Step Method Invocation Process

Understanding how Java selects a method is crucial. It happens in two distinct phases.

Step 1: Compile Time (Static Resolution)

The compiler looks at the **Compile-Time Type (CTT)** of the variable (the type on the left side of the equals sign).

1. It checks the CTT class for methods matching the name.
2. It applies Overloading Resolution to pick the most specific method signature available in the CTT hierarchy.
3. **Result:** The **method descriptor** is locked in. If the CTT does not have the method, compilation fails.

Between compile time and runtime, the method descriptor **CANNOT** change.

Step 2: Runtime (Dynamic Dispatch)

The Java Virtual Machine (JVM) looks at the **Runtime Type (RTT)** of the actual object (the type after `new`).

1. It starts searching from the RTT class.
2. It looks for a method that matches the **exact method descriptor** chosen in Step 1.
3. If found, it executes it. If not, it moves up the parent hierarchy until it finds the implementation.

```
1  class Animal {
2      void speak() { System.out.println("Silence"); }
3  }
4  class Dog extends Animal {
5      @Override
6      void speak() { System.out.println("Bark"); }
7      void fetch() { System.out.println("Ball!"); }
8  }
9  public class Main {
10     public static void main(String[] args) {
11         // CTT = Animal, RTT = Dog
12         Animal a = new Dog();
13
14         // Step 1 (Compile): Animal has speak(). Descriptor locked:
               void speak()
15         // Step 2 (Runtime): RTT is Dog. Does Dog have void speak()?
               Yes.
16         a.speak(); // Prints "Bark"
17
18         // Step 1 (Compile): Animal does NOT have fetch().
19         // COMPILATION ERROR! Even though RTT (Dog) has it.
20         // a.fetch();
21     }
22 }
```

15.8.5 The `instanceof` Operator

Sometimes we need to check the runtime type of an object explicitly before casting it to access specific methods.

- **Syntax:** `obj instanceof TypeName`
- **Logic:** Returns `true` if the RTT of `obj` is a subtype of (or the same as) `TypeName` ($RTT <: TypeName$). This is very similar to our `isinstance` operator in Python.

```
Object obj = "Hello";

if (obj instanceof String) {
    // Safe to downcast
    String s = (String) obj;
    System.out.println(s.length());
}
```

Looking Forward: CS2030S

This chapter provides a foundational overview of Object-Oriented Programming in Java. In the follow-up module, **CS2030S (Programming Methodology II)**, you will dive much deeper into advanced OOP and Functional Programming concepts in Java, including:

- **Generics:** Writing flexible, reusable code that works with any data type.
- **Functional Interfaces & Streams:** Modern functional programming patterns in Java.
- **Asynchronous Programming:** Managing parallel tasks and non-blocking code.

If you need more detailed notes about OOP in Java, please refer to the official CS2030S materials at:

https://nus-cs2030s.github.io/2526-s1/index.html