Module 10 — OOP Inheritance

ITP 120 — Java Programming I

Northern Virginia Community College | Randy Michak

🎯 Learning Objectives

- Explain what inheritance is and describe the superclass/subclass relationship
- Use the `extends` keyword to create a subclass
- Call a superclass constructor using `super()`
- Override a method and explain when to use `@Override`
- Describe how every Java class inherits from `Object`
- Write polymorphic code using superclass references and dynamic binding
- Define abstract classes and abstract methods
- Implement an interface and distinguish it from an abstract class
- Explain the `protected` access modifier
- Build a multi-class hierarchy (Shape, Circle, Rectangle) from scratch

Prerequisites: Modules 1–9 — comfortable with variables, loops, arrays, and the OOP fundamentals from Module 8 (classes, objects, encapsulation).

1. What Is Inheritance?

Inheritance lets one class acquire the fields and methods of another class. Instead of copying code into every new class, you put shared behavior in one place — the **superclass** — and let the **subclasses** build on top of it.

🧠 **Think About It**

Picture a **Vehicle**. Every vehicle has a make, a model, and the ability to start. A **Car** is a Vehicle — plus it has a trunk. A **Truck** is also a Vehicle — plus it has a payload capacity. They both get Vehicle's features for free.

That phrase — "is-a" — is your test for inheritance. If you can say "a Car **is a** Vehicle," inheritance fits. If you have to say "a Car **has a** Vehicle," that's composition, not inheritance.

📖 **Key Terms**

Term	Meaning
Superclass	The parent class — defines shared fields and methods
Subclass	The child class — inherits from the superclass and can add its own members
Inheritance	The mechanism that lets a subclass reuse superclass code
"is-a" relationship	The test for inheritance: "A Dog *is an* Animal" checks out

2. The `extends` Keyword

You declare inheritance with the `extends` keyword in the class header.

```java
// Superclass
public class Vehicle {
    protected String make;
    protected String model;
    protected int year;

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

    public void start() {
        System.out.println(year + " " + make + " " + model + " is starting.");
    }

    public String getInfo() {
        return year + " " + make + " " + model;
    }
}

// Subclass — Car inherits from Vehicle
public class Car extends Vehicle {
    private double trunkVolume;

    public Car(String make, String model, int year, double trunkVolume) {
        super(make, model, year);   // call superclass constructor
        this.trunkVolume = trunkVolume;
    }

    public double getTrunkVolume() {
        return trunkVolume;
    }
}
```

What does `Car` inherit from `Vehicle`? The `make`, `model`, and `year` fields (because they're `protected`), and the `start()` and `getInfo()` methods.

> ⚠️ **Constructors Are NOT Inherited**
>
> This catches everyone the first time. Constructors belong to the class that declares them — they do not pass down to subclasses. Every subclass needs its own constructor.

> 💡 **What Gets Inherited?**
>
> - **public** and **protected** fields and methods — inherited
> - **private** fields and methods — not directly accessible in the subclass
> - **Constructors** — never inherited

3. Calling the Superclass Constructor

Since constructors aren't inherited, your subclass constructor must explicitly call the superclass constructor using `super()`. This initializes the inherited fields.

```
public class Truck extends Vehicle {
    private double payloadTons;

    public Truck(String make, String model, int year, double payloadTons) {
        super(make, model, year);   // MUST be the very first line
        this.payloadTons = payloadTons;
    }

    public double getPayloadTons() {
        return payloadTons;
    }
}
```

> ⚠️ **`super()` Must Be First**
>
> If you call `super()`, it must be the **first statement** in the constructor body. The compiler rejects it otherwise — no exceptions.

If you leave out `super()`, Java automatically inserts a call to the superclass's **no-argument constructor**. If the superclass doesn't have one, you'll get a compile error. This trips up a lot of beginners.

```
// Chaining three levels deep: ElectricCar -> Car -> Vehicle
public class ElectricCar extends Car {
    private int batteryRange;

    public ElectricCar(String make, String model, int year,
                       double trunkVolume, int batteryRange) {
        super(make, model, year, trunkVolume);  // calls Car constructor
        this.batteryRange = batteryRange;
    }
}
```

🧠 **Constructor Chaining**

Before `ElectricCar` can set up its own field, `Car` needs to be set up. Before `Car` finishes, `Vehicle` needs to be set up. The chain always runs upward through the hierarchy before coming back down to finish initialization.

4. Overriding Methods

A subclass can replace a superclass method with its own version. Same name, same parameter list, same return type — different body. This is **method overriding**.

```
public class ElectricCar extends Car {
    private int batteryRange;

    public ElectricCar(String make, String model, int year,
                       double trunkVolume, int batteryRange) {
        super(make, model, year, trunkVolume);
        this.batteryRange = batteryRange;
    }

    @Override
    public void start() {
        System.out.println(getInfo() + " is powering up silently.");
    }

    @Override
    public String getInfo() {
        return super.getInfo() + " [Electric, " + batteryRange + " mi range]";
    }
}
```

```
ElectricCar tesla = new ElectricCar("Tesla", "Model 3", 2024, 2.8, 358);
tesla.start();
// Output: 2024 Tesla Model 3 [Electric, 358 mi range] is powering up silently.

System.out.println(tesla.getInfo());
// Output: 2024 Tesla Model 3 [Electric, 358 mi range]
```

Notice `super.getInfo()` in the override — that calls the `Vehicle` version and then appends the electric-car details on top of it. You're extending behavior, not replacing it from scratch.

> **💡 Always Use @Override**
>
> Put `@Override` on the line above every method you intend to override. If you typo the method name, you'd silently create a brand new method instead of an override. `@Override` makes the compiler catch that mistake for you.

> 💡 **Override vs Overload — Don't Confuse These**
>
> **Overriding:** Same signature, subclass replaces superclass behavior. Resolved at *runtime*.
> **Overloading:** Same name, different parameter list, same class. Resolved at *compile time*.

5. The `Object` Class

Every class in Java — whether you write it or it comes from the standard library — inherits from `java.lang.Object`. You never have to write `extends Object`; Java adds it automatically behind the scenes.

Method	What It Does	Default Behavior (Before Override)
`toString()`	String representation of the object	ClassName@hexHash — useless in practice
`equals(Object o)`	Tests equality	Compares memory addresses (same as ==)
`hashCode()`	Integer hash of the object	Based on memory address
`getClass()`	Runtime class of the object	Returns a `Class` object — final, not overrideable

Override `toString()`

```
public class Vehicle {
    protected String make, model;
    protected int year;
    // constructor omitted for brevity

    @Override
    public String toString() {
        return year + " " + make + " " + model;
    }
}

Vehicle v = new Vehicle("Honda", "Civic", 2023);
System.out.println(v);          // calls toString() automatically
// Output: 2023 Honda Civic

// Without the override you'd see: Vehicle@7852e922
```

Override `equals()`

```
// Default behavior — compares addresses, not content
Vehicle v1 = new Vehicle("Honda", "Civic", 2023);
Vehicle v2 = new Vehicle("Honda", "Civic", 2023);
System.out.println(v1 == v2);        // false — different objects
System.out.println(v1.equals(v2));   // false — default checks address

// Override to compare field values instead
@Override
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (!(obj instanceof Vehicle)) return false;
    Vehicle other = (Vehicle) obj;
    return this.year == other.year
        && this.make.equals(other.make)
        && this.model.equals(other.model);
}

System.out.println(v1.equals(v2));   // true — same content
```

6. Polymorphism

Polymorphism means "many forms." In Java, a superclass reference variable can hold a subclass object, and the correct method version is automatically called at runtime based on what the object actually is — not what the variable's type is. This is called **dynamic binding**.

```
Vehicle v = new Car("Toyota", "Camry", 2022, 13.5);
v.start();   // Car's start() is called if Car overrides it
```

Why This Matters — One Loop, Many Behaviors

```
Vehicle[] fleet = new Vehicle[3];
fleet[0] = new Vehicle("Ford",  "F-150",  2021);
fleet[1] = new Car("Toyota",    "Camry",  2022, 13.5);
fleet[2] = new ElectricCar("Tesla", "Model 3", 2024, 2.8, 358);

for (Vehicle v : fleet) {
    v.start();
}
```

```
2021 Ford F-150 is starting.
2022 Toyota Camry is starting.
2024 Tesla Model 3 [Electric, 358 mi range] is powering up silently.
```

One loop. No type checks. Each object handles its own behavior. That's the payoff of polymorphism — write code against the superclass type, and behavior adjusts automatically to the real object.

⚠️ Accessing Subclass-Only Members

A `Vehicle` reference only knows about `Vehicle` methods. To call something Car-specific, you must cast first:

```
Vehicle v = new Car("Toyota", "Camry", 2022, 13.5);
// v.getTrunkVolume();  // compile error

if (v instanceof Car) {
    Car c = (Car) v;
    System.out.println(c.getTrunkVolume());   // 13.5
}
```

Always check with `instanceof` before casting — otherwise you risk a `ClassCastException` at runtime.

7. Abstract Classes

Sometimes a superclass is a template only — it would make no sense to create an instance of it directly. Java lets you enforce this with the `abstract` keyword.

```
public abstract class Shape {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    // Abstract methods — no body, subclasses MUST provide one
    public abstract double area();
    public abstract double perimeter();

    // Concrete method — subclasses get this for free
    public void displayInfo() {
        System.out.printf("Shape: %s | Color: %s | Area: %.2f%n",
            getClass().getSimpleName(), color, area());
    }
}
```

⚠️ You Cannot Instantiate an Abstract Class

```
Shape s = new Shape("red");   // compile error
```

That's intentional. `Shape` is incomplete — it has no formula for area or perimeter. Only a concrete subclass like `Circle` fills in that blank.

```
public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }

    @Override
    public double perimeter() {
        return 2 * Math.PI * radius;
    }
}
```

Abstract Class	Concrete Class
Declared with `abstract`	No `abstract` keyword
Can have methods with no body	All methods must have a body
Cannot use `new` directly	Can be instantiated with `new`
Used as a base/template type	Represents an actual usable object

8. Interfaces

An **interface** is a pure contract. It defines what a class can do without saying anything about how to do it. A class "signs the contract" with the `implements` keyword.

```
public interface Drawable {
    void draw();              // implicitly public and abstract
    void resize(double factor);
}

public interface Printable {
    void print();
}
```

```
public class Circle extends Shape implements Drawable, Printable {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override public double area()      { return Math.PI * radius * radius; }
    @Override public double perimeter() { return 2 * Math.PI * radius; }

    @Override
    public void draw() {
        System.out.println("Drawing circle, radius = " + radius);
    }

    @Override
    public void resize(double factor) { radius *= factor; }

    @Override
    public void print() {
        System.out.printf("Circle | color=%s | radius=%.2f%n", color, radius);
    }
}
```

💡 Multiple Interfaces Are Allowed

A class can only `extend` one superclass — Java does not support multiple class inheritance. But a class can `implement` as many interfaces as it needs. That's how Java sidesteps the "diamond problem."

Feature	Interface	Abstract Class
Keyword used	`implements`	`extends`
Multiple allowed?	Yes — implement as many as you like	No — one superclass only
Fields	Only `public static final` constants	Any field type
Method bodies	No (pre-Java 8); `default` methods in Java 8+	Yes — mix abstract and concrete
Constructor	None	Can have constructors
Use it when	Defining a capability ("can draw")	Defining a base type ("is a shape")

9. The `protected` Access Modifier

`protected` sits between `private` and `public`. It gives subclasses direct access to a field or method while still hiding it from unrelated code outside the package.

Modifier	Same Class	Same Package	Subclass	Everywhere Else
`private`	Yes	No	No	No
(default)	Yes	Yes	No	No
`protected`	Yes	Yes	Yes	No
`public`	Yes	Yes	Yes	Yes

```
public class Vehicle {
    protected String make;    // Car can read/write this directly
    private   String vin;     // Car cannot touch this
}

public class Car extends Vehicle {
    public void printInfo() {
        System.out.println(make);   // fine — protected
        // System.out.println(vin);  // compile error — private
    }
}
```

> 💡 **When to Use protected**
>
> Use `protected` for fields and methods that subclasses genuinely need to access directly. Avoid making everything `protected` just for convenience — it weakens encapsulation.

10. Practical Example — The Shape Hierarchy

Let's pull everything together. We'll build a complete hierarchy: an abstract `Shape` superclass, two concrete subclasses (`Circle` and `Rectangle`), and a driver that demonstrates polymorphism.

The Abstract Superclass

```
public abstract class Shape {
    protected String color;

    public Shape(String color) {
        this.color = color;
    }

    public abstract double area();
    public abstract double perimeter();

    public void displayInfo() {
        System.out.printf("%-12s | color=%-6s | area=%8.2f | perimeter=%8.2f%n",
            getClass().getSimpleName(), color, area(), perimeter());
    }

    @Override
    public String toString() {
        return getClass().getSimpleName() + "[color=" + color + "]";
    }
}
```

The Subclasses

```
public class Circle extends Shape {
    private double radius;

    public Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    public double area() { return Math.PI * radius * radius; }

    @Override
    public double perimeter() { return 2 * Math.PI * radius; }
}

public class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(String color, double width, double height) {
        super(color);
        this.width  = width;
        this.height = height;
    }

    @Override
    public double area() { return width * height; }

    @Override
    public double perimeter() { return 2 * (width + height); }
}
```

Polymorphism in Action

```
public class ShapeDriver {
    public static void main(String[] args) {
        Shape[] shapes = {
            new Circle("red",   5.0),
            new Circle("blue",  3.0),
            new Rectangle("green", 4.0, 6.0),
            new Rectangle("yellow", 10.0, 2.5)
        };

        System.out.printf("%-12s | %-8s | %8s | %10s%n",
            "Type", "Color", "Area", "Perimeter");
        System.out.println("-".repeat(48));

        for (Shape s : shapes) {
            s.displayInfo();   // polymorphism — correct version every time
        }
    }
}
```

```
Type         | Color    |     Area |  Perimeter
------------------------------------------------
Circle       | red      |    78.54 |     31.42
Circle       | blue     |    28.27 |     18.85
Rectangle    | green    |    24.00 |     20.00
Rectangle    | yellow   |    25.00 |     25.00
```

📋 What This Example Demonstrates

- **Abstract class** — `Shape` cannot be instantiated directly
- **Abstract methods** — `area()` and `perimeter()` have no body in `Shape`
- **Concrete subclasses** — `Circle` and `Rectangle` each implement both methods
- **super() call** — both subclass constructors pass `color` up to `Shape`
- **protected field** — `color` is accessible in subclasses directly
- **Polymorphism** — a single `Shape[]` array holds different types; one loop handles them all
- **Dynamic binding** — `s.displayInfo()` calls `area()` on the actual object, not just any shape

🔧 Try It Yourself

Add a `Triangle` class that extends `Shape`. It should store three side lengths (`a` , `b` , `c`). Use Heron's formula for area:

```
double s = (a + b + c) / 2;
double area = Math.sqrt(s * (s-a) * (s-b) * (s-c));
```

Then add a `Triangle` to the `shapes` array in `ShapeDriver`. Does it just work with the existing loop?

📋 Module 10 Summary

Concept	Key Point
Inheritance	Subclass extends superclass; "is-a" relationship; reuses code
`extends`	Declares the subclass; one superclass only
`super()`	Calls superclass constructor; must be first statement
Method overriding	Same signature in subclass; use `@Override`
`Object` class	Root of every class; provides `toString()`, `equals()`
Polymorphism	Superclass reference holds subclass object; dynamic binding at runtime
Abstract class	Cannot instantiate; can have abstract (no-body) methods
Interface	Pure contract; `implements`; multiple allowed per class
`protected`	Accessible in same package and subclasses; invisible to outside world

Module 10 Quiz

Circle the best answer for each question. Each question is worth 1 point.

1. Which keyword creates a subclass in Java?

A. `implements`

B. `inherits`

C. `super`

D. `extends`

2. What is the output of the following code?

```
public class Animal {
    public void speak() { System.out.println("..."); }
}
public class Dog extends Animal {
    @Override
    public void speak() { System.out.println("Woof"); }
}
Animal a = new Dog();
a.speak();
```

A. Compile error — cannot assign Dog to Animal variable

B. `...`

C. `Woof`

D. Nothing — speak() is not defined on Animal

3. A class that is declared `abstract` cannot be:

A. Instantiated directly with `new`

B. Extended by a subclass

C. Used as a variable type

D. Given a constructor

4. Which statement about constructors and inheritance is TRUE?

A. Constructors are inherited just like regular methods

B. A subclass can only have one constructor

C. Constructors are NOT inherited; each subclass defines its own

D. The `super()` call can appear anywhere in a constructor body

5. What does the `@Override` annotation do?

A. Forces the JVM to run the subclass method instead of the superclass method

B. Tells the compiler to verify you are actually overriding a method; catches typos

C. Makes the method private so it cannot be called from outside the class

D. Prevents further subclasses from overriding the method

6. Every Java class implicitly extends which class?

A. `java.lang.Base`

B. `java.lang.Class`

C. `java.lang.Super`

D. `java.lang.Object`

7. Consider this code:

```
public interface Flyable {
    void fly();
}
public class Bird extends Animal implements Flyable {
    @Override public void speak() { System.out.println("Tweet"); }
    @Override public void fly()   { System.out.println("Flap flap"); }
}
```

How many superclass/interface relationships does `Bird` have?

A. One — it only extends Animal

B. Three — Animal, Flyable, and Object

C. Two — it extends Animal and implements Flyable

D. This is a compile error; a class cannot extend and implement simultaneously

8. What is the access level of `protected`?

A. Accessible in the same package and in subclasses

B. Accessible anywhere in the program

C. Accessible only within the same class

D. Accessible only in subclasses, never in the same package

9. What is the primary difference between an **interface** and an **abstract class**?

A. Abstract classes can be instantiated; interfaces cannot

B. A class can implement multiple interfaces but can only extend one abstract class

C. Interfaces can have constructors; abstract classes cannot

D. There is no difference — they are interchangeable

10. Look at this code:

```
public abstract class Shape {
    public abstract double area();
}
public class Square extends Shape {
    private double side;
    public Square(double side) { this.side = side; }
    // area() NOT implemented here
}
Shape s = new Square(5);
```

What happens when you try to compile this?

A. It compiles and runs; area() returns 0.0 by default

B. Runtime error when `area()` is called

C. Compile error — `Square` must either implement `area()` or be declared `abstract`

D. It compiles because `Square` is a concrete class

✅ Answer Key

Question	Answer	Explanation
1	**D**	The `extends` keyword establishes an inheritance relationship.
2	**C**	Polymorphism allows a parent reference to call overridden methods in child classes.
3	**A**	Abstract classes cannot be instantiated directly — you must extend them.
4	**C**	Constructors are not inherited; subclasses call parent constructors with `super()`.
5	**B**	`@Override` tells the compiler to verify the method overrides a parent method.
6	**D**	The `Object` class is the root of every class hierarchy in Java.
7	**B**	Java supports single class inheritance but allows implementing multiple interfaces.
8	**A**	`protected` members are accessible within the same package and by subclasses.
9	**B**	Interfaces define method signatures; abstract classes can include implemented methods.
10	**C**	Abstract methods have no body and must be implemented by concrete subclasses.

📚 Open Educational Resource (OER) — CC BY 4.0

ITP 120 — Java Programming I — Northern Virginia Community College

Author: Randy Michak | Contribute on GitHub