a2 # oop

Polymorphism: The Power of Many Forms

TCSS 305 Programming Practicum

Polymorphism is the cornerstone of object-oriented programming. It's what transforms rigid, inflexible code into systems that gracefully handle new requirements. This guide explains what polymorphism is, how it works in Java, and why it's essential for building maintainable software—using the Road Rage assignment as our concrete example.

---

1 What is Polymorphism?

The word **polymorphism** comes from Greek: *poly* (many) + *morph* (form). In programming, it means: **the same operation behaves differently depending on the object performing it.**

Consider this scenario: you have a collection of vehicles—trucks, cars, bicycles, taxis. Each vehicle needs to choose a direction to move. Without polymorphism, you'd need separate code to handle each vehicle type. With polymorphism, you write one piece of code that works for all of them:

```
for (Vehicle v : vehicles) {
    Direction dir = v.chooseDirection(neighbors);  // Different behavior for
each type!
}
```

The method call `v.chooseDirection(neighbors)` is the same for every vehicle. But the *behavior*—which direction is chosen—depends entirely on what kind of vehicle `v` actually is. A Truck makes different decisions than a Bicycle, even though the code calling them looks identical.

This is the "magic" of OOP. One line of code, many possible behaviors.

Important

Polymorphism isn't just a convenience—it's a fundamental design principle that enables code to be extended without modification. You'll see this power clearly when we examine the Road Rage `advance()` method.

2 Method Overriding

Polymorphism works through **method overriding**. When a child class provides its own implementation of a method declared in a parent class or interface, it *overrides* that method.

2.1 The @Override Annotation

Always mark overridden methods with `@Override`:

```
public class Truck extends AbstractVehicle {

    @Override
    public Direction chooseDirection(Map<Direction, Terrain> theNeighbors) {
        // Truck-specific direction logic
    }

    @Override
    public boolean canPass(Terrain theTerrain, Light theLight) {
        // Truck-specific terrain/light logic
    }
}
```

The `@Override` annotation tells the compiler: "I intend to override a method from my superclass or interface." If you make a mistake—wrong method name, wrong parameters—the compiler catches it immediately.

Without @Override—silent bugs:

```
// Programmer thinks they're overriding chooseDirection
public Direction choseDirection(Map<Direction, Terrain> theNeighbors) {  //
Typo!
    // This creates a NEW method, doesn't override anything
}
```

The code compiles fine, but `chooseDirection` is never overridden. The bug might not surface until runtime.

With @Override—compiler catches the mistake:

```
@Override  // ERROR: Method does not override method from superclass
public Direction choseDirection(Map<Direction, Terrain> theNeighbors) {
    // Compiler immediately catches the typo
}
```

> **Tip**
>
> Make `@Override` a habit. Add it every time you override a method, without exception. The annotation costs nothing and catches subtle bugs that would otherwise waste hours of debugging.

2.2 Rules for Overriding

When overriding a method, you must follow these rules:

Rule	Explanation
Same method signature	Same name and parameter types (names can differ)
Covariant return types	Return type must be the same or a subtype
Access cannot be more restrictive	If parent method is `public`, override must be `public`
Cannot override `final` methods	The `final` keyword prevents overriding

```
// Parent class or interface declares:
public Direction chooseDirection(Map<Direction, Terrain> neighbors);

// Valid override in child class:
@Override
public Direction chooseDirection(Map<Direction, Terrain> theNeighbors) {
    // Parameter name can differ, but type must match
}
```

2.3 Cannot Override Final Methods

A method declared `final` cannot be overridden:

```
public final int getMass() {
    return MASS;  // Child classes cannot change this behavior
}
```

The `final` keyword is used when a method's behavior must remain constant across all subclasses—such as a vehicle's mass, which shouldn't change based on vehicle type beyond what the class defines.

2.4 Extending Parent Behavior

When overriding a method, you have two choices:

- **Replace** the parent's behavior entirely
- **Extend** it by calling `super.method()` and adding your own logic

The `super.method()` pattern is essential when the parent does setup work you need:

```
@Override
public void reset() {
    super.reset();       // Parent resets position, direction, poke count
    myCustomField = 0;   // Child resets its own state
}
```

Example: Taxi's crosswalk counter

Consider a `Taxi` class that tracks how long it has waited at a red crosswalk:

```
public class Taxi extends Car {
    private int myTimeAtRed;

    @Override
    public void reset() {
        super.reset();      // Reset inherited state (position, direction,
etc.)
        myTimeAtRed = 0;    // Reset Taxi-specific state
    }
}
```

Without the `super.reset()` call, the parent's fields (position, direction, enabled status) would never be reset—only `myTimeAtRed` would be cleared.

Common Bug: Forgetting `super.method()`

Forgetting to call `super.reset()` is a common bug. The code compiles fine, but parent state never resets. The bug might not appear until much later when a vehicle behaves strangely after a simulation reset.

```
@Override
public void reset() {
    // BUG: Forgot super.reset()!
    myTimeAtRed = 0;  // Parent state is now stale
}
```

Design implication: If you want child classes to be able to extend a method's behavior, don't mark it `final`. If you want to guarantee behavior cannot be changed at all, use `final`. This is a deliberate design decision you'll make in Assignment 2 when creating `AbstractVehicle`.

---

3 Dynamic Binding (Late Binding)

How does Java know which version of `chooseDirection()` to call? Through **dynamic binding**, also called **late binding**.

3.1 Compile Time vs. Runtime

Consider this code:

```
Vehicle v = new Truck(10, 5, Direction.NORTH);
Direction dir = v.chooseDirection(neighbors);
```

At **compile time**, Java only knows that `v` is of type `Vehicle`. It verifies that `Vehicle` has a `chooseDirection` method—but doesn't know which implementation will run.

At **runtime**, the JVM looks at the *actual object type*. `v` refers to a `Truck`, so the JVM calls `Truck.chooseDirection()`.

This decision happens every time the method is called:

```
Vehicle v;

v = new Truck(0, 0, Direction.NORTH);
v.chooseDirection(neighbors);  // Calls Truck.chooseDirection()

v = new Bicycle(0, 0, Direction.EAST);
v.chooseDirection(neighbors);  // Calls Bicycle.chooseDirection()

v = new Human(0, 0, Direction.SOUTH);
v.chooseDirection(neighbors);  // Calls Human.chooseDirection()
```

Same variable `v`, same method call, but the behavior changes based on what object `v` currently references.

3.2 The Lookup Process

When `v.chooseDirection(neighbors)` is called, the JVM:

1. Looks at the actual object `v` refers to (e.g., a `Truck`)
2. Searches that class for `chooseDirection`
3. If found, executes it; if not, searches the parent class
4. Continues up the hierarchy until a matching method is found

This is why you can treat all vehicles uniformly—the JVM handles routing each call to the correct implementation.

3.3 Contrast with Static Binding

Not all method calls use dynamic binding. **Static binding** (early binding) resolves the method at compile time:

Binding Type	When Used	Resolved At
Dynamic (late)	Instance methods on objects	Runtime
Static (early)	`static` methods, `private` methods, `final` methods	Compile time

```
// Static binding - resolved at compile time
Vehicle.getDefaultSpeed();  // Static method

// Dynamic binding - resolved at runtime
vehicle.chooseDirection(neighbors);  // Instance method
```

> **Note**
>
> Polymorphism only works with dynamic binding. Static methods cannot be overridden—they can be *hidden* by a method with the same signature in a subclass, but this isn't true polymorphism.

---

4 Abstract Classes: The Middle Ground

You've seen two extremes in Java's type system:

- **Interfaces** — Pure contracts with no implementation (before Java 8's default methods)
- **Concrete classes** — Full implementations with all methods defined

Abstract classes occupy the middle ground. They provide *partial* implementation—some methods are fully implemented, others are left abstract for subclasses to define.

4.1 Why Abstract Classes Exist

Consider the `Vehicle` interface in Road Rage:

```
public interface Vehicle {
    Direction chooseDirection(Map<Direction, Terrain> theNeighbors);
    boolean canPass(Terrain theTerrain, Light theLight);
    int getX();
    int getY();
    void setX(int theX);
    void setY(int theY);
    Direction getDirection();
    void setDirection(Direction theDirection);
    boolean isEnabled();
    void poke();
    // ... many more methods
}
```

If you implement this interface directly in each vehicle class (`Truck` , `Car` , `Bicycle` , etc.), you face a problem: **massive code duplication**.

Every vehicle stores `myX` and `myY` fields. Every vehicle implements `getX()` , `setX()` , `getY()` , `setY()` identically.

Without an abstract class, you'd copy-paste these implementations six times—once per vehicle type.

4.2 The Solution: Share What's Common, Defer What's Different

An abstract class lets you:

1. **Implement shared behavior once** — Both shared state (fields) and shared behavior (methods)
2. **Leave vehicle-specific behavior abstract** — `chooseDirection()` , `canPass()`

```
public abstract class AbstractVehicle implements Vehicle {
    // Shared state - ALL vehicles have these
    private int myX;
    private int myY;

    // Concrete methods - same for ALL vehicles
    @Override
    public int getX() {
        return myX;
    }
```

```
    @Override
    public void setX(int theX) {
        myX = theX;
    }

    // Abstract methods - DIFFERENT for each vehicle type
    @Override
    public abstract Direction chooseDirection(Map<Direction, Terrain>
theNeighbors);

    @Override
    public abstract boolean canPass(Terrain theTerrain, Light theLight);
}
```

Now each vehicle class (`Truck`, `Car`, `Bicycle`) extends `AbstractVehicle`:

- **Inherits** all the shared implementation (`getX()`, `setX()`, etc.)
- **Must implement** the abstract methods (`chooseDirection()`, `canPass()`)

4.3 The Three-Level Hierarchy

Level	Role	Example
Interface	Defines the contract—what operations must exist	`Vehicle` declares all methods
Abstract Class	Implements common behavior, leaves specific behavior abstract	`AbstractVehicle` implements getters/setters, leaves movement logic abstract
Concrete Class	Provides complete implementation	`Truck` implements `chooseDirection()` and `canPass()`

This is why abstract classes exist: **to reduce duplication while preserving flexibility.**

You write the shared code once in `AbstractVehicle`. Each subclass inherits it automatically and only implements what makes it unique.

4.4 You Cannot Instantiate Abstract Classes

Because abstract classes have incomplete implementations (the abstract methods), you cannot create instances of them:

```
// ERROR: Cannot instantiate AbstractVehicle
Vehicle v = new AbstractVehicle(0, 0, Direction.NORTH);
```

This makes sense—what would `v.chooseDirection(neighbors)` do? There's no implementation to call.

You can only instantiate concrete subclasses:

```
// OK: Truck is concrete
Vehicle v = new Truck(0, 0, Direction.NORTH);
```

Protected Constructors in Abstract Classes

Abstract classes often use `protected` constructors instead of `public` ones. Since you cannot instantiate the abstract class directly, there's no reason for outside code to call its constructor—only subclasses need access. Using `protected` signals this intent clearly.

```
public abstract class AbstractVehicle implements Vehicle {
    protected AbstractVehicle(int theX, int theY, Direction
theDirection) {
        // Subclasses call this via super(...)
    }
}
```

Testing Abstract Classes

While you cannot instantiate abstract classes directly, you can test their inherited behavior through their concrete subclasses. In Assignment 2, you'll test `AbstractVehicle` by writing tests for `Truck`, `Car`, and other concrete vehicles—each test exercises the inherited methods (`getX()`, `setX()`, etc.) to verify they work correctly.

For more focused testing, you can also create a minimal concrete subclass specifically for testing. This test subclass provides the simplest possible implementation of the abstract methods, allowing you to isolate and verify just the inherited behavior. See the testing guides for examples.

When to Use Abstract Classes

Use an abstract class when you have:

1. **Shared state or behavior** across multiple subclasses (fields and methods)
2. **Some methods that can be implemented generically** (like `getX()`, `setX()`)
3. **Other methods that must vary by subclass** (like `chooseDirection()`)

If there's no shared implementation—just a contract—use an interface instead. If everything can be implemented—no abstract behavior needed—use a concrete class.

---

5 Abstract Methods

Sometimes you want to *require* subclasses to provide an implementation without providing a default one yourself. This is what **abstract methods** are for.

5.1 Defining Abstract Methods

An abstract method has no body—just a signature followed by a semicolon:

```
public abstract class AbstractVehicle implements Vehicle {

    // Abstract method - subclasses MUST implement
    @Override
    public abstract Direction chooseDirection(Map<Direction, Terrain>
theNeighbors);

    // Abstract method - subclasses MUST implement
    @Override
    public abstract boolean canPass(Terrain theTerrain, Light theLight);

    // Concrete method - subclasses inherit this implementation
    @Override
    public int getX() {
        return myX;
    }
}
```

The abstract methods say: "Every vehicle must be able to choose a direction and determine if it can pass terrain—but I can't provide a sensible default. You must implement these yourself."

5.2 The Contract

Abstract methods create a **contract**: any concrete (non-abstract) subclass must provide implementations for all abstract methods. If you forget one, the compiler refuses to compile:

```
public class Truck extends AbstractVehicle {
    // ERROR: Truck must implement chooseDirection(Map<Direction, Terrain>)
    // ERROR: Truck must implement canPass(Terrain, Light)
}
```

This is exactly what you want. The compiler enforces that every vehicle type provides its own direction-choosing and terrain-checking logic.

5.3 Abstract Methods in A2

In Road Rage, `canPass()` and `chooseDirection()` should be abstract in your `AbstractVehicle` class. Why?

- **`canPass()`**: A Truck can ignore red lights. A Human must wait at crosswalks. There's no universal default—each vehicle type has unique terrain and light rules.
- **`chooseDirection()`**: A Bicycle prefers trails. A Car prefers streets. An ATV can go almost anywhere. Direction-choosing logic is specific to each vehicle type.

By making these methods abstract, you ensure that every new vehicle type *must* define its own behavior. The compiler won't let you forget.

---

6 Polymorphism in A2: The advance() Loop

Now let's see polymorphism in action. Here's the actual `advance()` method from the Road Rage simulation (simplified):

```
@Override
public void advance() {
    for (final Vehicle v : myVehicles) {
        final Map<Direction, Terrain> neighbors = generateNeighbors(v);

        // move the vehicle
        if (v.isEnabled()) {
            final Direction newDirection = v.chooseDirection(neighbors);
            v.setDirection(newDirection);

            // move one square in current direction, if it's okay to do so
            if (v.canPass(neighbors.get(newDirection), myLight)) {
                v.setX(v.getX() + newDirection.dx());
                v.setY(v.getY() + newDirection.dy());
```

```
            }
        } else {
            // become one move closer to revival
            v.poke();
        }

        // look for collisions
        for (final Vehicle other : myVehicles) {
            if (v == other) { // don't collide with self
                continue;
            }

            if (v.getX() == other.getX() && v.getY() == other.getY()) {
                // resolve collision using the current Comparator
                resolveCollision(v, other);
            }
        }
    }
    // ... system-level steps: advance timestep, change lights, notify
observers
}
```

6.1 What's Happening Here

The loop iterates over `myVehicles`, a `List<Vehicle>`. The code doesn't know—or care—what specific type each vehicle is. It just calls:

- `v.chooseDirection(neighbors)` — returns the direction this vehicle wants to move
- `v.canPass(terrain, light)` — returns whether the vehicle can enter that terrain

Each vehicle responds according to its own rules:

- A **Truck** can drive through red lights (returning `true` from `canPass` even when `light == Light.RED`)
- A **Human** can only move on grass, crosswalks, and trails
- A **Bicycle** prefers trails and must stop at yellow and red lights on crosswalks

The simulation doesn't implement any of these rules. It just asks each vehicle, and each vehicle knows its own behavior.

6.2 The Power: Zero Changes for New Vehicles

Here's the profound implication: **adding a new vehicle type requires ZERO changes to the `advance()` method.**

Suppose you want to add a `Motorcycle` class. You:

1. Create `Motorcycle extends AbstractVehicle`

2. Implement `chooseDirection()` with motorcycle-specific logic
3. Implement `canPass()` with motorcycle-specific rules
4. Add `Motorcycle` to `config/vehicles.txt`

The simulation works immediately. The `advance()` loop continues to iterate over `List<Vehicle>`, and when it encounters a `Motorcycle`, polymorphism ensures the correct methods are called.

No `if` statements. No switch cases. No modifications to existing code.

> **Important**
>
> This is the **Open/Closed Principle** in action: the simulation is *open for extension* (you can add new vehicle types) but *closed for modification* (you don't change existing code). Polymorphism makes this possible.

---

7 The Fragility Without Polymorphism

What would the code look like *without* polymorphism? Let's see the alternative—and understand why it's a maintenance nightmare.

7.1 The Giant if/instanceof Chain

Without polymorphism, you'd need to check each vehicle's type explicitly:

```
public void advanceWithoutPolymorphism() {
    for (final Vehicle v : myVehicles) {
        final Map<Direction, Terrain> neighbors = generateNeighbors(v);

        if (v.isEnabled()) {
            Direction newDirection;

            // Check type and apply specific logic
            if (v instanceof Truck) {
                // Truck prefers going straight, only turns if necessary
                if (neighbors.get(v.getDirection()) == Terrain.STREET
                        || neighbors.get(v.getDirection()) == Terrain.LIGHT
                        || neighbors.get(v.getDirection()) ==
Terrain.CROSSWALK) {
                    newDirection = v.getDirection();
                } else {
                    newDirection = findRandomValidDirection(v, neighbors);
                }
```

```
            } else if (v instanceof Car) {
                // Car can go on streets, lights, crosswalks
                // Different direction preference than Truck
                newDirection = findCarDirection(v, neighbors);
            } else if (v instanceof Bicycle) {
                // Bicycle prefers trails, can use streets and crosswalks
                newDirection = findBicycleDirection(v, neighbors);
            } else if (v instanceof Human) {
                // Human walks on grass, crosswalks, trails
                newDirection = findHumanDirection(v, neighbors);
            } else if (v instanceof Taxi) {
                // Taxi has complex stopping behavior at crosswalks
                newDirection = findTaxiDirection(v, neighbors);
            } else if (v instanceof Atv) {
                // ATV can go almost anywhere except walls
                newDirection = findAtvDirection(v, neighbors);
            } else {
                throw new IllegalStateException("Unknown vehicle type: " +
v.getClass());
            }

            v.setDirection(newDirection);

            // Now check if vehicle can pass... same pattern again
            boolean canMove;
            if (v instanceof Truck) {
                canMove = checkTruckCanPass(neighbors.get(newDirection),
myLight);
            } else if (v instanceof Car) {
                canMove = checkCarCanPass(neighbors.get(newDirection),
myLight);
            } // ... and so on for every vehicle type

            if (canMove) {
                v.setX(v.getX() + newDirection.dx());
                v.setY(v.getY() + newDirection.dy());
            }
        }
    }
}
```

7.2 Why This is Terrible

This approach has serious problems.

Problem 1: Violates the Open/Closed Principle

Every time you add a new vehicle type, you must modify this method. Add a `Motorcycle`? Edit `advanceWithoutPolymorphism()`. Add a `Skateboard`? Edit it again. The class is never "done"—it changes every time the system grows.

Problem 2: Easy to Forget Cases

With six `else if` blocks, it's easy to forget one when adding a new vehicle. The compiler won't catch it—you'll get a runtime exception (or worse, silent incorrect behavior) when the new vehicle type appears.

Problem 3: Logic Scattered Everywhere

Truck behavior is spread across multiple places: direction-choosing logic here, terrain-checking logic there, collision logic somewhere else. To understand how Trucks work, you need to search the entire codebase.

With polymorphism, all Truck behavior lives in `Truck.java`. Want to understand Trucks? Read one file.

Problem 4: Harder to Test

Testing the non-polymorphic version requires testing every code path in one massive method. Testing the polymorphic version means testing each vehicle class independently—smaller, focused, easier to maintain.

Problem 5: Breaks Encapsulation

The `advance()` method needs to know *intimate details* about how each vehicle type works. This tight coupling means changes to one vehicle's rules might break the central method.

> **Caution**
>
> If you find yourself writing long `if/else if` chains checking `instanceof`, stop and ask: "Should this behavior live in the objects themselves?" Usually, the answer is yes.

---

8 Tell, Don't Ask

Polymorphism embodies an important OOP principle: **"Tell, don't ask."**

8.1 Ask (Bad)

```
if (v instanceof Truck) {
    // ask what type it is, then do Truck-specific logic here
} else if (v instanceof Car) {
    // ask what type it is, then do Car-specific logic here
}
```

You're *asking* the object what type it is, then *you* decide what to do.

8.2 Tell (Good)

```
Direction dir = v.chooseDirection(neighbors);
```

You're *telling* the object what you need, and *it* decides how to accomplish it.

The difference is crucial. When you "tell," each object owns its own behavior. When you "ask," the calling code owns everyone's behavior—and becomes bloated, fragile, and hard to maintain.

> **Gen AI & Learning: Polymorphism and AI-Assisted Development**
>
> When working with AI coding assistants, polymorphism becomes even more valuable. If you describe your system using interfaces and polymorphic design, AI tools can understand the *contract* each class must fulfill and generate appropriate implementations. Conversely, if your code is a tangle of `instanceof` checks, AI tools (and humans) struggle to understand the intended behavior. Clean polymorphic design makes your code more understandable to both humans and AI collaborators.

---

9 When to Use Polymorphism

Polymorphism is appropriate when you have:

9.1 Multiple Types with Shared Behavior

You have several classes that need to perform the same *kind* of action, but each does it differently:

- Vehicles that choose directions (but each has different preferences)
- Shapes that draw themselves (but circles and rectangles draw differently)
- Payment methods that process transactions (credit card vs. PayPal vs. bank transfer)

9.2 Extensibility Requirements

You anticipate adding new types without modifying existing code:

- New vehicle types in Road Rage

- New file formats in a document editor
- New authentication providers in a login system

9.3 The "Tell, Don't Ask" Test

If you find yourself checking types to decide behavior, polymorphism is probably the answer:

```
// BEFORE: Asking
if (payment instanceof CreditCard) {
    processCreditCard((CreditCard) payment);
} else if (payment instanceof PayPal) {
    processPayPal((PayPal) payment);
}

// AFTER: Telling
payment.process();  // Each payment type knows how to process itself
```

9.4 When NOT to Use Polymorphism

Polymorphism isn't always the answer:

- **Simple data containers**: If classes just hold data without behavior, polymorphism may be overkill
- **Performance-critical tight loops**: Virtual method dispatch has a (tiny) overhead; in extreme cases, direct calls are faster
- **Truly divergent types**: If classes have nothing meaningful in common, forcing them into a hierarchy creates artificial abstraction

---

Summary

Concept	Key Point
Polymorphism	Same method call, different behavior based on actual object type
Method Overriding	Child class provides specific implementation of parent's method
@Override	Annotation that catches override mistakes at compile time
Dynamic Binding	JVM determines which method to call at runtime based on actual object type

Concept	Key Point
Abstract Methods	Force subclasses to provide implementation; no default behavior
Open/Closed Principle	Open for extension, closed for modification
Tell, Don't Ask	Let objects decide their own behavior; don't check types externally

Polymorphism transforms rigid code into flexible systems. In Road Rage, a single `advance()` loop handles every vehicle type—current and future—without modification. This is the power of object-oriented design: write code once, extend it forever.

Further Reading

External Resources

- Oracle Java Tutorial: Polymorphism - Official Java documentation
- Baeldung: Polymorphism in Java - Practical tutorial with examples
- Refactoring Guru: Replace Conditional with Polymorphism - Classic refactoring pattern
- Martin Fowler: Tell Don't Ask - Discussion of the principle

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 18: Favor composition over inheritance; Item 19: Design and document for inheritance or else prohibit it; Item 64: Refer to objects by their interfaces.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance — Polymorphism, Dynamic Binding, Abstract Classes.
- Reges, S., & Stepp, M. (2020). *Building Java Programs* (5th ed.). Pearson. Chapter 9: Inheritance and Interfaces — Polymorphism.

Language Documentation:

- Oracle Java Tutorial: Polymorphism — Official polymorphism documentation

- Oracle JDK 25: java.lang.Override — @Override annotation documentation

Design Principles:

- Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall. Chapter 9: The Open-Closed Principle.
- Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer*. Addison-Wesley. Tip 17: Program to an Interface, Not an Implementation.
- Open/Closed Principle (Wikipedia) — Overview of the OCP

This guide is part of TCSS 305 Programming Practicum, School of Engineering and Technology, University of Washington Tacoma.