a2 # java-fundamentals

Java Enums

TCSS 305 Programming Practicum

This guide explains Java's enumeration types (enums)—a feature that provides type-safe constants representing a finite set of named values. In Assignment 2, you'll work with three provided enums (`Direction`, `Light`, and `Terrain`) that define the movement rules for vehicles in the Road Rage simulation. Understanding how to *use* enums effectively is essential for implementing vehicle behavior.

---

1 What is an Enum?

An **enum** (short for enumeration) is a special Java type that represents a fixed set of constants. Unlike regular classes, an enum defines all possible values at compile time—you cannot create new instances at runtime. **Each enum constant is an object**—specifically, a singleton instance of the enum type.

```
public enum Light { GREEN, YELLOW, RED }
```

This declares a type called `Light` with exactly three possible **objects**: `GREEN`, `YELLOW`, and `RED`. These three objects are created when the class loads, and no other instances can ever exist.

1.1 Key Characteristics

Property	Meaning
Type-safe	The compiler enforces that only valid enum values can be used
Finite	All possible values are known at compile time
Named	Each value has a descriptive name (not just a number)

Property	Meaning
Singleton	Each enum constant exists as exactly one object

1.2 Enums vs. Plain Constants

Before enums existed (prior to Java 5), programmers used `int` or `String` constants:

```
// Old-style integer constants (problematic)
public static final int LIGHT_GREEN = 0;
public static final int LIGHT_YELLOW = 1;
public static final int LIGHT_RED = 2;

// Or string constants (also problematic)
public static final String DIRECTION_NORTH = "NORTH";
public static final String DIRECTION_SOUTH = "SOUTH";
```

These approaches have serious problems that enums solve.

---

2 Why Enums Exist

2.1 The Problem: Magic Numbers and Strings

Consider a method that takes a direction as an `int`:

```
// BAD: Using integers for directions
public void turn(int direction) {
    if (direction == 0) {        // What does 0 mean?
        // turn north
    } else if (direction == 1) { // What does 1 mean?
        // turn south
    }
    // What if someone passes 5? Or -1?
}
```

This code has multiple problems:

1. **No type safety** — Any integer value can be passed, including invalid ones
2. **No documentation** — What does `0` mean? You have to look it up
3. **Runtime errors** — Invalid values cause bugs that aren't caught until the program runs

4. **Easy typos** — Was north `0` or `1`? Easy to get wrong

String constants have similar issues:

```
// BAD: Using strings for directions
public void turn(String direction) {
    if (direction.equals("NORTH")) {  // What if someone passes "North"?
        // turn north
    }
    // Typos compile fine: "NROTH" won't cause a compiler error
}
```

2.2 The Solution: Type Safety

Enums solve all these problems:

```
// GOOD: Using an enum
public void turn(Direction direction) {
    switch (direction) {
        case NORTH -> // turn north
        case SOUTH -> // turn south
        case EAST  -> // turn east
        case WEST  -> // turn west
    }
}
```

Benefit	Explanation
Compiler enforcement	Only `Direction.NORTH`, `SOUTH`, `EAST`, or `WEST` can be passed
Self-documenting	The code reads clearly: "turn north"
IDE autocomplete	Type `Direction.` and see all valid options
No invalid values	Cannot accidentally pass `Direction.NROTH` (won't compile)
Refactoring safety	Rename an enum constant and the IDE updates all usages

Compile-Time vs. Runtime Errors

With `int` or `String` constants, mistakes become runtime bugs that might not appear until your program is running (or worse, in production). With enums, the compiler catches these mistakes immediately—you can't even build a program that uses an invalid enum value.

Gen AI & Learning: Enums and Code Generation

AI coding assistants understand enums well and can generate switch statements, `canPass()` implementations, and direction logic. However, you should verify that the generated code matches the specific vehicle behavior requirements. An AI might generate valid Java but implement the wrong business logic—for example, making a Bicycle stop at green lights instead of yellow and red. Always check the generated behavior against the assignment specifications.

---

3 Using Enums

3.1 Declaring Variables

Declare enum variables like any other type:

```
// Declare and initialize
Direction currentDirection = Direction.NORTH;

// Can reassign to other values of the same type
currentDirection = Direction.EAST;

// Can be null (but usually shouldn't be)
Direction maybeDirection = null;
```

3.2 Comparing Enum Values

Enums can be compared using either `==` or `.equals()`, but `==` is preferred:

```
Direction dir = Direction.NORTH;

// PREFERRED: Use == for enum comparison
if (dir == Direction.NORTH) {
    System.out.println("Heading north!");
}

// Also works, but unnecessary
if (dir.equals(Direction.NORTH)) {
    System.out.println("Heading north!");
}
```

Why `==` is preferred:

1. **Null-safe** — `==` never throws `NullPointerException`, while `.equals()` does if the object on the **left** is null
2. **Clearer intent** — Signals that you're comparing identity, which is exactly what you want for enums
3. **Marginally faster** — Direct reference comparison vs. method call (negligible in practice, but still)

```
Direction dir = null;

// SAFE: Returns false
if (dir == Direction.NORTH) { }

// UNSAFE: Throws NullPointerException!
if (dir.equals(Direction.NORTH)) { }
```

Both Work Because Enums are Singletons

Each enum constant is guaranteed to be a single object. `Direction.NORTH` is always the exact same object in memory, so reference equality (`==`) and value equality (`.equals()`) produce identical results. This is different from `String`, where two strings with the same characters might be different objects.

3.3 Switch Statements with Enums

Enums work beautifully with switch statements. You can use the traditional or modern switch syntax:

```
// Modern switch expression (Java 14+)
String message = switch (light) {
    case GREEN  -> "Go!";
    case YELLOW -> "Caution!";
    case RED    -> "Stop!";
};

// Traditional switch statement
switch (light) {
    case GREEN:
        System.out.println("Go!");
        break;
    case YELLOW:
        System.out.println("Caution!");
        break;
    case RED:
        System.out.println("Stop!");
        break;
}
```

Exhaustiveness Checking

When using switch expressions (with `->` and returning a value), the compiler verifies that all enum values are handled. If you forget a case, you get a compile error. This is another safety benefit of enums.

3.4 Iterating Over All Values

Every enum automatically has a `values()` method that returns an array of all constants:

```
// Print all directions
for (Direction dir : Direction.values()) {
    System.out.println(dir);
}
// Output: NORTH, WEST, SOUTH, EAST
```

This is useful for:

- Populating dropdown menus
- Testing all possible values
- Generating documentation

3.5 Getting the Name: `name()` and `toString()`

```
Direction dir = Direction.NORTH;

System.out.println(dir.name());      // "NORTH" (always the exact constant
name)
System.out.println(dir.toString());  // "NORTH" (same by default, but can be
overridden)
System.out.println(dir);             // "NORTH" (implicitly calls toString())
```

The `name()` method always returns the exact constant name as declared. The `toString()` method returns the same by default but can be overridden in custom enums.

3.6 Getting the Ordinal: `ordinal()`

Each enum constant has an **ordinal**—its position in the declaration (starting from 0):

```
System.out.println(Direction.NORTH.ordinal()); // 0
System.out.println(Direction.WEST.ordinal());  // 1
System.out.println(Direction.SOUTH.ordinal()); // 2
System.out.println(Direction.EAST.ordinal());  // 3
```

Avoid Relying on Ordinal Values

While `ordinal()` exists, you should rarely use it in application logic. If someone reorders the enum constants, all your ordinal-based code breaks silently. Joshua Bloch advises in *Effective Java*: "Never derive a value associated with an enum from its ordinal."

```
// BAD: Depends on declaration order
if (direction.ordinal() < 2) {  // What does 2 mean?
    // ...
}

// GOOD: Explicit and clear
if (direction == Direction.NORTH || direction == Direction.WEST)
{
    // ...
}
```

3.7 Converting from String: `valueOf()`

Convert a string to an enum constant using `valueOf()`:

```
Direction dir = Direction.valueOf("NORTH");  // Returns Direction.NORTH

// Case-sensitive! This throws IllegalArgumentException:
Direction bad = Direction.valueOf("north");  // Exception!
```

---

4 A2 Enums

Assignment 2 provides three enums in the `logic` package. You'll use these extensively when implementing vehicle behavior.

4.1 Direction

The `Direction` enum represents the four cardinal directions a vehicle can face or move:

```
public enum Direction {
    NORTH, WEST, SOUTH, EAST;

    public static Direction random()  // Returns a random direction
    public Direction left()           // Returns direction 90° counter-
clockwise
```

```
    public Direction right()          // Returns direction 90° clockwise
    public Direction reverse()        // Returns the opposite direction
}
```

Helper method examples:

```
Direction dir = Direction.NORTH;

dir.left();     // Returns Direction.WEST
dir.right();    // Returns Direction.EAST
dir.reverse();  // Returns Direction.SOUTH

// Chain calls
dir.left().left();  // Returns Direction.SOUTH (turned left twice)

// Random direction (useful for poke() when vehicle becomes enabled again)
Direction randomDir = Direction.random();
```

Common usage in A2:

```
@Override
public Direction chooseDirection(Map<Direction, Terrain> theNeighbors) {
    Direction current = getDirection();

    // Check if we can go straight
    if (canGoStraight(theNeighbors.get(current))) {
        return current;
    }

    // Try turning left
    Direction left = current.left();
    if (canGoStraight(theNeighbors.get(left))) {
        return left;
    }

    // Try turning right
    Direction right = current.right();
    if (canGoStraight(theNeighbors.get(right))) {
        return right;
    }

    // Last resort: reverse
    return current.reverse();
}
```

4.2 Light

The `Light` enum represents traffic light states:

```
public enum Light {
    GREEN, YELLOW, RED;
```

```
    public Light advance()  // Returns the next light in the cycle
}
```

The `advance()` method cycles through lights:

```
Light light = Light.GREEN;
light.advance();  // Returns YELLOW
light.advance().advance();  // Returns RED (GREEN -> YELLOW -> RED)

// The cycle wraps: RED.advance() returns GREEN
Light.RED.advance();  // Returns GREEN
```

Using Light in `canPass()`:

```
@Override
public boolean canPass(Terrain theTerrain, Light theLight) {
    boolean result = false;

    if (theTerrain == Terrain.TRAIL) {
        result = true;  // Fictional vehicle: always passes trails
    } else if (theTerrain == Terrain.LIGHT) {
        // Fictional rule: only stops for yellow
        result = theLight != Light.YELLOW;
    } else if (theTerrain == Terrain.GRASS) {
        // Fictional rule: passes grass only on green
        result = theLight == Light.GREEN;
    }

    return result;
}
```

4.3 Terrain

The `Terrain` enum represents the types of terrain on the map:

```
public enum Terrain {
    GRASS, STREET, LIGHT, WALL, TRAIL, CROSSWALK
}
```

Terrain	Description
`GRASS`	Open grass area—only certain vehicles can traverse
`STREET`	Regular road—most vehicles travel here
`LIGHT`	Traffic light intersection—vehicles may need to stop

Terrain	Description
`WALL`	Impassable barrier—no vehicle can enter
`TRAIL`	Off-road trail—only bicycles prefer this terrain
`CROSSWALK`	Pedestrian crossing—special light rules apply

Using Terrain in vehicle logic:

```
// Bus: can only travel on streets, lights, and crosswalks
private boolean isValidTerrain(Terrain terrain) {
    return terrain == Terrain.STREET
        || terrain == Terrain.LIGHT
        || terrain == Terrain.CROSSWALK;
}

// ATV: can travel on everything except walls
private boolean isValidTerrain(Terrain terrain) {
    return terrain != Terrain.WALL;
}

// Dirigible: can fly over everything!
private boolean isValidTerrain(Terrain terrain) {
    return true;
}
```

4.4 Putting It All Together

The `chooseDirection()` method receives a `Map<Direction, Terrain>` representing the four neighboring cells:

```
@Override
public Direction chooseDirection(Map<Direction, Terrain> theNeighbors) {
    // theNeighbors maps each direction to the terrain in that direction
    // Example: if vehicle is at (5, 5) facing NORTH:
    //   theNeighbors.get(Direction.NORTH) -> terrain at (5, 4)
    //   theNeighbors.get(Direction.SOUTH) -> terrain at (5, 6)
    //   theNeighbors.get(Direction.EAST)  -> terrain at (6, 5)
    //   theNeighbors.get(Direction.WEST)  -> terrain at (4, 5)

    Direction current = getDirection();

    // Check what's ahead
    Terrain ahead = theNeighbors.get(current);

    // Check what's to the left
    Terrain toLeft = theNeighbors.get(current.left());
```

```
    // Make a decision based on terrain and vehicle rules
    // ...
}
```

Summary

Concept	Key Point
What is an enum	A type-safe way to represent a fixed set of named constants
Why enums exist	Prevent bugs from magic numbers/strings; compiler catches invalid values
Comparing enums	Use `==` (preferred) or `.equals()`; both work because enums are singletons
Switch statements	Enums work naturally with switch; modern syntax provides exhaustiveness checking
`values()`	Returns array of all enum constants
`name()`	Returns the exact constant name as a String
`ordinal()`	Returns position in declaration (avoid using in logic)
A2: Direction	NORTH, WEST, SOUTH, EAST with `left()`, `right()`, `reverse()`, `random()`
A2: Light	GREEN, YELLOW, RED with `advance()` for cycling
A2: Terrain	GRASS, STREET, LIGHT, WALL, TRAIL, CROSSWALK

Further Reading

External Resources

- Oracle Java Tutorials: Enum Types — Official tutorial on enums
- Baeldung: A Guide to Java Enums — Comprehensive practical guide
- Oracle JDK 25: Enum Class — Official Enum class documentation

---

References

Primary Texts:

- Bloch, J. (2018). *Effective Java* (3rd ed.). Addison-Wesley. Item 34: Use enums instead of int constants. Item 35: Use instance fields instead of ordinals.
- Horstmann, C. S. (2022). *Core Java, Volume I: Fundamentals* (12th ed.). Oracle Press. Chapter 5: Inheritance — Section on Enumeration Classes.

Language Documentation:

- Oracle JDK 25: Enum Types — Official Java tutorial on enumeration types
- Oracle JDK 25: java.lang.Enum — Enum base class documentation

Additional Resources:

- Java Language Specification: Enum Types — Formal specification of enum behavior

---

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