A07-Robot

COMP301

Figure 1: "Come with me if you want to live"

Introduction

The year is 2077, and the zombie apocalypse has reached the heart of Chapel Hill. UNC, once a thriving center of learning and innovation, is now one of the last strongholds of human resistance. The outbreak spread faster than anyone predicted, overwhelming cities and turning technology against its creators. But all is not lost. Deep within the campus lie several experimental robotics labs—homes to decades of research in autonomous systems, AI, and mechanical augmentation.Survival demands evolution. Our task is urgent and clear: upgrade our robots. The undead are closing in, and the clock is ticking. If humanity is to make a stand, it starts here—with us, and with the machines we build. We will use decorators to do that!

Enums Aside

Let's talk about enums. In Java, enums are not just syntactic sugar for int constants—they are full-blown class-like objects that can encapsulate behavior and data. This is especially useful in strategy-like patterns, state machines, or configuration-based logic like game difficulty, robot modes, or UI states.

While they are commonly used to define a fixed set of constants, Java enums can also have fields, constructors, and methods, making them behave similarly to classes.

Core Concepts

- Each value of an enum is an instance of the enum class.
- You can add state variables (i.e., fields) to each enum constant.
- You can define a constructor (which must be private) to initialize those fields.
- You can define methods that can be called on each enum instance.

In this assignment, we will assign the enum a `toString()` method. You can see an example of how methods work with enums below. Example:

```
public enum Direction {
  UP(0, 1),
  DOWN(0, -1),
  LEFT(-1, 0),
  RIGHT(1, 0);

  private final int dx;
  private final int dy;

  Direction(int dx, int dy) {
    this.dx = dx;
    this.dy = dy;
  }

  public int getDx() {
    return dx;
  }

  public int getDy() {
    return dy;
  }

  // Applies the direction to a given (x, y) point and returns the new point
  public int[] move(int x, int y) {
    return new int[] {x + dx, y + dy};
  }
}

public class Main {
    public static void main(String[] args) {
        int x = 5;
        int y = 3;

        Direction moveDir = Direction.UP;
        int[] newCoords = moveDir.move(x, y);

        System.out.println("Moved " + moveDir + " to: (" + newCoords[0] + ", "
            + newCoords[1] + ")");
        // Output: Moved UP to: (5, 4)
    }
}
```

JavaFX Overview

JavaFX is a modern UI framework for Java, allowing you to build rich desktop applications. In this assignment, you'll use JavaFX to display your robot's image and stats on-screen. We rely on five core JavaFX classes:

Parent

A base class for all JavaFX nodes that can contain child nodes. In this project, all visual-returning methods use the `Parent` type, so you can return different layouts (e.g., `StackPane`, `HBox`).

StackPane

A layout that stacks its children on top of each other, with the first child at the bottom. We use `StackPane` to layer robot images on top of each other.

How it works:
Child nodes are drawn in the order they appear in `getChildren()`, so later additions appear on top.

Methods you'll use:

- `new StackPane(Node... children)` — creates a pane with initial children.
- `getChildren().add(Node child)` — adds a node on top.
- `getChildren().addFirst(Node child)` — adds a node at the bottom.

HBox

A horizontal layout pane that arranges its children in a single row. We use `HBox` to build health and shield bars side by side.

How it works:
Each child node is placed next to the previous one, from left to right.

Methods you'll use:

- `new HBox()` — creates an empty horizontal box.
- `hbox.getChildren().add(Node child)` — appends an icon to the bar.

Image & ImageView

To display images in JavaFX, use the `Image` and `ImageView` classes. You'll load PNG files for the robot base and decorators.

How it works:
Image loads the raw pixel data from a resource path or URL. An **ImageView** is a node that displays an `Image` in the scene graph, and you can resize icons by setting their fit dimensions.

Methods you'll use:

- `new Image(String url)` — loads an image from the given file path (e.g., "base.png").
- `new ImageView(Image image)` — creates a view node to display the image.
- `imageView.setFitWidth(double value)` — scales the width to the given number of pixels.
- `imageView.setFitHeight(double value)` — scales the height to the given number of pixels.

Apprentice

This stage is where we set up everything. We cannot give our robot upgrades just yet, but we will prepare for those upgrades by defining a few classes and enums to help us along the way.

Robot interface (`Robot.java`)

This interface is given to you with the following methods:

```
public interface Robot {
    String getDescription();
    int getHealth();
    int getShield();
    int getPower();
    Parent getHealthBar();
    Parent getShieldBar();
    Parent getVisual();
}
```

BasicRobot (`BasicRobot.java`)

The `BasicRobot` represents your starting robot. Create an instance variables, constructors, and methods as necessary so that it has the following properties:

- Description: "Basic Robot".
- Stats: has 50 health , 0 shield , and 10 power.
- JavaFX: for now, `getVisual()`, `getHealthBar()`, and `getShieldBar()` should all return `null`.

ArmorType (`ArmorType.java`)

Define armor tiers as an `enum` with values `BRONZE`, `IRON`, `GOLD`, and `DIAMOND`.
We use `ArmorType` later in decorators to determine shield bonuses.

1. Add a constructor that takes in and initializes an instance variable called `shieldUpgrade`. It should be set to the following values:
    - BRONZE: 10
    - IRON: 15
    - GOLD: 20
    - DIAMOND: 25
2. Create a getter method for the same field.
3. Override `toString()` to return the enum name (e.g., "BRONZE").

PowerType (`PowerType.java`)

Define power levels as an `enum` with values `APPRENTICE`, `ENCHANTER`, and `SORCERER`.
We use `PowerType` later to determine health multiplier and power boost.

1. Add a constructor that takes in and initializes an instance variable called `multiplier`. It should be set to the following values:
    - APPRENTICE: 1
    - ENCHANTER: 2
    - SORCERER: 3
2. Write a getter method for this field.
3. Override `toString()` to return the enum name (e.g., "SORCERER").

RobotDecorator (`RobotDecorator.java`)

The `RobotDecorator` is base class that wraps another `Robot` instance. It exists to help you **practice the DRY (Don't Repeat Yourself) principle**: instead of re-implementing every method in each decorator, you delegate default behavior to this class and only override the methods you need. You should not be able to make an instance of this class.

- Delegation: Any method not overridden by a subclass should delegate to the encapsulated instance as a default.
- JavaFX: For the Apprentice and Enchanter stages `getHealthBar()`, and `getShieldBar()` should simply return `null`. In the Sorcerer stage, you'll provide real implementations.
  - `getVisual()` should be an abstract method and will not have an implementation in this class.

Enchanter

This is the stage that sets us up for success. We implement most of the logic for our upgrades in this stage. We will create six new classes that will be used to "decorate" our robot and dynamically add more upgrades to him as we like. Each decorator class will take in a `Robot` and a `ArmorType` to determine the amount of stat change as described below:

Class	getDescription	getHealth	getShield	getPower
ArmsArmorDecorator	Append `", [TYPE] Arms Armor"`	No change	bonus: BRONZE+10, IRON+15, GOLD+20, DIAMOND+25	No change
HeadArmorDecorator	Append `", [TYPE] Head Armor",`	No change	+ same bonus	No change
LegsArmorDecorator	Append `", [TYPE] Legs Armor",`	No change	+ same bonus	No change
TorsoArmorDecorator	Append `", [TYPE] Torso Armor",`	No change	+ same bonus	No change
JetpackDecorator	Append `", Jetpack",`	No change	+50	No change
PowerUpDecorator	Append `", [TYPE] Power Up",`	*Multiply health by factor:* APPRENTICE×1, ENCHANTER×2, SORCERER×3	No change	*Multiply power by factor:* APPRENTICE×10, ENCHANTER×20, SORCERER×30

Sorcerer

At last! Now that we finished up all the logic, we get to bring our robot to life and see our upgrades in action. In this stage we will use JavaFX to get it all hooked up and finally run our main method to see everything.

Note: We will be creating visuals associated with our robot. To display a png, use must create an `Image` that is injected into an `ImageView`

```
ImageView iv = new ImageView(new Image("base.png"));
```

BasicRobot UI Methods

Even if we never wrap a `BasicRobot` in a decorator, it still needs to display health and shield bars. Implement these methods in **BasicRobot**:

- `getHealthBar()`: For every full 10 points of health, add a heart icon (`health.png`) where each icon scales to 30×30 pixels. If there's any remaining health beyond those tens, add a half-heart (`health_half.png`) the same way. Each heart should be placed next to each other in a row.
- `getShieldBar()`: For every full 10 points of shield, add an shield icon (`bar.png`) resized to 30×30. If there's any leftover shield, add a half-bar (`bar_half.png`) similarly.

These implementations ensure that even the plain `BasicRobot` shows its stats.

RobotDecorator UI Methods

To avoid repeating code, **override these methods only once** in `RobotDecorator`:

- `getHealthBar()`: Your implementation for this will be identical to the implementation you have for BasicRobot. However, we need to do it again here to account for all decorators displaying the right health bars based on their stats.
- `getShieldBar()`: Likewise, implementation will be identical to above.

Visual Layering

At this point, our decorators will all have unique implementations. The abstract class `RobotDecorator` should leave getVisuals as an abstract method since there is no generic implementaiton for this method.

For the concrete decorator classes, all `getVisual()` methods should stack images in a `StackPane`. The base image comes from `BasicRobot.getVisual()`. Each decorator adds its own layer:

1. **BasicRobot.getVisual():**

```
ImageView iv = new ImageView(new Image("base.png"));
StackPane pane = new StackPane(iv);
return pane;
```

2. **Decorator.getVisual():** (in each decorator)

- Cast the result of `decoratedRobot.getVisual()` to `StackPane`.
- Create an `ImageView` for this decorator's image:
  - For **BRONZE**, use the base filename without color (e.g., `arms.png`). You can find these images under `resources`
  - For other tiers, include the body part and color in the filename (e.g., `arms_iron.png`, `head_gold.png`, `torso_diamond.png`).
  - For the jetpack, use `jetpack.png`.
  - For power-ups, use `yellow_power.png`, `red_power.png`, `blue_power.png` with APPRENTICE, ENCHANTER, and SORCERER respectively.

- The armor does not need any scaling. However for powerups and icons, call `imageView.setFitWidth(...)` and `imageView.setFitHeight(...)` with values appropriate for your image (e.g., 600×700 for power-ups, 30×30 for small icons) to scale correctly.
- Use `pane.getChildren().add(...)` to place the new layer on top, except for `PowerUpDecorator`, which should use `addFirst(...)` to sit behind other layers.
- Return the same pane.

Build your Robot

In the `Main` class, you will see a method called build. Right now, that method just returns a Basic Robot. Create your own decorated robot. After all of its modifications, the robot should have the following stats:
Health: 300
Shield: 120
Power: of your choice

Merlin: Create Your Own!

Create your own decorator module! It must:

- extend `RobotDecorator` and follow the format we have established.
- have its own visual image (png) that is properly formatted for the JavaFX and fits on the Image Stack to visually change the result. You are welcome to use AI to generate your image, but you will likely have to modify it to fit our robot.
- modify the stats according to what it does. The final stats of the `build()` method need to stay the same, but you should use your module somewhere in the decorator stack.

Congratulations! You just saved the entirety of UNC students by programming the logic for robot upgrades. We can now always rely on your system to upgrade all our robots. You can now successfully use the `build()` method in `Main.java` to chain your decorators with whatever parameters you like to see your custom upgraded robot in action!

Deck out your robot, and submit to gradescope!