SWEN20003
Object Oriented Software Development
Workshop 5

Eleanor McMurtry

Semester 2, 2020

Workshop

This week, we are learning all about inheritance.

- Inheritance allows many classes (the **subclasses**, **derived**, or **child classes**) to share attributes and methods from one class (the **parent**, **base**, or **superclass**).
- Subclasses automatically **inherit** all attributes and methods from their parent.
- Subclasses can call the parent's constructor with `super()`.
- Inheritance represents an **is-a** relationship; if `Circle` extends `Shape`, then `Circle` **is a** shape.
  - This means subclasses can be **upcasted** to their parent class type:
    ```
    Circle circle = new Circle();
    Shape shape = circle; // upcast happens here
    ```
  - A reference to a parent class can only be safely **downcasted** to the subclass type using `instanceof`.
    ```
    if (shape instanceof Circle) {
        circle = (Circle) shape;
    }
    ```
  - Upcasting is an example of **polymorphism**.
- Methods in a parent class can be **overridden** by subclasses. This replaces their functionality, **even if upcasting is used.**

Questions

1. Download the chess example from Canvas, and run it in IntelliJ. Make sure you understand how it works. Implement the following additional chess pieces:
   - `Pawn`, which can move two spaces vertically if it hasn't yet moved, otherwise one space vertically
   - `Bishop`, which can move any number of spaces diagonally
2. Consider the below class.

```
public class Shape {
    public final double x;
    public final double y;

    public Shape(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getArea() { return 0; }
    public double getPerimeter() { return 0; }
    public String toString() { return "Plain Shape"; }
}
```

(a) Create the following subclasses:
- `Rectangle`, which has a **width** and **height**
- `Circle`, which has a **radius**

(b) Override the `getArea()` and `getPerimeter()` methods in your subclasses.

(c) Write appropriate `toString()` methods in your subclasses using the overridden methods.

(d) Make the `getArea()` and `getPerimeter()` methods abstract (in the `Shape` class).

(e) Write a `main` class. You should create a `Shape` array with 10 shapes (randomly chosen between `Circle` and `Rectangle`), where their parameters are chosen randomly between 0 and 5. Loop over this array and print the result of calling `toString()` on them.

3. Consider the below classes.

```
public class HttpRequest {
    private static final short PORT = 80;
    public final String address;
    public final String file;
    public final String method;

    public HttpRequest(String address, String file, String method) {
        this.address = address;
        this.file = file;
        this.method = method;
    }

    public short getPort() {
        return PORT;
    }

    public String getAddress() {
        return address;
    }

    public String getFullRequest() {
        return String.format("%s %s HTTP/1.1\r\n\r\n", method, file);
    }
}
public class FtpRequest {
    private static final short PORT = 21;
    public final String address;
    public final String file;

    public FtpRequest(String address, String file) {
        this.address = address;
        this.file = file;
    }

    public short getPort() {
        return PORT;
    }

    public String getAddress() {
        return address;
    }

    public String getFullRequest() {
        return String.format("RETR %s\u0015\u0012", file);
    }
}
```

(a) Create a `Request` base class with appropriate methods, including at least one abstract method. Change the above classes to inherit from `Request`.

(b) Write a `main` method to test your code. It should accept **command-line arguments** (via the `args` parameter of `main`). To test these in IntelliJ, click the main class name in the top-right (next to the Play button), click `Edit Configurations...`, and add data to the `Program arguments` field.

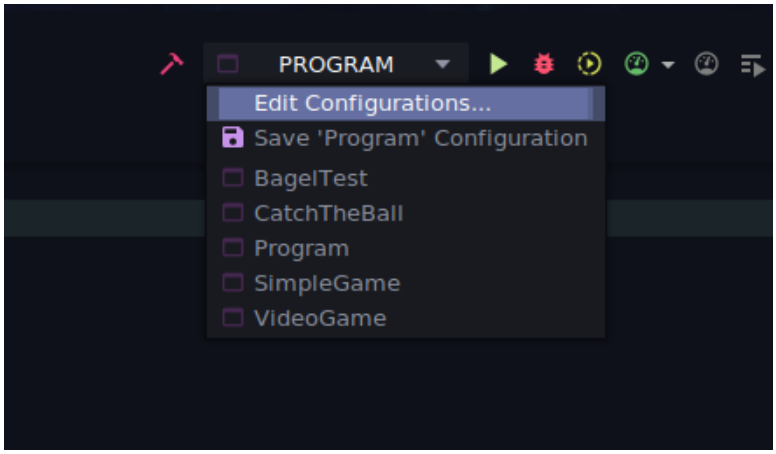

Figure 1: Accessing the Edit Configurations dialog.

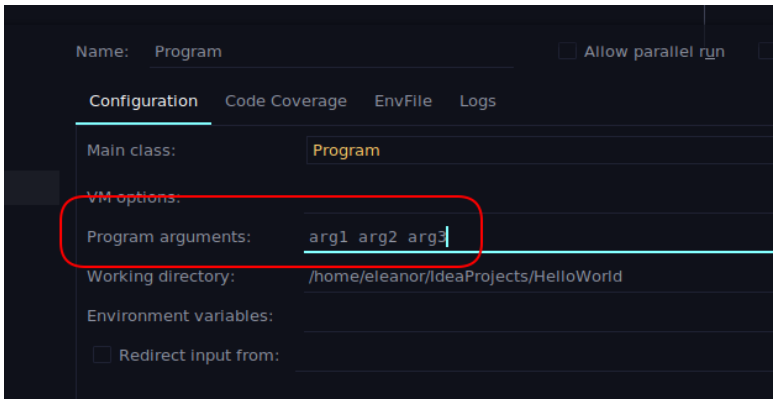

Figure 2: Adding command-line arguments to your project.

Your program should accept each of the following formats:

- `http google.com GET /`, in which case it should create an instance of `HttpRequest` with address `google.com`, method `GET`, and file `/`
- `ftp eleanorm.info swen20003.txt`, in which case it should create an instance of `FtpRequest` with address `eleanorm.info` and file `swen20003.txt`

It should create an appropriate instance and upcast it to `Request`, then print the result of calling `getFullRequest()`.

4. We will now use inheritance with data structures.

(a) Define a `Node` class, containing a `public final` integer (its **value**—) and two `Nodes` (the **next** and **previous** nodes). Add a getter for each node. (The node may be `null`, meaning there is no next/previous node.)

(b) Define a `LinkedList` class that contains a node (the **root**). It should have the following methods:

- `Node root()`: return the root node
- `int length()`: return the length (i.e. the number of nodes in the list)[1]
- `void append(int value)`: create a new node with the appropriate value, and add it to the end of the list
- `void insert(int index, int value)`: create a new node with the appropriate value, and insert it at the index (starting from 0)
- `void remove(int n)`: remove the nth node (starting from 0)

[1] Bonus challenge: make this $O(1)$.

(c) Define a `Stack` class inheriting from `LinkedList`. Add the following methods:

- `void push(int value)`: add a node with the appropriate value at the end of the list
- `Integer pop()`: remove the last node and return its value (or `null` if there is no such node)

(d) Define a `Queue` class inheriting from `LinkedList`. Add the following methods:

- `void enqueue(int value)`: add a node with the appropriate value to the end of the list
- `Integer take()`: remove the first node and return its value (or `null` if there is no such node)

5. We are ambitious Java enthusiasts and are already ready to begin creating our own small 'graphics' library. Our first task is to **design** a system to render simple shapes onto the screen. For now, we are concerned about two types of shapes in particular: **square**s and **triangle**s. A shape has a specific area associated with it, and it can also be rendered to the screen. **You are not required to implement any rendering logic.** For simplicity, you are to print a description of the shape to be rendered to the console instead of rendering anything to the screen.

   A shape also has a **colour** associated with it. We will be using the the RGB colour system which specifies a colour through three values: *red*, *green*, *blue*. The red, green, and blue values of a colour must be within the range of 0-255 (inclusive) at all times. If a colour is not specified, a shape's default colour is black (red = 0, green = 0, blue = 0). Your `Colour` implementation should be immutable. **The system should be compatible with the following driver class:**

```
public class Driver {

    private static final int MAX_SHAPES = 4;

    public static void main(String[] args) {
        Shape[] shapes = new Shape[MAX_SHAPES];
        // Black rectangle (red=0, green=0, blue=0) with width 20.52px and height 50px
        shapes[0] = new Rectangle(20.52, 50);
        // Crimson-coloured triangle (r=220, g=20, b=60) with base 392.2px and height 0.01px
        shapes[1] = new Triangle(392.2, 0.01, new Colour(220, 20, 60));
        // White (r=255, g=255, b=255) rectangle with width 50px and height 50.3px
        shapes[2] = new Rectangle(50, 50.3, Colour.WHITE);
        // Black triangle (red=0, green=0, blue=0) with base 10px and height 20.12px
        shapes[3] = new Triangle(10, 20.12);

        for (Shape shape : shapes) {
            shape.render();
        }

        double average = 0;
        for (int i = 0; i < MAX_SHAPES; i++) {
            average = (average * i + shapes[i].getArea()) / (i + 1);
        }
        System.out.format("Average area of rendered shapes: %.2f\n", average);
    }
}
```

   Example output:

```
Drawing a Rectangle with colour:(0, 0, 0) and area:1026.00px2
Drawing a Triangle with colour:(220, 20, 60) and area:1.96px2
Drawing a Rectangle with colour:(255, 255, 255) and area:2515.00px2
Drawing a Triangle with colour:(0, 0, 0) and area:100.60px2
Average area of rendered shapes: 910.89px2
```