Lab 08 C212 FA22

Last modified 2022-10-19 3:00 PM

Summary

The goals of this lab are as follows:

- Part A - Exceptions and exception handling
    - What are exceptions and why & how do we use them?
    - Refactor three previous tasks using exception handling
- Part B - An introduction to OOP with our own custom classes & objects
    - Study and enhance the Circle class

Part A - Exceptions and exception handling

In Lab07's demo and tasks, we saw that sometimes you're forced to declare that your method throws an exception. But why, and what does that mean?

Many of the problems we solve in class are pretty well-controlled and predictable, but the real world, and the data we get from it, is messy. Think about Assignment 5, where we ask the user to type in the name of the file. We might assume that the user never makes a typo, that the file always exists, and things like that, but if you were to make an application that is used in the real world – something that is useful, something that is reliable, and something that makes money – your code should be able to handle that.

Exceptions in Java are one of the many tools that resulted from this need. In essence, an **exception** is an error pointed out, or **thrown**, by some code that means the program cannot continue as intended. So far, most of the exceptions you've come across were put there by the developers of the Java language, but soon we'll be writing our own code to throw and to handle exceptions. **Handling exceptions** is a technique used when we anticipate some possible errors and want to account for that.

Let's see an example of exceptions and exception handling. We all know calculating your age in dog years is doable simply by multiplying by seven. Say we write some code that prompts a user for their age and tells them their age in dog years.

```
public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      System.out.print("Enter your age: ");
      int input = sc.nextInt();
```

```
	if (input < 0) {
		System.out.println("You cannot input a negative age");
	} else {
		System.out.println("Your age in dog years is " + ageInDogYears(input);
	}
}
public static int ageInDogYears(int humanYears) {
	return humanYears * 7
}
```

However, we don't want to let them enter a negative age. Normally, we'd do something like the above to deal with this. But what if someone else on the development team (or you, at 3am, two months from now) forgets and accidentally writes some code that calls our `ageInDogYears` method with a negative number? Can we prevent this from being possible? Yes, you might think, we could just do this:

```
public static int ageInDogYears2(int humanYears) {
	if (humanYears < 0) {
		return -1;
	} else {
		return humanYears * 7;
	}
}
```

But this didn't really *prevent* anything, did it? You can still call the method and get a result, and if you don't know how the code works (either because you didn't write it and made assumptions, or you did but it's been months and you're sleep-deprived) you might not even notice what went wrong – or more importantly, where it went wrong. How can we really prevent this from happening? Enter exceptions.

```
public static int ageInDogYears3(int humanYears) /*throws IllegalArgumentException*/ {
	if (humanYears < 0) {
		throw new IllegalArgumentException("Cannot input a negative age");
	} else {
		return humanYears * 7;
	}
}
```

(If you're curious why throws IllegalArgumentException isn't required in this method header, whereas something like throws FileNotFoundException is required: it's due to IllegalArgumentException being an unchecked exception. This StackOverflow answer explains the concept well, but you haven't been taught everything you need to fully understand it yet.)

Now if `ageInDogYears3` is called with a negative number, the program will crash with an `IllegalArgumentException`. But this isn't very desirable, either. We want a way for the program to keep going in case it runs into predictable issues like this. So let's put it all together using exception handling:

```
public static void main(String[] args) {
	Scanner sc = new Scanner(System.in);
	System.out.print("Enter your age: ");
	int input = sc.nextInt();
	try {
		System.out.println(ageInDogYears3(input));
	} catch (IllegalArgumentException e) {
		System.out.println("Error: " + e.getMessage());
		//e.printStackTrace(); // good for development purposes, but the
user doesn't care to see this
	}
}
```

These are pretty contrived examples, but the same techniques apply for using and handling exceptions in more serious code, too. A few of your tasks for this lab include following a similar process for some past work you've done to get used to this process.
Task 1: Refactor your solution to Lab02 Task C. It should have:

- A `main` method that collects user input, calls a method `getMonthName`, and prints the result. The method call should be wrapped in a try/catch block, which tells the user if their input was invalid, and why it was invalid, and does not crash the program.
- A method with the header

`public static String getMonthNumber(int monthNumber) throws IllegalArgumentException`
which has an array or ArrayList of the month names as strings, and returns the appropriate month name. For example, `getMonthNumber(1)` should return "January" and `getMonthNumber(12)` should return "December". The method should throw an `IllegalArgumentException` if the monthNumber is not valid.

Task 2: Refactor your Lab04 Task 3. It should have:

- A `main` method that collects user input, calls a method `getRectangleString`, and prints the result. The method call should be wrapped in a try/catch block, which tells the user if their input was invalid, and why it was invalid, and does not crash the program.
- A method with the header

`public static String getRectangleString(int rows, int cols, char symbol) throws IllegalArgumentException` which, using any loop you'd like, returns the rectangle with the appropriate dimensions made of the user-chosen `symbol`. The method should throw an `IllegalArgumentException` if `rows` or `cols` are less than 1.

Task 3: Refactor your Lab07 Task 3. When the user inputs a file name, the program should tell them if the file could not be found, and repeatedly ask them for the file name until they input a valid file name. This way, the program should not crash, and the main method header should no longer need to specify throws `FileNotFoundException`. Accomplish this using a try/catch block.

Part B - An introduction to OOP with our own custom classes and objects

Object-oriented programming is arguably the most common of the well-known **programming paradigms**. Another includes **functional programming**, small elements of which you've seen either in C211 with Racket (a multi-paradigm language) or C200 with Python's lambda syntax. One of the overall purposes of the class is to learn object-oriented programming, but so far, we've worked almost exclusively within very fundamental imperative programming: notions of loops, variables, conditionals, etc. that date back to languages like C and beyond. OOP falls under imperative programming, but allows us to code with much more flexibility and power through concepts like abstraction, encapsulation, inheritance, and polymorphism. It sounds like a lot – and it is – but this is the focus of the rest of the course, and we'll take things step by step.

Java is intrinsically built on OOP concepts in many ways. For one, you may have noticed that all the code you write is within some class. We've technically been making our own classes since the first week, but we've never actually made objects of those classes.

What is a circle? Mathematical definition aside, we know we can represent a circle with a few properties that belong to that circle; for now, let's say, its radius and its color. Thus, with some template of a circle that includes a "blank" for its radius and color to fill in, we can create as many circles as we want, which are each different from each other but have a common structure linking them. This means that if I tell you I'm going to give you a circle, you know what to expect that circle to have, no matter what. We also might want to give each circle some behaviors in common. We'll now see these concepts translated into code.

Here is our Circle class. Notice it has:

- Two private instance variables, `radius` (of type double) and `color` (of type String).
- Two overloaded constructors.
- Two public methods: `getRadius()` and `getArea()`.

```
public class Circle { // saved in the file "Circle.java"
//private instance variable, meaning: not accessible from outside this class
  private double radius;
  private String color;

// 1st constructor, which sets both radius and color to some defaults
  public Circle() {
     radius = 1.0;
     color = "red";
  }

// 2nd constructor with given radius, but color default
public Circle(double r) {
     radius = r;
     color = "red";
  }

// A public method for retrieving the radius
  public double getRadius() {
    return radius;
  }

// A public method for computing the area of circle
  public double getArea() {
     return radius*radius*Math.PI;
  }
}
```

Can you compile and run Circle.java? In IntelliJ, you'll notice that this code has no green Run button. This is because it has no main method, and therefore the class cannot be run. We can correctly guess, then, that `Circle` is meant to be used as a component of another program.

Let's write that other program, `TestCircle`.

```
public class TestCircle { // saved in the file "TestCircle.java"
  public static void main(String[] args) {
    // Declare and allocate an instance of class Circle called c1 with
    // default radius and color
    Circle c1 = new Circle();
    // Use the dot operator to invoke methods of instance c1
    System.out.println("The circle has radius of " + c1.getRadius()
        + " and area of " + c1.getArea());
```

```
    // Declare and allocate an instance of class Circle called c2 with
    // specified radius
    Circle c2 = new Circle(2.0);
    System.out.println("The circle has radius of " + c1.getRadius()
        + " and area of " + c1.getArea());
  }
}
```

Task 4: Your remaining task for this lab is to study and enhance this class.

1. New Constructor: Modify Circle to include a third overloaded constructor for constructing a Circle object with both radius and color.
2. New Getter: Add a getter method `getColor` for retrieving the color of a Circle instance.
3. Study public vs private: In TestCircle, can you access the instance variable radius directly (e.g., `System.out.println(c1.radius)`) or assign a new value to radius (e.g., `c1.radius=5.0`)? Try it out, then comment out the code you used to try it and explain the error messages in a comment somewhere in the TestCircle file.
4. Add Setters: There may be a need to change the values of radius and color of a Circle instance after it is constructed. Since its instance variables are private and no code in Circle modifies them, this is currently not possible. Add two public methods called setters for changing the radius and color of a Circle instance as follows, and modify TestCircle to test these methods and see the change occur:

```
public void setRadius(double r) {
    radius = r;
}

public void setColor(String c) {...}
```

There are plenty more best practice concepts to cover with OOP design considerations in Java, such as the "this" keyword and overriding the toString() method, which we haven't reached yet, but this is a good start.