Lab1b: 161 Review and New
CSSSKL 162: Programming Methodology

Summary

This lab both reviews old materials and introduce new materials that correspond with the readings in the schedule for weeks one and two.

Introduction

In this next section, we'll use a driver provided for you to explore each of the sections below. In BlueJ or Eclipse, make a new Java project and add the "NewAndReviewExamples.java" driver file to get started. Also used in this lab is the Rectangle.java class file; download both of these and read the comments inside.

Data & Methods

You may complete this section in lab if time permits, but this section is designed to be completed outside of lab and before lecture next week. In this section, you will review some old topics and explore some new ones as well. We'll start with topics related to **data** (primitives and objects), and then look at topics that focus on **methods**. Topics in blue are review, and topics in orange are new and should be approached with the book in hand. Download the NewAndReviewExamples.java driver file and Rectangle.java class to get started.

Data Section (Review & New)

Data in Java: Primitives and Objects

In this section, we'll build a class that is composed of both primitives and objects. Java has classes to help in converting from primitives to objects (for example an int to an **Integer**) and vice-versa. Let's start by building a small class used to represent a single concept such as a **Car** or **Vehicle**.

- Make a new Java project and and create a new class called "**Car**".
- Cars should define primitives for things like odometers, etc., and Strings for make and model.
    - Inside your Car class but outside of any method, define three (instance) variables for the odometer, make, and model.
- Write a main that builds 2 cars and prints them out:
    - "Car c1 = new Car();
    - "System.out.println(c1.toString());"

Variable Scope in Java: Local and Class-Level

Lets practice building classes again and defining two variables: one local and one with class-level scope. Prove to yourself that you can access the class-level variable throughout the class in which it's defined. Then, try to access the local variable from a method other than the one in which it's defined. Finally, look inside of Rectangle.java and identify at least 4 instance variables (class-level) and at least 2 local variables. Indicate these using comments.

This (the Implicit Parameter)

Take a class you've already built (or build the Car Class described in the "Data in Java" section) and build an object from that class. Observe the address of that object in main by using println with toString(). Next, from a method inside the class, print out the address of the "this" object using println. Call that method on the object you've just built, and explain why the two addresses are the same.

Access Modifiers: Public and Private

Build yet another simple class (say, a Point or Pair). Then, create another (separate, distinct) class that is to be the "driver" for this example (just like the driver above). In your driver, build an object of the Point class and try to access a method declared as public. Now, on the same vehicle object, try to call a method declared as private. What message does Java print out? Next, declare some class-level data item as public (an int, say), and declare another class-level data item as private. In your driver's "main", try again to access these two data items – what message does the Java compiler display now?

Methods Section (Review & New)

Accessors and Mutators (or, Getters and Setters)

Each of these sections encourages you to practice building small classes, and we'll continue that pattern here. Construct a simple class used to store the time or date. Add a field for minute, second, and hour, but make these class-level variables private. So that our class can be used by external clients, we need to declare some public methods for use with our private data. Build two methods for each data item: one to get the value of the data item, and one to set the value of the data item. See the getters and setters defined in the Rectangle class for examples of these getters & setters.

Overriding toString() (and equals())

Build a simple class to represent a Vehicle (or reuse your Car class above). In a main, build an object of that class, and print out the object using System.out.println(). Notice that this simply reports the memory address of the object in question, and we'd like to do something more useful. To replace (or override) the toString (or equals) function, first see the examples defined in the NewAndReviewExamples.java file for both toString and equals. Now, build a toString function that prints out the make, model, and odometer reading for a vehicle object.

Overloading Methods

Check out the Rectangle class constructors to see an example of overloading – defining multiple methods with the same name. Now build a SquareSomething class, with all static functions, whose purpose is to take an int, a double, or a float, and report back the square of the number (returning the correct type). This will mean your square class has three functions (all named "square") that each take a different type of data as input (int, double, float). Your square function that takes a double should return a double as well. This is how the println() method accomplishes such flexibility – you can hand it an int, a string, a double, a float, etc. and it simply prints the data. How it does so is by overloading the println() method to provide a different function for each possible type of input.

Constructors as Methods

First, check out the set of constructors provided for you in the Rectangle class. Notice how they are overloaded (meaning many methods with the same name, but different with regards to input) to provide flexibility for users of this class. Add to your Car class two constructors – one to take a string make, the other to take two strings: a make and a model. Test these constructors by building multiple Cars in your main() driver, calling each constructor in turn.