Lecture Set #6: Encapsulation, “this”, junit testing and Libraries

1. Review of Parameter passing
2. this
3. public vs. private Choices
4. junit testing
5. Libraries

Tracing Parameters and Methods

- Recall that methods / constructors can have parameters
  ```java
  public static int funnyAdd(int first, int second) {
    int localVal = 8;
    localVal += first++ + ++second;
    System.out.println(first + " + " + second);
    return localVal;
  }
  ```
- What is printed by the following?
  ```java
  public static void main(String[] args) {
    int x = 3, y = 2;
    int sum = funnyAdd(x, y);
    System.out.println("sum = ", sum);
    int sum = funnyAdd(x - y, y + 1);
    System.out.println("sum = ", sum);
  }
  ```
Parameters and Constructors

- Recall that methods / constructors can have parameters
  ```java
  public Student(String newName, int IDDesired) {
      name = newName;
      id = IDDesired;
      tokenLevel = 3;
  }
  ```
- What is printed by the following?
  ```java
  String newName = "Joe";
  Student s = new Student(newName + " Schmoe", 123456789);
  System.out.println(s.name);
  System.out.println(newName);
  ```
  - Joe Schmoe
  - Joe
  
How Does Java Evaluate Method / Constructor Calls?

- Arguments are evaluated using stack in effect at call site (place where method called)
  - `newName + " Schmoe", evaluates to Joe Schmoe`
  - `123456789` evaluates to `123456789`
- Stack frame (temporary addition to stack) created to associate method parameters with values
- Stack frame put into stack
- Body of method executed in modified stack
- Stack frame removed from stack
**this**

- a reference to the current object. (Only makes sense in a non-static method.)
- In an instance method, this is the object that is assumed
  - easy to refer to members (data or methods) using the assumed object
  - difficult to refer to the whole object without having a name to call it
- Only use when needed – using it all the time makes the code more difficult to read

**Public Declarations**

- **public** variables/methods and classes
  - Keyword `public` used in declaration
  - Every user of an object can access any `public` element
- Sometimes access should be restricted!
  - To avoid giving object users unnecessary info (keep API small)
  - To enforce consistency on instance variables
Private Declarations

- **private** variables, methods and classes
  ```java
dprivate int tokenLevel = 3;
```
- Private variables / members cannot be accessed outside the class definition
- Declaring instance variables private means they can only be modified using public methods

What Should Be Public / Private?

- **Class interface** = API = public variables / methods
- Only make something public if there is a reason to
- Why? **Encapsulation**
  - As long as interface is preserved, class can change without breaking other code
  - The more limited the interface, the less there is to maintain
- Rule of thumb
  - Make instance variables private
  - Implement **set** / **get** methods
  - Make auxiliary methods private
Separate:
API and the workings of the class

- Design so that
  - you can change how the class works without having to change the API
  - the only things in the API are things the user will absolutely need (make the interface as simple as possible)
- Demonstrations in Class
  - Significantly Modifying the Student class – without changing the API (or the driver)
  - The Cat class and its drivers
    - with adding a copy constructor
  - Project 3
    - API described – you are using those classes
    - documentation / comments needed

Floating Point Calculations

What will this print?

```java
public class SimpleMath {
    public static void main(String[] args) {
        if (3.9 - 3.8 == 0.1) {
            System.out.println("I am a very smart computer.");
        } else {
            System.out.println("I can't do simple arithmetic.");
        }
    }
}
```

→ I can't do simple arithmetic.
  - Why?
  - Conversion of floating point to binary leads to precision errors!
  - What can we do?
Floating Point Calculations (cont.)

Two important rules:

- You can never use `==` to compare floating point values. Instead, check if two numbers are within a certain tolerance of each other.
- Never use floating point values to represent money, e.g., 3.52 to represent $3.52. Instead, use integer 352 to represent 352 pennies.

The problem

- Problems:
  - need to be able to make sure all parts are tested
  - need to know in testing exactly which part was not as expected
  - need to be able to keep the tests for modifications made later
- **Unit testing** helps overcome this problems of making sure everything is tested
  - Unit testing: test each class and each part of the class (unit) individually
  - Goal is to eliminate inconsistencies between the API and the actual working of the code
Unit Testing

- **Unit testing** helps overcome this problems of making sure everything is tested in a structured way
  - Unit testing: test each unit individually (micro level – each method or specifically each interaction described in the API)
  - Goal is to eliminate errors within classes
- **Needs for unit testing**
  - Method for defining tests = inputs, expected outputs
  - Method for running tests
  - Method for reporting results
- **One possibility:** write a driver for each class
  - Driver class contains main method
  - main method creates objects in class to be tested, calls methods, prints outputs
  - User checks outputs, determines correctness
  - Good: easy, no special tools needed
  - Bad: tedious, relies on human inspection of outputs
- **Another approach:** [JUnit](#)

JUnit

- A unit-testing tool for Java
- Includes capabilities for:
  - Test definition, including output checking
  - Test running (execution)
  - Result reporting
- Seamless integration with Eclipse
- **Note**
  - In this class we will use JUnit 3.8.1
  - So, when given a choice select [JUnit 3](#)
Structure of a JUnit 3.8.1 Test Case

import junit.framework.TestCase;

public class FunnyIntegerSetTest01 extends TestCase {

    public void testInsert() {
        FunnyIntegerSet set = new FunnyIntegerSet();
        set.insert(3);
        assertTrue(set != null);
    }

    public void testFindClosest() {
        FunnyIntegerSet set = new FunnyIntegerSet();
        set.insert(3);
        set.insert(6);
        assertEquals(6, set.findClosest(5));
    }
}

A Test Case Is … A Class!

- assertion checkers
  - assertTrue(expression);
    - If statement is true, keep running test; otherwise, halt test, report “fail”
  - assertFalse(expression);
    - If statement is false, keep running test; otherwise, halt test, report “fail”
  - assertEquals(expression1, expression2);
    - If expression1, expression2 equal, keep running test; otherwise, halt test, report “fail”
  - If test terminates without failing an assertion and without throwing an uncaught exception, then it passes that test
- It continues with all subsequent tests regardless of passing or failing the current test
Hints on Testing

- Give names to tests that relate to class being tested
- Develop some tests before you code
  - Helps you to clarify what you are supposed to be doing
  - Gives you some ready-made tests to run while you code
- Use tests to debug
- How many tests?
  - **Statement coverage**: develop tests to make sure each statement in class is executed at least once (including constructors)
  - **Decision coverage**: develop tests to make each condition (if statement) in program both true and false
  - You should at least reach statement coverage in your own testing

Taking Care of Corner Cases

- **FunnyIntegerSet example from CVS**
  - Set of null was a corner case that we needed to test for
  - Write new test cases or new asserts in the test cases that already exist to take care of this

```java
public void testNullSet(){
    FunnyIntegerSet s = null;
    s.insert(4);
    assertEquals(s.findClosest(3),4);
}
```
Documentation Types

- Three Styles
  - `// ...`
  - `/* ... */`
  - `/** ... */`

- Two Purposes
  - Internal – those reading code
  - External – those using the class

Javadoc Documentation Standard

- When documenting a method, list exceptions that method can throw
  - Use `@exception` tag
  - Be sure to include unhandled exceptions that operations in method may throw

- Example:
  ```java
  /**
   * Returns the year part of a date string
   * @param d date string in mm/dd/yyyy format
   * @return an integer representing the date
   * @exception IndexOutOfBoundsException
   * @exception NumberFormatException
   */
  public static int getYear(String d) {
      ...  
  }
  ```
Libraries in Java

- **Library**: implementation of useful routines that are shared by different programs
- **Java mechanism for creating libraries**: packages
  - **Package**: group of related classes
  - **Example**: `java.util` (contains `Scanner` class)
  - To use a class from a package, you can use a **fully qualified name** (package name + class name):
    ```java
    java.util.Scanner s = new java.util.Scanner(System.in);
    ```
  - You can also import the class in the beginning of the file
    ```java
    import java.util.Scanner;
    ```
  - To import class in a package:
    ```java
    import java.util.*;
    ```
    (Imports `Scanner` as well as other classes in package)

Package java.lang

- A special package containing widely used classes:
  - `String`
  - `Math`
  - etc.
  - `java.lang.*` is **automatically imported** by every Java program
Package Management

- A class can be added to a package by including:
  
  ```java
  package <name of package>;
  
  in source file (usually very first line)
  ```

- The variables / methods provided by a class / package are often called its **API** (= Application Programmers Interface)

- APIs should be documented

- java.lang documentation:
  

- On the resources page of the class web site – javadoc generated descriptions.

---

String API & Math API

- **String** implements lots of string functions
  
  - StringExample.java

- **Math** implements lots of mathematical functions
  
  - MathExample.java