Lab 2: Object-Oriented Design
12:00 PM, Jan 31, 2018

Contents

1 Terminology 1

2 Class Hierarchy Diagrams 2
   2.1 An Example: Animals ......................................... 2
   2.2 A “Class” Hierarchy ......................................... 3

3 Class Time! 4

4 Javadoc 5
   4.1 Javadoc Comments ......................................... 5
   4.2 Javadoc Tags .................................................. 5

5 Uh-Oh! Fixing a Broken Hierarchy 6

Objectives

By the end of this lab, you will know:

- how much fun object-oriented programming can be
- how to design class hierarchies

By the end of this lab, you will be able to:

- write simple object-oriented programs in Java
- use Javadocs

1 Terminology

In this lab, you’ll learn how to successfully design object-oriented (OO) programs. Good OO design dictates the organization of large and complex programs into small and simple components called classes. If all classes are independently tested and refined, so we are certain they have been designed and implemented correctly, then they can be used with confidence in combination with each other.

Here are some important object-oriented terminology and concepts:

Key Concepts:
- **Object**: The foundation on which object-oriented programming is built. The data and functionality of objects are the building blocks of object-oriented programs.

- **Inheritance**: When one class inherits data and functionality from another.

- **Class Hierarchy**: The organization of classes within a program; dictates inheritance relations.

- **Encapsulation**: The grouping of related data and functionality together within a class.

- **Information Hiding**: Ignoring implementation details (i.e., data representations), and needing only to understand code at the level of its functionality.

- **Subtype Polymorphism**: The power to treat subtypes that inherit from the same class or implement a common interface similarly.

**Types of classes:**

- **Class**: A template (or blue print) which describes attributes and behaviors, and is used to create objects. Cannot have unimplemented methods.

- **Abstract Class**: A class that cannot itself be instantiated, but whose data and functionality can be inherited by other subclasses. Can have fields and both implemented and unimplemented methods.

- **Interface**: A class that cannot be instantiated, containing a set of method declarations which any subclass must inherit and any concrete subclass must implement. Can only have constant (not changing) fields and unimplemented methods.

**Features of classes:**

- **Attribute/Field**: A value associated with an object.

- **Behavior/Functionality**: A method (or several methods) associated with an object that can make use of its attributes.

- **Superclass/supertype**: A class which is inherited from or implemented by other class(es), known as its subclass(es)/subtype(s).

- **Subclass/subtype**: A class which inherits from or implements other class(es), known as its superclass(es)/supertype(s).

2 **Class Hierarchy Diagrams**

Given a class hierarchy, we can create a **class hierarchy diagram** to visually represent the shared data and functionality of related classes.
2.1 An Example: Animals

Suppose we want to represent a set of different animals that are all, in one way or another, similar. For example, consider simulating a collection of mammals. Some mammals behave similarly, while others don’t. Some are pets, while others live in the wild. They all have some commonalities but also behaviors and traits that are unique to their species.

We will build a class hierarchy to represent mammals. This hierarchy will abstract out commonalities among subclasses into superclasses.

Let’s start by abstracting out what is common to all mammals. At their core all mammals have some similar behaviors. They all eat, drink, and sleep. It makes sense to abstract out this shared functionality into an interface, `MammalControls`.

Then, as mentioned previously, mammals split into two groups – those that are pets and those that live in the wild. Mammals that are pets have commonalities – for example, most pets have names and often seek the attention of their owners. Mammals that are wild, on the other hand, do not necessarily have names and don’t have owners. Instead, they share a different set of traits. For example, they have habitats and have to find their own food. It makes sense then, to create two abstract classes, one for pets and one for wild animals. Since both pets and wild animals are mammals they should both extend the `MammalControls` interface.

After we have the abstract classes `Pets` and `WildAnimals`, we can create classes for each specific type of animal. For example, we could create a Dog or a Cat class that extends the Pets abstract class. We could create Bear and Squirrel classes that extend the WildAnimals class. In this way, we can represent mammals in a class hierarchy.

We can see this class diagram in the figure above.
2.2 A “Class” Hierarchy

Here at Brown, and within this very class, we have three entities: Professors, TAs, and Students, each of which has different roles and responsibilities. With your partner, discuss the roles and responsibilities of these three Brown entities.

Do Professors, TAs, and Students have anything in common in terms of data and/or functionality? What types of data does Brown need to maintain about each (e.g. name, Banner ID, etc.)? And what sort of functionality do the various entities partake in? Can their commonalities be abstracted out into superclasses or interfaces (such as an Instructor interface)?

**Task:** Design a class diagram that relates Professors, TAs, and Students. Draw your solution, and for each class describe the data and functionality it contains.

**Hint:** Your diagram need not be a tree!

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

### 3 Class Time!

Now that you have learned how to visualize class hierarchy, it’s time to get to coding! Read the following specifications, and design and implement classes that reflect our requirements. Try thinking in an OOP-oriented way!

- All people should be able to eat; this method should take in a String representing which dining hall is having a special that day, and return the String location of where the person chooses to eat. Students eat at the dining hall with the special; Professors eat Kabob and Curry. TAs eat leftovers at the CIT unless the special is “V-Dub”, in which case they eat at the V-Dub (chicken finger Friday is universally loved).

- All students at Brown (not just students in 18, but the TAs as well!) can go to CS 18 lecture. When a TA attends class, a message “A TA is reviewing CS 18 material!” should print. When a CS 18 Student attends class, a message “<student name> is attending CS 18!” should print.

- All CS 18 students have a preferred name and an int grade for lab 1.

- All TAs have a number of semesters they’ve TAd so far – it determines pay[^1]

- All instructors (TAs and Professors) should have a method that takes a number of hours worked and returns the dollar amount they should be paid. This depends on the number of semesters TAd so far for TAs. Suppose a TA works for h hours and has TAd n semesters before. Then, their pay should be \( h \times (10 + n) \), increasing with each subsequent semester[^2]. Meanwhile, Professors would be paid \( h \times 20 \) regardless of how long they have worked[^3] (in fact, Professors should not have a field denoting how long theyve worked!)

- A Professor has a boolean indicating if they have tenure or not.

[^1]: This is true
[^2]: This is not actually how TAs get paid; it’s been simplified for this problem
[^3]: This is also not how professors get paid; it too has been simplified for this problem
· A Professor should be able to, given a Student, calculate their grade. This method should take in a Student and return their only grade (the lab grade!) as a percentage. For example, a Student with a 4 out of 5 on Lab 1 would have a grade of 80.

· All instructors (TAs and Professors) should be able to, given a Student, call out their preferred name at hours. **Hint:** Use printlns!

**Task:** Implement classes to achieve the above functionality.

**Hint:** When implementing the eat method for TAs, and checking to see if the special is “V-Dub”, you will not be able to use == like you can with ints and doubles! Instead, you should use the equals method. For example, `myString.equals("hi")` returns true if `myString` is “hi” and false otherwise. You’ll learn more about this soon!

**Hint:** If you decide a different class structure than what you originally had would work better, you’re more than welcome to change your class structure accordingly!

---

| You’ve reached a checkpoint! Please call over a lab TA to review your work.

---

### 4 Javadoc

*Javadoc* is a system that generates documents in the form of web pages from annotated Java code. That is, if you comment your Java code using in Javadoc annotations, then you can use the Javadoc system to automatically generate web pages that document your program.[4] In CS 18, you will not be required to use Javadoc for document generation, but you will be required to write your comments using standard Javadoc annotations (so that you will be ready for your first internship!).

#### 4.1 Javadoc Comments

As you know, the Java compiler recognizes comments delimited by // or /* ... */. Javadoc recognizes comments delimited like this: /** ... */. These comments are placed above the class, variable, or method about which you are writing a comment.

#### 4.2 Javadoc Tags

The keywords that are included in Javadoc annotations are called *tags*. Tags inform the Javadoc system of the type of information a comment contains. Below we list a few of the tags of interest in CS 18. To learn more about built-in tags, and how to make your own, check out the [Javadoc Documentation](#).

- **@author:** Identifies the author of the class or interface.
- **@version:** Provides version information.
- **@param:** Documents a specific parameter (argument) of a method or class constructor.

---

[4]Isn't that cool?!
In CS 18, you should have an @param tag for every parameter in every method you write. Further, you should have an @return tag for every method you write. When applicable, you should have an @throws tag for every exception thrown by every method you write.

**Note:** You do not need to add javadocs comments for methods that are overriding already declared methods. For example, you should comment all your methods in abstract classes or interfaces but when implementing them in subclasses, you do not need to comment.

**Warning:** You should never use the @author tag in your CS 18 assignments. We anonymize your assignments so we can grade as bias-free as possible, but sometimes our scripts fail to remove logins or names from handins. We would really appreciate you working with us to reduce the chance of student identity being revealed while we try to grade anonymously.

Now, let’s put together everything you’ve learned about Javadocs! The following Javadoc comments describe a class representing a hamster, a variable representing their name, and a method used to calculate their age in hamster years:

```java
/**
 * Class representing a hamster, which is a subclass of the Pet superclass.
 */
public class Hamster {

    /**
     * field representing the hamster's name
     */
    public String name;

    /**
     * A constructor for a Hamster
     * @param name - a String representing the Hamster's name
     */
    public Hamster(String name) {
        this.name = name;
    }

    /**
     * Method to calculate the Hamster's `hamster age'.
     * @param humanAge - the human age of the hamster.
     * @throws InvalidArgumentException when a number below 0 is entered.
     * @return an int representing the hamster age of the hamster.
     */
    public int hamsterAge(int humanAge) throws InvalidArgumentException {
        if (humanAge < 0)
            throw new InvalidArgumentException();

        int toReturn;
        // do calculations to toReturn
        return toReturn;
    }
}
Task: Fully fill in the required Javadoc comments to your classes.

Hint: In Eclipse, if you type /** followed by enter (above an existing class, variable, or method declaration), Eclipse will auto-generate some of the tags for you!

5 Uh-Oh! Fixing a Broken Hierarchy

Your friend has heard you’ve just become an expert in class hierarchy! They’ve come to you with a messy program, seeking your advice. Help them fix it!

Following is your friend’s code, representing various vehicles:

```java
class Vehicle {
    int seats;
    int mileage;
    int baseHourlyRate;

    int rentalCost(int numHours) {
        return numHours * this.baseHourlyRate;
    }
}

class Luxury {
    int premiumPrice;
}

class Rickshaw extends Vehicle {
    Rickshaw(int seats, int mileage) {
        super(seats, mileage);
    }
}

class SchoolBus extends Vehicle {
    String color;

    boolean needsOilChange() {
        return (this.mileage > 6000);
    }
}

class Limo extends Vehicle, Luxury {
    boolean hasDiscoLights;

    // luxury vehicles incur a higher hourly rate, raised by premium rate
    int hourlyCost() {
        return this.premiumPrice + this.baseHourlyRate;
    }

    boolean needsOilChange() {
        return (this.mileage > 3000);
    }
```
class Unicycle extends Vehicle {
    Unicycle(int seats, int mileage) {
        super(1, mileage);
    }

    boolean needsOilChange() {
        return false;
    }
}

Note: Clearly, this student has not taken CS 18– they’re missing all their Javadoc comments!

Task: First, construct a class diagram (draw it out on paper) corresponding to the existing code, without changing anything yet.

Task: Brainstorm how you can better organize this program, and then draw your new and improved class diagram on another sheet of paper. Be ready to explain why you chose to make the changes that you did!

Hint: Look for duplicate field and method declarations!

Once a lab TA signs off on your work, you’ve finished the lab! Congratulations! Before you leave, make sure both partners have access to the code you’ve just written.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs018/feedback