Lab 2: Object-Oriented Design
12:00 PM, Jan 29, 2020

Contents

1 The Mystery Begins... 1
2 Terminology 1
3 Class Hierarchy Diagrams 2
   3.1 An Example: TAs 3
   3.2 A Class “Tree” 3
4 Class Time! 4
5 Javadoc 4
   5.1 Javadoc Comments 4
   5.2 Javadoc Tags 5

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 The Mystery Begins...

CS 18 TA Deniz walks into the CIT at the crack of dawn, ready to work on a project. He’s really excited to use his favorite department machine, poundcake. He runs up the stairs into CIT 201 but to his horror, poundcake is gone! All that’s left is a monitor and a plate with an actual slice of poundcake. Shocked, he grabs the plate and heads to the one person that can unravel this mystery: you, the resident CIT detective.
Deniz bursts into your office, cradling the poundcake in his arms. He’s so upset that he instinctively shoves a bite of the poundcake in his mouth and freezes. This wasn’t just any poundcake — he immediately recognizes it as the poundcake that was sitting in the TA lab last night. You both realize that the culprit must have been a TA, since only TAs can access the lab. But who?

In order to narrow down the search for the thief, we can use Object Oriented Programming (OOP) as a way of categorizing groups of TAs. Some are UTAs, some are HTAs. They all have some similarities but also behaviors and traits that are unique to their group.

2 Terminology

In this lab, you’ll learn how to successfully design object-oriented programs. Good OOP 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:

- **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).

3 Class Hierarchy Diagrams

Given a class hierarchy, we can create a **class hierarchy diagram**, also called a **class diagram** or a **class tree**, to visually represent the shared data and functionality of related classes.

3.1 An Example: TAs

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

Let’s start by abstracting out what is common to all TAs. All TAs have some similar behaviors. They all eat, code, and sleep. It makes sense to abstract out this shared functionality into an interface, *ITA*.

Then, TAs split into two groups — UTAs and HTAs. UTAs tend to use SSH, and not care about what department machine they’re using. On the other hand, HTAs tend to have favorite department machines. It therefore makes sense to create two abstract classes, one for UTAs and one for HTAs. Since both UTAs and HTAs are TAs they should both extend the *ITA* interface.

After we have the abstract classes *UTA* and *HTA*, we can create classes for each specific TA. For example, we could create a Deniz or a Jefferson class that extends the UTA abstract class.
could create Cody and Carrie classes that extend the HTA class. In this way, we can represent TAs in a class hierarchy.

We can see this class diagram in the figure above.

### 3.2 A Class “Tree”

Remember BSTs (binary search trees)? Well, get ready, because it’s time to revisit them!

As a refresher, a binary tree is a tree where each node has up to two children. A tree is either a node, which has data and 2 children (which are also trees), or a leaf, which for this lab has neither data nor children.

A binary search tree is a tree where for every node \( p \), all the values in \( p \)'s left subtree are strictly less than the value of \( p \), and all the value in \( p \)'s right subtree are greater than or equal to the value of \( p \).

The condition is also satisfied if either or both child nodes are leaf nodes, so long as any non-leaf children still meet the requirements.

Note that this is *not* a balanced tree.

We also want to be able to perform some operations on binary search trees. Namely:

1. `insert(int n)`, a function that inserts a value into the BST and returns the resulting BST.
2. `contains(int n)`, a function that returns true if the given value is found in the BST, and false otherwise.

**Task:** We want you to design this using your newfound OOP knowledge! First, draw a class diagram on paper to represent a BST; specifically, represent a BST where each value is an int. Don’t start coding yet!

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

### 4 Class Time!

Now that you have learned how to visualize class hierarchy, it’s time to get to coding!

**Task:** Implement classes to achieve the functionality described in the previous section.

**Note:** You should name your interface `IBST` and your node and leaf classes `Node` and `Leaf`, respectively. Your `Node` class should hold its value in a field called `value` and its left and right subtrees in fields called `left` and `right`, respectively. Finally, your `Node` constructor should take three arguments: `value`, `left`, and `right`, in that order.

**Task:** Now that you’re all done coding your BST, why were there no abstract classes? Be prepared to explain your rationale to a TA.

**Task:** Test out your implementation using the tester like in previous labs!

You’ve reached a checkpoint! Please call over a lab TA to review your work.
5 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 Javadoc annotations, then you can use the Javadoc system to automatically generate web pages that document your program. 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!).

5.1 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.

5.2 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.
- **@return**: Documents the return value of a method.
- **@throws**: Documents the exceptions (errors) the method may throw, and under what conditions. You will likely not use this until later in the semester.

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 do our best to keep all grading anonymous, but if you put your name in your code, we will know who you are (and will deduct points also).

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:
/**
 * 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!

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

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: https://cs.brown.edu/courses/cs018/feedback