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 “Tree” 3

3 Class Time! 4

4 Javadoc 4
   4.1 Javadoc Comments 4
   4.2 Javadoc Tags 4

5 isBST() and toList() 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**, also called a **class diagram** or a **class tree**, 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. Some animals 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 animals. This hierarchy will abstract out commonalities among subclasses into superclasses.

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

Then, as mentioned previously, animals 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 animals they should both extend the `IAnimal` 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 animals in a class hierarchy.
We can see this class diagram in the figure above.

2.2 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 \textit{not} a balanced tree.

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

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

\textbf{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!

\textbf{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.

\textbf{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.

3 Class Time!

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

\textbf{Task:} Implement classes to achieve the functionality described in the previous section.

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

\textbf{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.

\textbf{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.
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 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!).

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.
- @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 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:
/**
 * 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.

5 isBST() and toList()

At this point, you’ve completed enough of the lab to receive full credit. However, next week’s lab material will build on the work you did today, so you should try to get as much done as possible before then. We’ll be using something called Oracle Testing to write tests for a binary search tree implementation. In order to do this, you will need two more methods in your BST: isBST() and toList().

The isBST method should return true if the tree is a valid binary search tree and false otherwise. As a reminder, a tree is a valid binary search tree if, for each node in the tree, all nodes in its left
subtree have values less than the node’s value and all nodes in its right subtree have values greater than or equal to the node’s value.

This task seems really large! Never fear: we’ll help you break it down. First, you’ll need a method that can determine if every value to the left of the root is less than the root value. However, the root is of type IBST, so you’ll need to add this method to the IBST interface! This means you’ll need to implement this for both Leaf and Node.

**Task:** Write a method `allLess`, which takes in an `Integer` and returns a boolean indicating if all the values in the `IBST` are less than that integer.

**Task:** Similarly, write a method `allGreaterEq`, which takes in an `Integer` and returns a boolean indicating if all the values in the `IBST` are greater than or equal to that integer.

**Hint:** You’ll need to add both of these methods to the `IBST` interface!

However, this is not enough! We need to check that for every node in the tree, (a) the value in the node is greater than every value in the left subtree, (b) the value in the node is less than or equal to every value in the right subtree, and (c) both the left and right subtrees are also BSTs.

**Task:** Write a new method, `isBST`, which checks that everything to the left of the `IBST` is less than the node’s data value, and everything to its right is greater than or equal to the node’s data value. Then, this method should ensure that both of the child nodes of the `IBST` are also BSTs.

**Hint:** Use the methods you just wrote, and some recursive calls!

**Hint:** Don’t forget to add this method to the interface as well!

**Task:** Write a `toList` method that takes in nothing and outputs a `List<Integer>` that is a list of all of the integers in the BST.

**Hint:** Here’s how you can make an empty list:

```java
List<Integer> myList = new LinkedList<Integer>();
```

**Hint:** Some helpful methods to look up are: `add()`, `addAll()`. Here’s how you can use them:

- `myList.add(5);` //adds one element to myList
- `myList.addAll(anotherList);` //adds all of the elements of a list to myList

Consult the documentation if you’d like more details.

**Hint:** It may be useful to use a helper function that can create a string representation of your list for testing purposes!

The `toList` method should return a list containing the elements contained in the BST. The order of these elements does not matter. However, duplicates do matter; each element should appear in the list the same number of times as it does in the tree.

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](http://cs.brown.edu/courses/cs018/feedback)