Learning Outcomes

Upon completing this assignment, students should be able to

- Extend a *binary search tree* container class to include *delete*, *size*, and *height* methods.
- Write `JUnit` tests for *add* and *delete* methods.

Introduction

Students will write a `OrderedShapes` program that reads a file full of `Shape` objects (class hierarchy below) and inserts them into a *binary search tree* according to their **area**. Then the user can enter commands to search for a shape with a given area, print out the whole collection in increasing order by area, print out the whole collection in decreasing order by area, or quit the program.

Note: This program includes grading based on using *test-driven development* (TDD). This requires you to write a *failing* test for a feature/method, commit the code with the broken test into `git`, make the code pass the test, and then commit the working code. This will be examined in terms of the *delete* and *height* methods in your tree class. You may use TDD for the size method if you wish, or not. the *right* way to do testing is to write tests for each feature as you go. This would be a good chance to practice this **but** you are not obligated to do this.

You can associate your local repo with your repo for turn-in on Gitea at any point. When you push the repo, the whole history of the project (all commits along with the date/time, commit message, and user who made the commit) is pushed. If you have problems pushing changed code, let Dr. Ladd know and he can help interpret `git` error messages.

Method

Getting Started

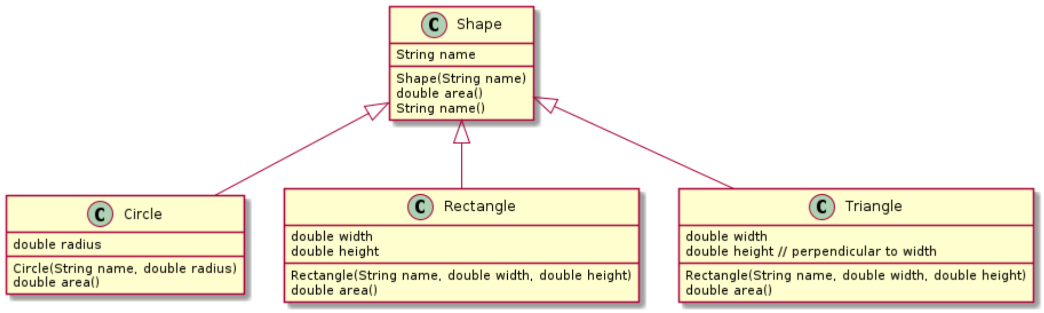
Start with your *working* `pTree-a` solution. This will give you

- a `Shape` class hierarchy with `Shape`, `Circle`, and `Rectangle`.
- a *menu-driven* client program that loads a binary-search tree with `Shapes` from a file.
- a `BST` class that can hold `Shape` objects.

You will be expected to turn this program in through the *same* Gitea repository you used for `pTree-a` (`F23-205-<ccid>-p0`

Class Hierarchy

You will create a new class, `shapes.Triangle`. All four `Shape` classes are in the new diagram with descriptions below.



`Shape` The ultimate parent of the hierarchy. Keeps track of the *name* of the shape (an identifier) and has an `area` method that returns zero.

Circle Keeps a *radius* and overrides area to return the area of the circle.
Rectangle Keeps *width* and *height*, overrides area to return the area of the rectangle.
Triangle Keeps *width* and *height*, overrides area to return the area of the triangle.

These are very short classes. They should have toString methods that returns the *type* of the shape, its *name*, and any specific information for the given type.

Binary Search Tree

You will modify your BST class to have the following public interface:

Binary Search Tree

Put the class BST in the package. The class interface (the public functions it should have) is:

```java
package tree;
class BST {
  public BST();
  public void add(Shape newbie);
  public Shape find(String nameToSearchFor);
  // true if found and deleted; false otherwise
  public boolean delete(String nameToDelete);
  // the list of Shapes, one per line
  public String inOrder();
  // the lint of Shapes, indented by tree depth, one per line
  public String inOrderIndented();
  // height of the tree (empty tree height 0)
  public int height();
  // size of the tree (number of nodes)
  public int size();
```

Testing

Before implementing delete, create a JUnit test class in the app/src/test/java/tree folder, BSTTest.java. (You may need to add back these directories.)

The important import statements for a test file like this are

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
```

Remember that Test is a Java *annotation* (it comes after an @ sign and labels a method as a test to be run).

Add a test to your test class that checks what happens when you delete from an empty tree:

```java
@Test
void DeleteFromNewTreeTest() {
  BST bst = new BST();
  assertFalse(bst.delete("Bet this isn't found."));
}
```

Try to compile. It will fail because there is no delete method yet. You have a failing test! Add and commit everything.

Make the test pass as simply as possible. That means your delete is going to be

```
public boolean delete(String nameToDelete) {
  return false;
}
```

Now your code should compile and the test pass. Add and commit. (This assignment will stop saying "Add and commit." after each *failing* test and each *passing* test; it is there implicitly.)

Next, add a new test that adds a shape to the tree and then deletes it. It is a new `@Test` method. This will fail because the call to `delete` should return `true` when it succeeds in removing the shape. (Okay, one more time: **Add and commit.**)

This means you need to make `delete` work. This is one of the easy cases.

Now, make a new test that creates a more complex tree. Check that you can successfully delete a node with no children, one with just a left child, one with just a right child, and one with two non-singleton tree children. This could be one or four test methods. Up to you.

You can use the indenting print function to get the structure of the tree, if you like, before and after the `delete`. Or you can use the search function to make sure only the expected node goes away.

Notice that you are, implicitly, testing `add`, too. You will test `add` explicitly now: write a test method for building a *degenerate* tree by inserting in sorted order. Use the indenting print method to make sure the structure is what you expect.

There is a problem: the test is supposed to initially fail. Why is that? so that you can be sure that you are testing the thing you mean to test. An *empty* test will pass without checking anything in your code. So: break your test code by changing the expected value. Now it will fail **and** you know that it is being run. Add and commit, fix, add and commit.

Also implement `size` and `height`. You are not required to use TDD for these methods.

Input/Output

`main.TreeClientProgram` expects a shape file name on the command-line. Only difference from previous the previous version is that the type might be `Triangle`; a `Triangle` has the same fields as a `Rectangle`.

The data file has the following format:

```
Shape
unspecified-shape-1
Circle
small-circle
1.5
Rectangle
square-B
12 12
Circle
big-circle
22.8
```

Notice that each shape begins with the name of the class on a line by itself. The next line is the name of the shape. A `Circle` then has a line with a radius on it and a `Rectangle` or `Triangle` has a line with the width and height.

Design Considerations

The tree must be kept ordered in ascending order by *name*: this is different from the previous project. The *search* method now prompts for a name rather than an area (as does `delete`).

Use the indented print method to make sure that the tree is constructed correctly. Play with the code to see if you can break it. If you do, you might want to write a test that breaks it the same way and then fix the code to make the test pass (without causing other tests to fail).

Documentation

README

Must document how you tested. You can point at your `JUnit` tests.

Must include instructions on how to **compile** and how to **run** the program as submitted.

Deliverables

Submission medium: `git` to Gitea at `cs-devel.potsdam.edu`.