Com S 227
Spring 2024
Assignment 4
300 points

Due Date: Friday, May 3, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm May 2)

NO LATE SUBMISSIONS! No extensions. All work must be in Friday night.

General Information

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, for details. You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Please start the assignment as soon as possible and get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

Read this specification carefully. It is not quite like reading a story from beginning to end, and you will probably have to reread some parts many times. See the section "If you have questions" if you find something unclear in this document.

Introduction

The purpose of this assignment is to give you some experience:

- using interfaces and abstract classes
- reusing code through inheritance ("is-a") relationship
- reusing code through association ("has-a") relationship

A portion of your grade on this assignment (roughly 15% to 20%) will be determined by your logical organization of classes in a class hierarchy and by how effectively you have been able to use inheritance and composition to minimize duplicated code. Be sure you have **done and understood Lab 8 checkpoint 2!**

In this project you will complete the implementation of a hierarchy of classes representing various types of elements in a video game. Your task is to implement these eight concrete classes:

`SimpleElement`
`AttachedElement`
`FlyingElement`
`FollowerElement`
`LiftElement`
`MovingElement`
`PlatformElement`
`VanishingElement`

All eight of these classes must directly or indirectly extend the given abstract class **`api.AbstractElement`**. In implementing them, you may put them into a class hierarchy that contains one or more additional abstract classes you create on your own to a) avoid duplicating common code between the eight types, and b) avoid having redundant code or attributes in any type.

All your code (the 8 required classes, and any extra classes you create) goes in the **`hw4`** package.

Using inheritance

Part of your grade for this assignment (e.g. 15-20%) will be based on how effectively you are able to implement the eight required concrete types with a minimum of code duplication. Here are some specific restrictions; see page 13 for more details.

- You may not use **`public`**, **`protected`** or package-private variables. (You can provide protected accessor methods or constructors when needed.)
- You may not modify the **`AbstractElement`** class itself (or anything else in the **`api`** package).

Overview

A 2D video game may include elements such as players, enemies, projectiles, PlatformElements, walls, and so on, each with slightly different behavior, but with many attributes in common as well. For example, at any given time, a game element has a *position* (x, y) within a window, where x and y are floating point values, (0, 0) is the upper left corner of the window, and the positive y- direction is *downwards*. It also has a width and height, forming a *bounding rectangle*. This rectangle is used for actually drawing the graphics on the screen, and also for detecting collisions between elements.

The capabilities needed for doing the actual drawing are represented by the simple interface **`api.Drawable`**, with these six methods:

`void draw(Graphics g)` – draws this object using the given graphics context
`int getXInt()` – returns the x-coordinate, rounded to the nearest integer

`int getYInt()` – returns the y-coordinate, rounded to the nearest integer|
`int getWidth()` – returns the width
`int getHeight()` – returns the height
`Rectangle getRect()` – returns the bounding rectangle (an instance of java.awt.Rectangle)

(These capabilities are isolated in this interface so that the graphical portion of applications need not be concerned with any of the details of our specific type hierarchy for game elements.)

Your starting point will be the abstract class **`api.AbstractElement`**, which is declared to implement the **`Drawable`** interface but leaves five of its methods abstract, and also declares seven additional abstract methods. The important point is that the **`AbstractElement`** class provides the implementation of the **`draw`** method, so that in your own code you do not ever need to be concerned with the graphical aspects of the application. To allow for customizing the way that a given game element is rendered on the screen, we define an interface **`api.Renderer`**, and a **`AbstractElement`** always has an attribute of type **`Renderer`**. The **`api`** package includes various examples of renderers that are used in the demo applications, such as the **`BasicRenderer`** that fills in an element's bounding rectangle with a solid color, or the **`ImageRenderer`** that displays an image in the bounding rectangle. Again, you will not need to be concerned with the graphical aspects of the implementation, except to the extent that you want to experiment with the demo applications and try things out.

The general design of the game framework is based on a timer that periodically triggers a callback method, say, 25 times per second (the so-called *frame rate*). You would, in practice, implement a game by setting up the game elements and then customizing the callback method to update element positions, handle collisions, generate new enemies, keep score, etc. The most fundamental operation in the callback is to invoke the **`update()`** method on each element. In the simplest case of something like **`SimpleElement`** that never actually moves or changes, the **`update`** method does nothing except increment a counter. In a (slightly) more interesting element type such as **`MovingElement`**, the **`update`** method might also change the element's position by adding a "velocity" (deltaX, deltaY) to the x and y coordinates of the position.

Here is a brief overview of the eight concrete types, in alphabetical order. Each one must directly or indirectly extend **`AbstractElement`**. *For detailed method specifications, see the Javadoc.*

`SimpleElement:` Simplest concrete subclass of the **`AbstractElement`** type. The update method just increments a frame counter.

`AttachedElement:` An element that is always associated to another “base” element such as a PlatformElement or LiftElement. It only moves with the associated base object, remaining a fixed distance above it. In addition to the methods specified in Drawable and AbstractElement, it has a method **`setBase`**.

`FlyingElement:` An element whose position is updated each frame according to its vertical velocity. Additionally, to simulate gravity, the vertical velocity is adjusted each frame by a gravitational constant. In addition to the methods specified in Drawable and AbstractElement, it has the methods **`setVelocity, setGravity, setGrounded, getDeltaX`**, and **`getDeltaY`**.

`FollowerElement:` An element that is always associated to another “base” element such as a PlatformElement or LiftElement. Optionally, it can be configured with a nonzero horizontal velocity and it will oscillate back and forth between the left and right edges of the base element it is associated to, while remaining directly above it. In addition to the methods specified in Drawable and AbstractElement, it has methods **`setVelocity, getDeltaX`**, **`getDeltaY, setBounds, getMax, getMin,`** and **`setBase.`**

`LiftElement:` An element with two distinctive behaviors: First, it can be configured to move vertically within a fixed set of boundaries. On reaching a boundary, the y-component of its velocity is reversed. Second, it maintains a list of elements that are associated with it (see AttachedElement and FollowerElement) whose basic motion all occurs relative to the LiftElement. Invoking update on a LiftElement updates the LiftElement's position, adjusts the velocity as necessary, and then invokes update on each associated element. In addition to the methods specified in Drawable and AbstractElement, it has the methods **`setVelocity, getDeltaX`**, **`getDeltaY, setBounds, getMax, getMin, addAssociated, deleteMarkedAssociated`**, and **`getAssociate`**.

`MovingElement:` An element in which the update method adjusts the position each frame with a velocity vector (deltaX, deltaY). The units of velocity are "pixels per frame". In addition to the methods specified in Drawable and AbstractElement, it has the methods **`setVelocity, getDeltaX`**, and **`getDeltaY`**.

`PlatformElement:` An element with two distinctive behaviors: First, it can be configured to move horizontally within a fixed set of boundaries. On reaching a boundary, the x-component of its velocity is reversed. Second, it maintains a list of elements that are associated with it (see AttachedElement and FollowerElement) whose basic motion all occurs relative to the PlatformElement. Invoking update on a PlatformElement updates the PlatformElement's position, adjusts the velocity as necessary, and then invokes update on each of its associated elements. In addition to the methods specified in Drawable and AbstractElement, it has the methods **`setVelocity, getDeltaX`**, **`getDeltaY, setBounds, getMax, getMin, addAssociated, deleteMarkedAssociated`**, and **`getAssociated`**.

`VanishingElement:` An element that does not move, but it has an internal counter that is decremented each frame. When the counter reaches zero, the element marks itself for deletion.

Note that the Javadoc was generated under the simple assumption that each concrete class was a direct extension of **`AbstractElement`**, i.e., that no inheritance was being used within these eight classes. So when you see in the Javadoc the line "**`extends api.AbstractElement`**", please do not take that literally as part of the spec! In a well-written implementation of the hierarchy, it is extremely unlikely, for example, that the **`FlyingElement`** class would directly extend **`AbstractElement`**.

The Demo applications

There are some sample graphical applications to try out in the `demo` package. These are highly experimental and almost certainly have bugs. As always, these are just for fun, do not rely on the GUI to accurately test your code!

- PigsExample – requires only that you have implemented SimpleElement
- MazeExample – requires SimpleElement and MovingElement, use arrow keys to move
- JumpExample1 – requires SimpleElement and FlyingElement, use arrow keys, and 'a' key to jump
- VanishingElementExample – requires VanishingElement
- WhenPigsFly – requires SimpleElement, MovingElement, FlyingElement, and
- PlatformExample – requires PlatformElement, LiftElement, AttachedElement, and FollowerElement
- JumpExample2 - requires PlatformElement, LiftElement, AttachedElement,
- FollowerElement, and FlyingElement, use arrow keys and 'a' key to jump
- SuperPigsExample – requires everything! use arrow keys, and 'a' key to jump

Testing and the SpecChecker

As always, you should try to work incrementally and write tests for your code as you develop it.

Do not rely on the UI demo code for testing! Trying to test your code using a UI is very slow, unreliable, and generally frustrating. The UI code is complex and not guaranteed to be free of bugs***. In particular, when we grade your work we are NOT going to run any of the demo programs, we are going to test that each method works according to its specification.***

We will provide a basic SpecChecker, but **it will not perform any functional tests of your code**. At this point in the course, you are expected to be able to read the specifications, ask questions when things require clarification, and write your own unit tests.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

The SpecChecker will verify the class names and packages, the public method names and return types, and the types of the parameters. If your class structure conforms to the spec, you should see a message similar to this in the console output:

```
x out of x tests pass.
```

This SpecChecker will also offer to create a zip file for you that will package up everything in your hw4 package. **Please check this carefully. In this assignment you may be including one or more abstract classes of your own, in addition to the 8 required classes, so make sure they have been included in the zip file.**

Importing the sample code

The sample code is distributed as a complete Eclipse project that you can import. However, the packages **`sample_tests`** and **`demo`** are posted separately, to keep you from getting a ton of compile errors at first. The **`hw4`** package includes a basic skeleton for each required class, including **only** the class and constructor declarations.

1. Download the zip file **`sample_code.zip`**. You don't need to unzip it.
2. In Eclipse, go to File -> Import -> General -> Existing Projects into Workspace, click Next.
3. Click the radio button for "Select archive file".
4. Browse to the zip file you downloaded and click Finish.
5. When you want to try something from **`sample_tests`** or **`demo`**, create the package in your project, download the respective zip file, extract it, and drag the file(s) you want into the appropriate package. If you have an older version of Java or if for some reason you have problems with the process above, or if the project does not build correctly, you can construct the project manually as follows:
6. Extract the zip file containing the sample code (*Note: on Windows, you have to right click and select "Extract All" – it will not work to just double-click it*)
7. In Windows Explorer or Finder, browse to the src directory of the zip file contents
8. Create a new empty project in Eclipse
9. In the Eclipse Package Explorer, navigate to the src folder of the new project.
10. Drag the **`hw4`** and **`api`** folders from Explorer/Finder into the **`src`** folder in Eclipse.

Getting started

At this point we expect that you know how to study the documentation for a class, write test cases, and develop your code incrementally to meet the specification, so this getting started section will not be quite as detailed as for the previous assignments. You'll find that the code is not terribly difficult – other than the inheritance aspects, most of these classes could have been given as a pretty easy homework 2.

Step 1: Most importantly – be sure you have done and understood the second checkpoint of lab 8. Sometimes the best way to start on an inheritance hierarchy is to, initially, forget about inheritance. That is, if you have two classes A and B that might be related, just implement both of them completely from scratch. Then look at what they have in common. Does it make logical sense for one to extend the other? Should the common code be factored into a common, abstract superclass?

2. It looks like **`SimpleElement`** is just a direct implementation of all the abstract methods specified in **`AbstractElement`**, so that would seem a likely place to start. As always begin by writing a simple test case so you know what you are expecting to happen, for example (see BasicTest.java in the **`sample_tests`** package):

```
SimpleElement e = new SimpleElement(2.3, 4.5, 10, 20);
System.out.println(e.getXReal()); // expected 2.3
System.out.println(e.getXInt()); // expected 2
System.out.println(e.getYReal()); // expected 4.5
```

```
System.out.println(e.getYInt()); // expected 5
System.out.println(e.getWidth()); // expected 10
System.out.println(e.getHeight());// expected 20
System.out.println(e.getRect());

// expected java.awt.Rectangle[x=2,y=5,width=10,height=20]
System.out.println(e.isMarked()); // expected false
```

Then try out some mutator methods too...

```
// setting position
e.setPosition(42, 137);
System.out.println(e.getXReal()); // expected 42
System.out.println(e.getYReal()); // expected 137

// update should increment the frame count
e.update();
e.update();
System.out.println(e.getFrameCount()); // expected 2

// mark
e.markForDeletion();
System.out.println(e.isMarked()); // expected true

// intersections
e = new SimpleElement(10, 0, 10, 10);
SimpleElement e2 = new SimpleElement(1, 5, 10, 10);
System.out.println(e.collides(e2)); // expected true e2.setPosition(0, 5);
System.out.println(e.collides(e2)); // expected false
```

(Tip: the java.awt.Rectangle class includes a method **`intersects`** that might be useful!)

3. You might next try **`MovingElement`**, **`FlyingElement`**, and **`VanishingElement`**, which appear to be fairly similar to

`SimpleElement`. Write some simple tests as above for all the methods. For example, how does gravity work? (See FlyingElementTest.java in the **`sample_tests`** package.)

```
FlyingElement p = new FlyingElement(0, 0, 0, 0);
p.setGrounded(false);
p.setVelocity(2, 3);
p.update();
System.out.println(p.getYReal()); // expected 3
System.out.println(p.getDeltaY());// expected 3
p.update();
System.out.println(p.getYReal()); // expected 6
System.out.println(p.getDeltaY());// expected 3
p.setGravity(5);
p.update();
System.out.println(p.getYReal()); // 6 + 3 = 9
System.out.println(p.getDeltaY()); // 3 + 5 = 8
p.setGrounded(true);
p.update();
```

```
System.out.println(p.getYReal()); // 9 + 8 = 17
System.out.println(p.getDeltaY()); // 8 (grounded)
p.update();
System.out.println(p.getYReal()); // 17 + 8 = 25
System.out.println(p.getDeltaY()); // 8 (grounded)
```

4. Maybe the next thing to tackle is **`PlatformElement`**. Sketch out a couple of examples by hand to be sure you can see how to update the position and velocity when reaching the min or max position. Try creating some test cases, for example (see **`PlatformElementTest.java`**):

```
// left side at x = 50, width 10, right side at 60
PlatformElement p = new PlatformElement(50, 200, 10, 10);
p.setBounds(40, 70);
p.setVelocity(6, 0);
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [56, 66]
System.out.println("Velocity " + p.getDeltaX()); // 6.0
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [60, 70]
System.out.println("Velocity " + p.getDeltaX()); // -6.0
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [54, 64]
System.out.println("Velocity " + p.getDeltaX()); // -6.0
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [48, 58]
System.out.println("Velocity " + p.getDeltaX()); // -6.0
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [42, 52]
System.out.println("Velocity " + p.getDeltaX()); // -6.0
p.update();
System.out.println(p.getXReal() + ", " + (p.getXReal() + 10));  // [40, 50]
System.out.println("Velocity " + p.getDeltaX()); // 6.0
```

5. You can't complete the implementation of **`PlatformElement`** without checking its behavior with associated elements. It looks like **`AttachedElement`** is quite a bit simpler than **`FollowerElement`**, so we could start with that. See **`PlatformWithAttachedTest.java`**.

```
// left side at x = 50, width 10, right side at 60
PlatformElement p = new PlatformElement(50, 200, 10, 10);
p.setBounds(40, 70);
p.setVelocity(6, 0);
// size 5 x 5, offset 2 units from left of PlatformElement, 15 above
AttachedElement c = new AttachedElement(5, 5, 2, 15);
// this should automatically make p the base of c
p.addAssociated(c);
// x position should be the base position + 2 = 52
// y position should be base y - AttachedElement height - hover = 180
System.out.println(c.getXReal());  // expected 52
System.out.println(c.getYReal());  // expected 180
p.update();
System.out.println(c.getXReal());  // expected 58
System.out.println(c.getYReal());  // expected 180
p.update();
System.out.println(c.getXReal());  // expected 62
```

```
System.out.println(c.getYReal());  // expected 180
// calling update on p should update associated elements too
System.out.println(c.getFrameCount()); // expected 2
```

6. Time to think about **`FollowerElement`**, maybe? It has the same oscillating behavior as a PlatformElement, but all motion is relative to the PlatformElement it's on. First think about what it should do when the PlatformElement isn't moving, that should be the easiest case. See **`PlatformWithFollowerTest.java`**.

```
// first try an example where the base doesn't move
// left side at x = 50, width 10, right side at 60
PlatformElement p = new PlatformElement(50, 200, 10, 10);
p.setBounds(40, 70);
// size 5 x 5, initial offset 2 units from left of PlatformElement
FollowerElement e = new FollowerElement(5, 5, 2);
e.setVelocity(2, 0);
// this should automatically make p the base of e
// and the left and right sides of p the boundaries of e
p.addAssociated(e);
System.out.println(e.getMin());  // 50
System.out.println(e.getMax());  // 60
// x position should be the base position + 2 = 52
// y position should be base y - FollowerElement height = 195
System.out.println(e.getXReal());  // expected 52
System.out.println(e.getYReal());  // expected 195
// PlatformElement doesn't move here, but FollowerElement does
p.update();
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));//expected 54,59
System.out.println(e.getDeltaX()); // expected 2.0
p.update();
// this should hit the right boundary
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));//expected 55,60
System.out.println(e.getDeltaX()); // expected -2.0
System.out.println();
```

7. How about if PlatformElement is moving, but the FollowerElement isn't?

```
// next, what if PlatformElement is moving, but FollowerElement velocity 0?
// left side at x = 50, width 10, right side at 60
p = new PlatformElement(50, 200, 10, 10);
p.setBounds(40, 70);
p.setVelocity(3, 0);
// size 5 x 5, initial offset 2 units from left of PlatformElement
e = new FollowerElement(5, 5, 2);
p.addAssociated(e);
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));//expected 52,57
// when p moves, the boundaries of e should shift
p.update();
System.out.println("bounds " + e.getMin() + ", " + e.getMax());// 53, 63
// but e is still 2 units from left side
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));//expected 55,60
System.out.println();
```

8. The trickiest bit is when both the PlatformElement and FollowerElement are both moving. For example:

```
// next, what if PlatformElement is moving, and FollowerElement velocity is
nonzero?
// left side at x = 50, width 10, right side at 60
p = new PlatformElement(50, 200, 10, 10);
p.setBounds(40, 70);
p.setVelocity(3, 0);
// size 5 x 5, initial offset 2 units from left of PlatformElement
e = new FollowerElement(5, 5, 2);
e.setVelocity(2, 0);
p.addAssociate(e);
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));//expected 52,57
p.update();
// e is now 4 units from left bound, since its velocity is 2
System.out.println("bounds " + e.getMin() + ", " + e.getMax());// 53, 63
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));  // 57, 62
p.update();
// p has moved to [56, 66], e attempts to move another 2 units
// relative to p, to [62, 67], but hits the right boundary at 66
// and reverses direction
System.out.println("bounds " + e.getMin() + ", " + e.getMax());// 56, 66
System.out.println(e.getXReal() + ", " + (e.getXReal() + 5));  // 61, 66
System.out.println("velocity " + e.getDeltaX());  // expected -2
```

You should be able to further develop the test cases above (for example, does your code work when the FollowerElement hits the left boundary?)

9. Try working on similar tests for **`LiftElements`** with **`AttachedElements`** and **`FollowerElements`**.

Requirements and guidelines regarding inheritance

A generous portion of your grade (maybe 20%) will be based on how well you have used inheritance, and possibly abstract classes, to create a clean design with a minimum of duplicated code. Please note that there is no one, absolute, correct answer – you have design choices to make.

You will explain your design choices in the class Javadoc for **`BaseElement`**.

Specific requirements

You are not allowed to use non-private variables. Call the superclass constructor to initialize its attributes, and use superclass accessor methods to access them. If your subclass needs some kind of access that isn't provided by public methods, you are allowed to define your own **`protected`** methods or constructors.

Other general guidelines

You should not use `instanceof` or `getClass()` in your code, or do any other type of runtime type-checking, to implement correct behavior. Rely on polymorphism.

- No class should contain extra attributes or methods it doesn't need to satisfy its own specification.

- Do not ever write a method like this (it is redundant):

```
public void foo()
{
  super.foo()
}
```

  There is almost never any reason to declare and implement a method that is already implemented in the superclass unless you need to override it to change its behavior. *That is the whole point of inheritance!*

Style and documentation

Special documentation requirement: you must add a comment to the top of the `SimpleElement` class with a couple of sentences explaining how you decided to organize the class hierarchy for the elements.

Roughly 10 to 15% of the points will be for documentation and code style. Some restrictions and guidelines for using inheritance are described above.

When you are overriding a superclass or interface method, **it is usually NOT necessary to re-write the Javadoc**, unless you are really, really changing its behavior. Just include the @Override annotation. (The Javadoc tool automatically copies the superclass Javadoc where needed.)

The other general guidelines are the same as in homework 3. Remember the following:

- You must add an @author tag with your name to the javadoc at the top of each of the classes you write.
- You must javadoc each instance variable and helper method that you add. Anything you add must be `private` or `protected`.
- Keep your formatting clean and consistent.
- Avoid redundant instance variables
- Accessor methods should not modify instance variables

If you have questions

It is extremely likely that the specification will not be 100% clear to you. Part of your job as a developer is to determine what still needs to be clarified and to formulate appropriate questions.

In particular, since you are not required to turn in any test code for this assignment, your tests and test cases can be freely shared and discussed on Piazza. *It is your responsibility to make sure that you correctly understand the specification and get clarifications when needed*.
For questions, please see the Piazza Q & A pages and click on the folder **`hw4`**. If you don't find your question answered already, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`hw4`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the buttons labeled "code" or "tt" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Acknowledgement

This project was inspired by the quirky and wonderful book, *Program Arcade Games With Python and Pygame*, http://programarcadegames.com/ , written by Paul Craven, who teaches at Simpson College in Indianola.

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw4.zip`** and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw4`**, which in turn contains the eight required classes along with any others you defined in the **`hw4`** package. **Please LOOK at the file you upload and make sure it is the right one and contains everything needed!**

Submit the zip file to Canvas using the Assignment 4 submission link and verify that your submission was successful.

We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw4**, which in turn should contain everything in your hw4 directory. You can accomplish this by zipping up the **hw4** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.