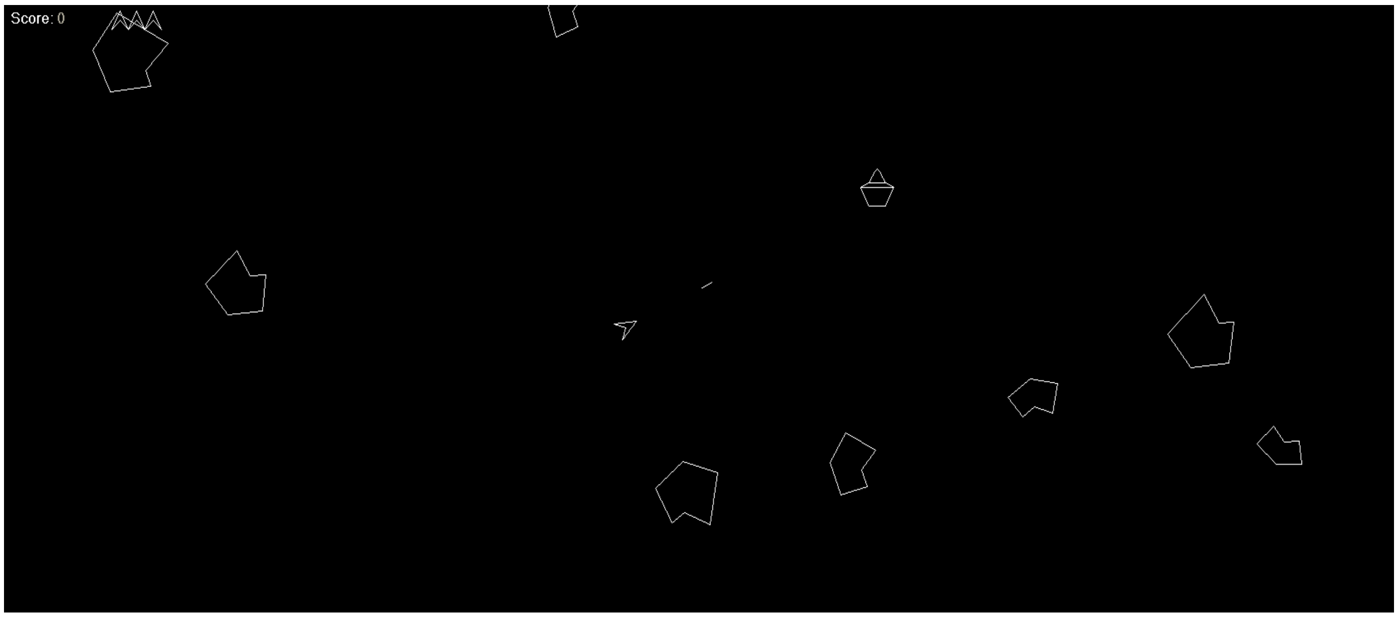
The Story

This week, you will reinvigorate a 1979 arcade classic, a timeless tale of triumph and sacrifice, of hatred and love, of space travel wrapped around our screens and our hearts. One solitary ship, an interminable journey, a violent struggle for freedom – these are the elements of our story…

ASTEROIDS



Getting Started

First things first: If you've never played Asteroids before, take no more than five minutes and click here to familiarize yourself with the game. Bask in the glory of its elegant gameplay.

I'm providing you with some distribution code -- courtesy of Leyzberg and Simon -- that displays a black screen. The goal here is less to become experts on graphical programming than it is to see inheritance applied. Your job in this problem set will be to take the blank screen and make the spaceship and asteroids appear.

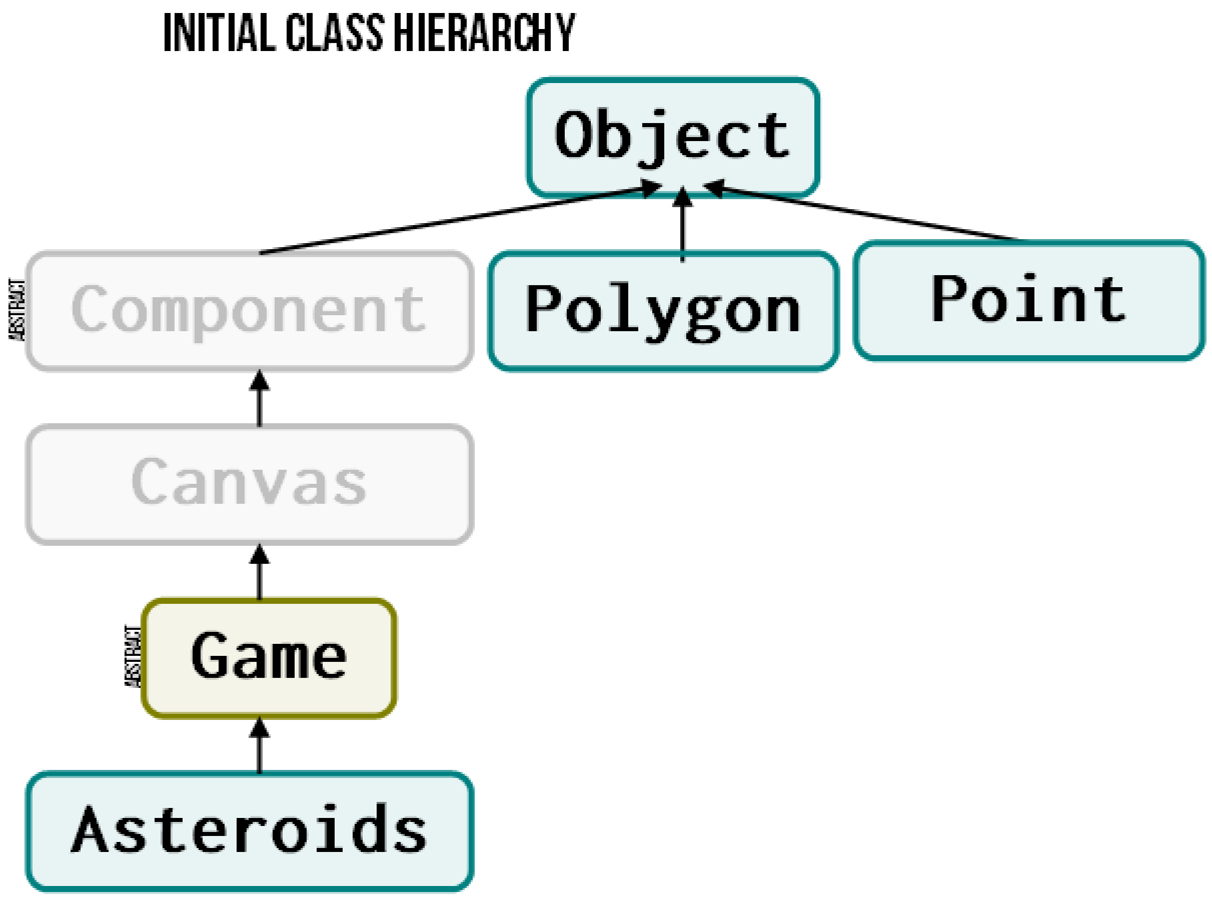
Setting Up

1. **Unzip the attached distribution code. Unzip the package, put the files somewhere convenient for you to access.**
2. **Create a new project in eclipse, name it** ADSB Problem Set 11
3. **Make a new package called mrgonzalez.<CSID>.ps11**
4. **Then, drag and drop the extracted files into your package (or copy and paste)**

Main Stuff

Part 1: Getting Started

Start by getting comfortable with the distribution code. In just a moment, you'll take a few minutes and read through the four classes included in the package. You can see that they exist in the following class hierarchy:



Note that `Component` and `Canvas` are not included in the distribution code; they're imported via the package `java.awt`. Also note that `Game` (and `Component`, for that matter) is an **`abstract`** class. Confused about why I haven't included `Component` and `Canvas`? `Game` itself is their direct subclass, and you won't need to make any other child classes of `Component` and `Canvas` other than `Game`. So we won't pay much attention to them.

Read through the four classes. Start with **`Point`**:

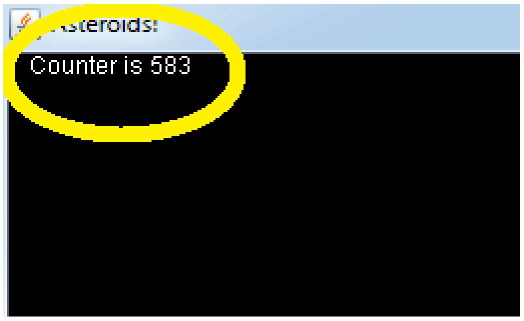
- Looks straightforward. There is a standard Point class in the AWT toolkit, but we'll use this given Point class instead. Note that here, the instance variables are declared as public. This is bad style but is done for your convenience so that you can access its instance variables (two int values) directly as `<objectName>.<inst var 1>` or, as an example with some point `p1`, `p1.x` and `p1.y`. If you'd like, you can also use the getters/setters. Either way works here. Do we have a main method here?

Move on to **`Polygon`**:

- What are the instance variables? What methods are defined? In a client class, how would you create a new Polygon object? Do we have a main method here?
- When you make a new Polygon object, you pass it three parameters:
  - An array of points -- the shape if your desired polygon if drawn in the top left corner (remember -- that's 0,0 in screen coordinates!)
  - A single point -- an offset for your polygon. Polygon has code that will shift your points (from the first parameter) so that your shape is now centered on the second parameter point. Changing this is how you "move" a shape.
  - A double -- how much you want to rotate your shape initially. 0 degrees is due east.

Look at **`Asteroids`**, which is your "control center" for this program:

- Interesting. What is this class's parent class?
- Does `Asteroids` have a main method? What happens when you run it?
- Look at the `Asteroids` constructor. What does it do?
  - Look at these two lines in the constructor:
    ```
    this.setFocusable(true);
    this.requestFocus();
    ```
    These lines just mean, "Make it so that this Component can be focused on" and then "Focus on this Component."
- Look at the `paint` method. To **paint** on this **canvas**, we use a **brush**. We use that brush (which gets passed to `paint` as a parameter, an instance of the `Graphics` class) in order to render different elements.
- By default, what is displayed in this program? You should be able to identify two items from the `paint` method. Identify the chunk of code that draws this:



```
Counter is 583
```

Look at the **`Game`** class, too -- though you won't need to make any modifications here.

`Asteroids` will call its `paint(Graphics brush)` method every tenth of a second to draw the next frame of the game's animation. Painting is simple: You call methods in the Graphics class using the brush, as in:

- `brush.fillPolygon(...)` or

- `brush.drawString(...)`
- `brush.drawLine(...)`

`Graphics API` here and `Graphics` tutorial here.

As we develop our code, you'll write `paint(Graphics brush)` methods for your own classes -- for instance, for a ship, for an asteroid, etc. The smart way to do call those functions is to do so from inside `Asteroids.paint(Graphics brush)`, passing the very same brush used in that method.

Part 2: Make the Spaceship Appear On Screen

Two important questions

A) How do we internally represent the ship?

That is, how do we store information about the ship, such as its shape, location on the screen, etc.

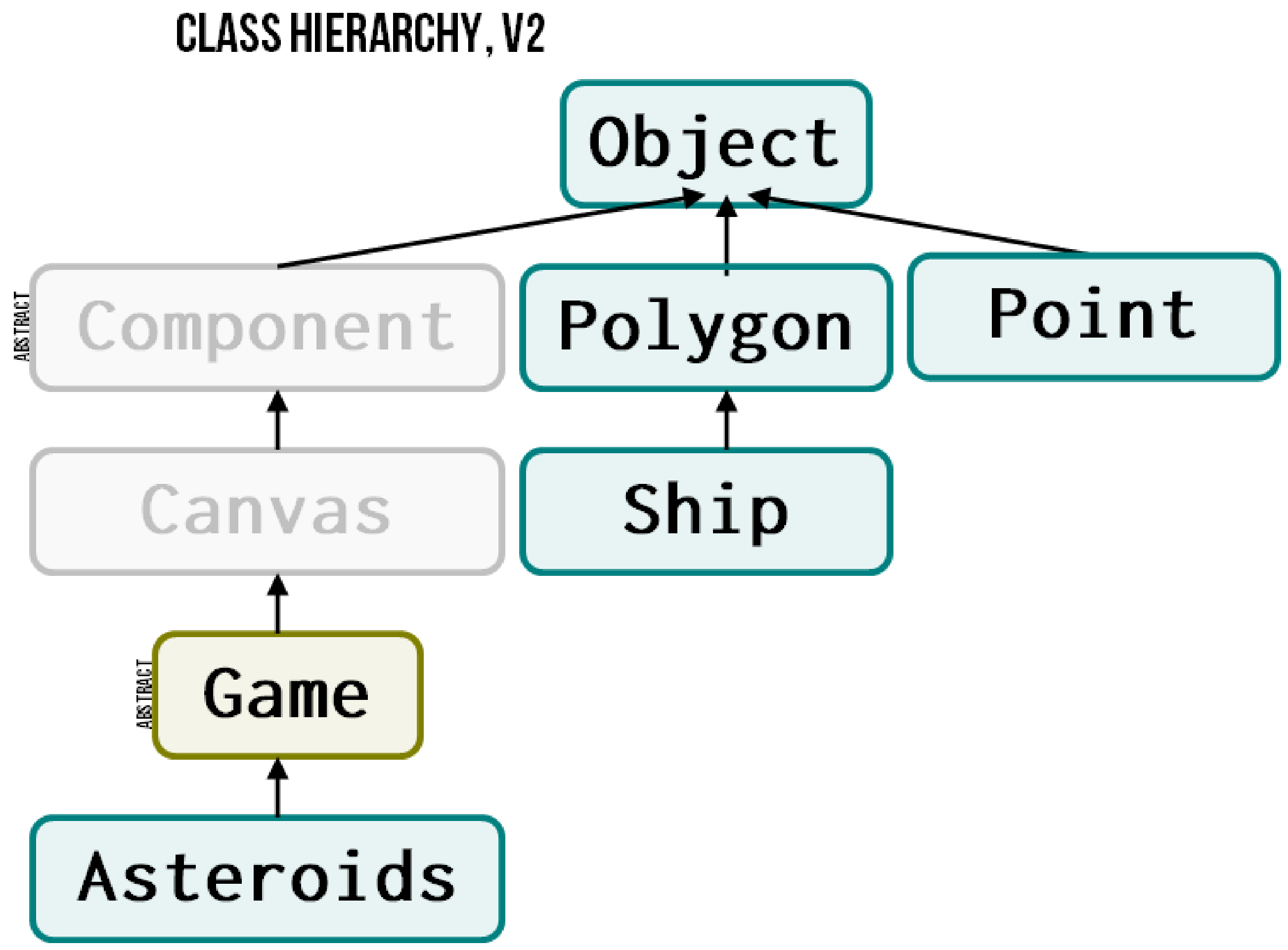
B) Where in the existing program do we write code to make the ship appear on screen?

Representing a Ship

We'll think about the first question first. **Look at the class `Polygon`.** Remember that triangles, rectangles, hexagons, etc., are all polygons. A `Polygon` object holds information about a polygon's shape, its position on the game canvas, and how much it has been rotated from its default configuration. Be sure to look at the instance variables and identify each. **Make sure you understand that a `Polygon` represents a shape as an array of `Point`s.**

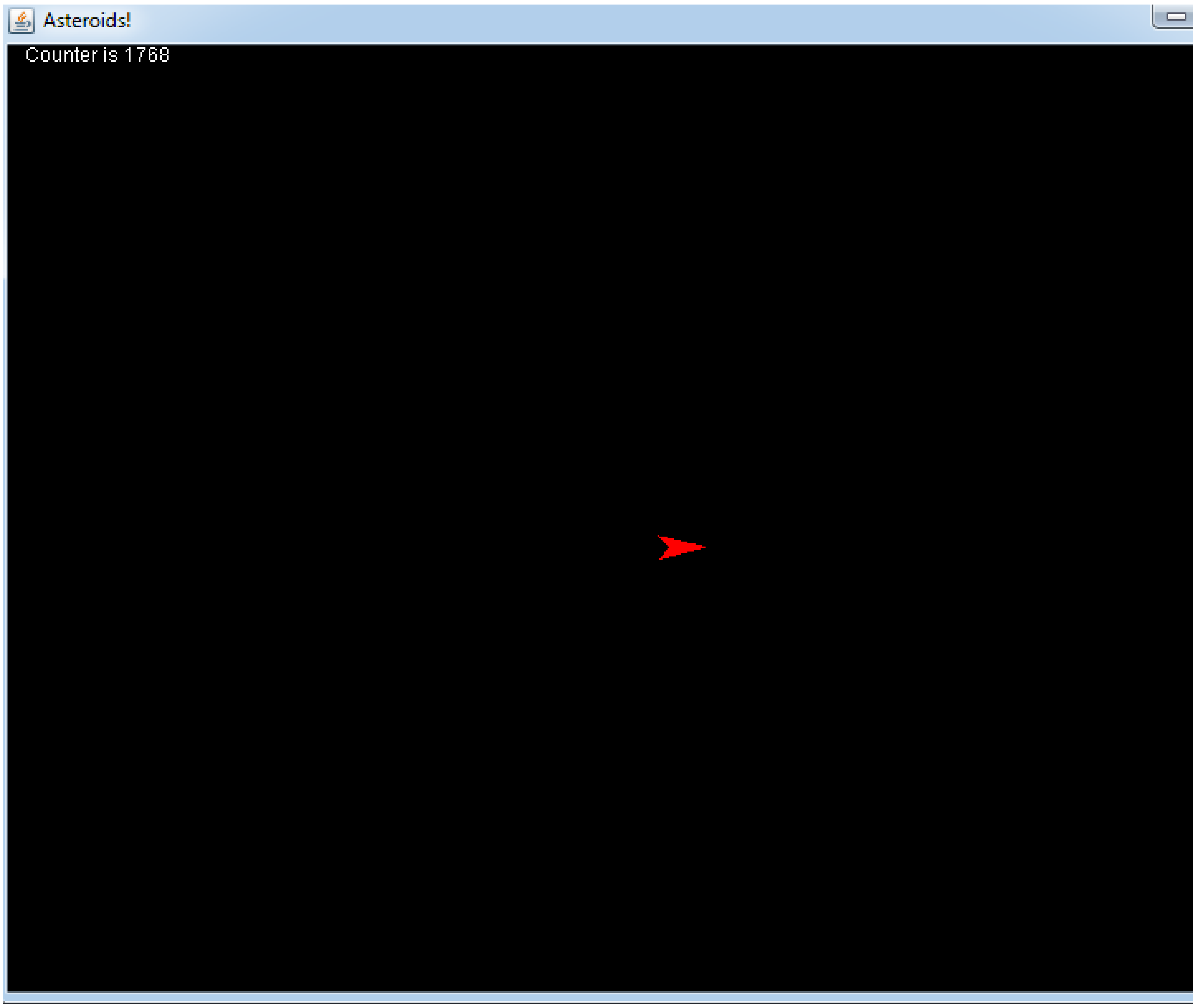
If you think about it, a ship is really just a type of `Polygon`. But we will want to use `Polygon` as a parent class for other types of objects in the game (e.g. the asteroids), so we won't alter `Polygon` at all to make it specific to the ship. Instead...

- **Write a new class, `Ship`, that is a subclass of `Polygon`**
- **Write the constructor for `Ship` -- for now, define the constructor that takes all the same parameters as the constructor in `Polygon`.**



Making the ship appear

Now, we need to create an instance of `Ship` and make it appear on the screen.



```
Counter is 1768
```

Look at the classes `Game` and `Asteroids`. Remind yourself of the relationship between these two classes. Is either abstract? Which class has `main` in it? What does the `Asteroids` constructor do?

Notice that `Game` is a subclass of `Canvas`, which is part of the user interface toolkit `AWT`. `Asteroids` therefore inherits (from `Game`, which inherits from `Canvas`) a method `paint(Graphics brush)` which is called every tenth of a second to update the animation.

We're eventually going to make an instance of the `Ship` class we defined earlier. But in order to make it appear, we need to give the `Ship` class a paint method that defines how it will be drawn.

- **In `Ship`, write an instance method `void paint (Graphics brush)` that draws the `Ship` instance.** You should use either the `fillPolygon` or `drawPolygon` method in the `Graphics` class.

- Note the three parameters of these methods. In your `Ship` class's paint method, you'll need to process your `Point[]` array called shape into two arrays that match the parameters of `fillPolygon` or `drawPolygon.` That means using `this.getPoints()` to get the array of `Point`s and then processing it into an array of x coordinate integers and an array of y coordinate integers.
- You might see that there are **overloaded versions of `fillPolygon` and `drawPolygon`** that take `Polygon`s as a parameter. However, these take a different type of `Polygon` object as a parameter -- one defined in the `awt` package, NOT the one we are using. So we **won't** be able to use those versions here.

Let's actually *make* our spaceship now. We want to create our ship when the game starts. **Think: Which method runs ONCE when we run this program? That's where you should instantiate your ship.** Note that we WON'T edit `main` at all.

- In that spot, you will instantiate a new `Ship` object. You'll declare your `Ship` ***variable*** as a class-level variable in`Asteroids`.
  - Before you actually construct the Ship object, first create an array of Point objects called `shipPoints` that contains a series of `Point` objects that define the shape of the ship. You'll pass this array into the constructor as the first parameter. For example, creating an array holding these points would make the ship a square with sides of ten pixels each:
    - `new Point(0,0)`
    - `new Point(10,0)`
    - `new Point(10,10)`
    - `new Point(0, 10)`

    - Note that the order in which you pass the points matters! The points should be in a logically drawn shape.

  - Make it start at the center of the screen.
  - Rotate it as necessary to get it to start the way you want it to.

  - OK, you've made your spaceship and you've defined how it will be drawn. Now make it draw.

- In `Asteroids.paint`, add a chunk of code that draws the ship you just made. (Now does it make sense why you declared your `Ship` variable outside of any methods? That makes it accessible to any `Asteroids` method.)

Now when you run `Asteroids`, you should see your ship!

Is your ship not showing up? A few things to consider:

- Casting has very high precedence. Are you properly casting the x coordinates and y coordinates of your ship's points when you get them in the Ship's paint method?

- Did you pass the Points to your Ship constructor in a logical order? i.e. is your ship polygon going to draw sensibly?

Part 3: Make the Spaceship Move Straight and Turn

Now that the ship is showing up on the screen, let's get it moving! Note that we'll ignore the zero-gravity floating/drifting effect that you see in the original game; you can implement that later as an extension activity. In other words, it's fine if your spaceship moves forward at a constant speed when the "up arrow" key is pressed. We'll also make it turn when the left/right arrow keys are pressed.

Make your spaceship move straight

In the `Ship` class, create a method `public void move()` that will change the position of your ship if the forward key is being held and change the rotation of the ship if either of the turn keys are being held. Don't worry yet about the keyboard responsiveness; first just focus on getting it to move on the screen without anything being pressed from the keyboard. We'll add keyboard responsiveness in a moment. We'll *call* this `move` method from `Asteroids.paint()` before we call the ship's `paint` method.

As you are implementing `move()`, think: What variable holds your ship's current position? How do you access the x coordinate and y coordinate of the ship's current position? Use a variable -- called something like `amountToMove` or `stepSize` -- to hold the basic step size of your ship.

Now add `boolean` instance variables for the forward, left, and right turn keys to your `Ship` class that can be set to `true` if the appropriate key is being held down and to `false` otherwise. For now, set the forward key variable to `true` and the other two to`false`. We'll set up the keyboard responsiveness in a moment.

Add code to `Ship`'s `move()` method so that the ship only moves forward when the the forward boolean variable is `true`.

Make your spaceship wrap around to the other side if it hits the edge

Now the ship should be moving (though not necessarily in the direction it's pointing). Make sure if it moves off of one side of the screen, the appropriate position variable wraps around so that the ship. Where will you check for this? Where do you have access to the width and height of the game screen?

You have a few options for how to do this:

- In the Ship class's move method. This means you have to hard-code the numbers for width and height (800 and 600 by default) in this method. This is sub-optimal because if you ever decide to change your game's size in Asteroids, this method immediately breaks.
  - Here's a way around it. Define a NEW constructor for the Ship class which ALSO takes width and height so that you can pass those in when you construct your new Ship object in Asteroids. Now your ship can refer to the width and the height.
- In the Asteroid class's paint method, check whether the ship has hit the edge after ship.move() but before ship.paint().

Add keyboard responsiveness

Now, your ship will move and will wrap around when it hits the edge. BUT it may not be moving in the direction that it's pointing. Let's fix that now by adding the `KeyListener` interface.

If you click the `KeyListener` API linked above, you'll see that implementing `KeyListener` means your class MUST include three methods:

- `public void keyPressed(KeyEvent e)`
- `public void keyReleased(KeyEvent e)`
- `public void keyTyped(KeyEvent e)`

These methods are called when keys are pressed (i.e. goes down) or released (i.e. key goes up). We'll actually leave the `keyTyped` method **empty** in our implementation, but we still have to *have* it there in order to satisfy the interface that we are implementing.

(Side note for the curious: `keyTyped` only functions when the key you type maps to a unicode character. For instance, hitting 'g' would trigger `keyTyped`, but hitting Escape would not.)

All of those `KeyListener` methods take a `KeyEvent` object as their parameter. This object contains information about which key was pressed or released. You can get the key code, which is a number representing the key pressed or released by calling the non-static (instance) method `getKeyCode()` on the `KeyEvent` instance that was passed in as a parameter. You'll want to check whether the `KeyEvent`'s key code is equal to whatever value you care about; see the pre-defined constants here. For instance, if the key code is equal to the constant `VK_ENTER`, it means the user hit the Enter key. Remember: These are `static` constants in the `KeyEvent` class. You have a `KeyEvent` object that was passed as a parameter into each of the `KeyListener` methods. That should tell you how to access them.

So: Fill in the `keyPressed()` and `keyReleased()` methods so that when your chosen keys are pressed or released, their corresponding `boolean` values (for forward, left, and right) are changed appropriately. Want an example? **Download this example code and put it in a package called `asteroidsKeyListenerDemo`.** Look at the class `Test` and run the `Asteroids` main method.

The `Canvas` object -- which is the superclass of `Game` and therefore of `Asteroids`, too -- generates `KeyEvent` objects when a key is pressed, released, or typed. Therefore, we need to register our new `KeyListener` with `Asteroids`, so that it will know where to send these event objects. We do this by adding the code

```
this.addKeyListener(whateverYourShipInstanceIsCalled);
```

in the constructor of `Asteroids`.

Set the default for your `boolean` variable for the forward key being pressed to `false`. Test your code to make sure that your ship moves when the forward key is pressed. Again, it might be moving in the wrong direction, but right now we just care about keyboard responsiveness.

Make the ship rotate

Add code in move() to make the ship rotate if the appropriate key is being held (i.e. if the correct boolean values are true). Test that the ship is rotating (though, again, it may still move in the wrong direction).

Fix the direction

Finally, let's get the ship moving in the direction it's facing.

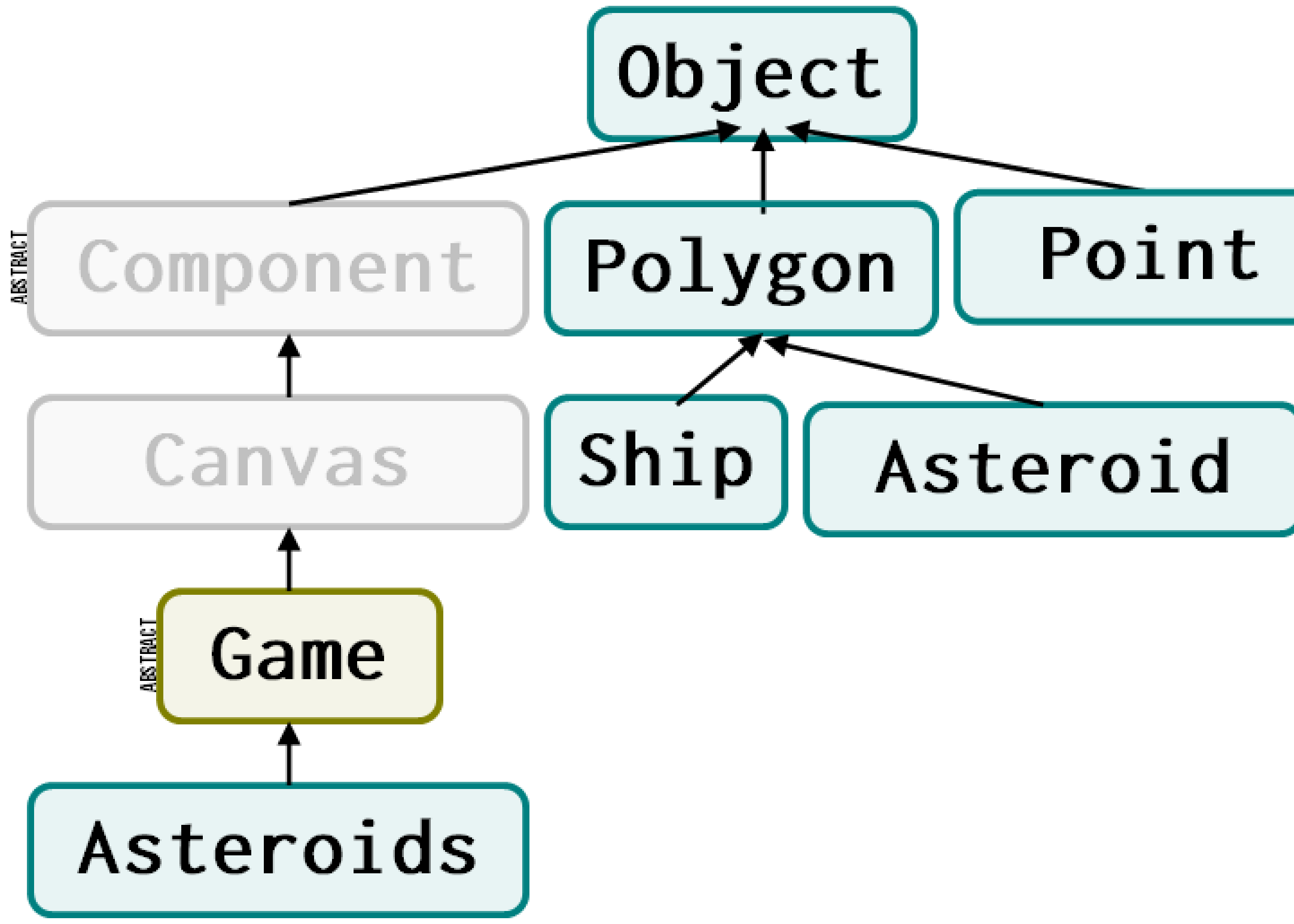
If the ship doesn't move in the correct direction even before pressing the turn key, then you need to change the direction that your ship is facing when it starts. Take a look at the third parameter you pass when you instantiate your `Ship` object.

If the ship moves in the correct direction when you start but then keeps moving in that same direction even after you press a turn key, then you need to use some trigonometry to make sure you are incrementing the x and y coordinates the correct amount to go the desired direction. Specifically, if you are currently incrementing the x and y coordinates by the same amount, you need to instead multiply the increase in the x coordinate by `Math.cos(Math.toRadians(rotation))` and multiply the increase in the y coordinate by `Math.sin(Math.toRadians(rotation))`.

Part 3: The Asteroid class

Now we'll create an `Asteroid` class that represents an asteroid -- not to be confused with the `Asteroid`**`s`** class, which is our game's "control center." We'll create an array of `Asteroid` objects and make them move across the screen.

CLASS HIERARCHY, V3



Write `Asteroid`

Create an `Asteroid` class that extends `Polygon`. Write a constructor and a `paint(...)` method that are similar to the ones in`Ship`. If you can figure out a way to use the same `paint(...)` method to draw both the ship and the asteroids (think about what class such a method should be in), you can save yourself some work by doing that instead. HINT: Consider an `abstract` class...)

Make a bunch of `Asteroids`

Now, create an array of `Asteroid` objects as an instance variable in the main game class, `Asteroids`. Use a loop in the constructor to instantiate them all. Test your code so far by making your array of Asteroids appear on the screen.

Now lets get the `Asteroids` moving. Write a `move()` method in `Asteroid` that is similar to the one for the ship, except it should move the `Asteroids` automatically, instead of waiting for the user to press a key.

Finally, add code to `Asteroids.paint(...)` in the main game class to call `move()` for each asteroid in the array.

Part 4: Check for Collisions between ship and asteroids

OK, now we want to check for whether or not our ship has collided with any/all of our asteroids. To test for this collision, we need to be able to test if two polygons are intersecting each other.

Look at the method `contains(Point point)` in `Polygon`. This method checks if the given `Point` is contained in whatever instance of `Polygon` is calling the method. That is, *are the coordinates of the given `Point` in the region defined by the `Polygon` calling it?* Notice that `contains()` calls `getPoints()`, so it is using the current location of the `Polygon` (with the offset and rotation applied) to get to the boundary.

Write a method called `collides(Polygon other)` in Polygon that uses the `contains()` method to test if two `Polygons` -- `this`and `other` -- intersect. What class should this go in? Should this method be static or non-static? What should the return type of this method be? There are a few different ways; your way need not be perfect, but it should at least detect the most obvious collisions.

Call this `collides` method from `Asteroids.paint`. You should have a loop in `Asteroids.paint` that iterates through your array of `Asteroid` objects, moves each one, checks if it has reached the edge and needs to be wrapped around, checks if it has collided with the ship, and then calls the `Asteroid`'s paint method.

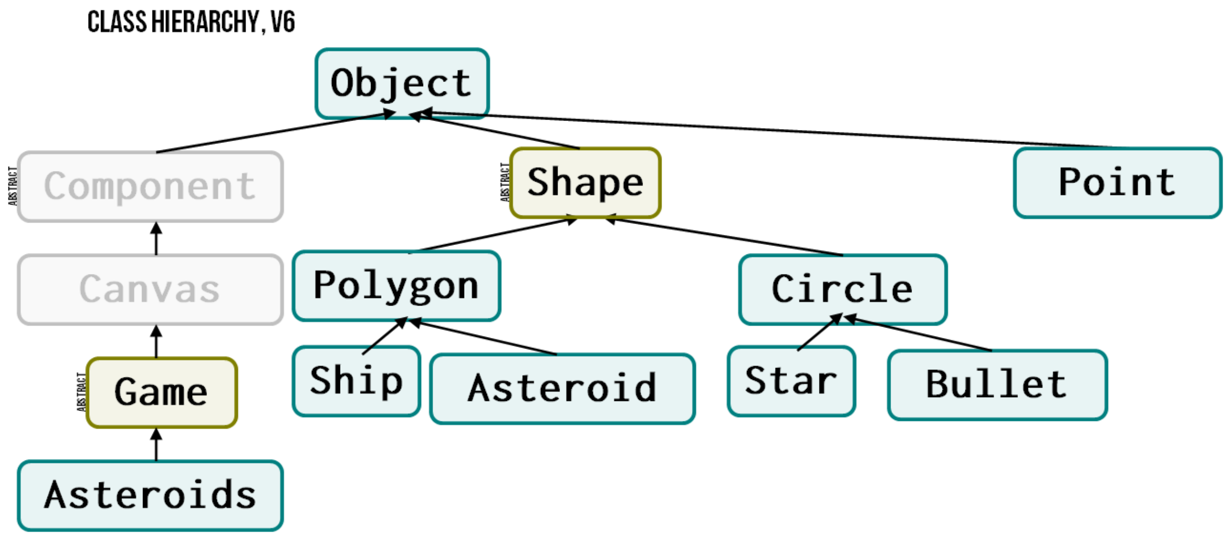
To test the code, make something happen on the screen if there is a collision between the ship and an asteroid. For example, make the word "Collision" appear at the top of the screen. Or you could do something fancier, like losing a life or losing points if your ship hits an asteroid. You could make this an extra feature for part 5.

Part 5: Add features of your choice

Now you've got some latitude. Obviously, our game isn't done, but now I'll give you an opportunity to do something a little more creative. You may either choose two of the following ideas to implement, or you can come up with your own. Feel free to do even more.

Weapons

- Define a `Circle` class (not unlike `Polygon`) and a bullet subclass; make the ship fire bullets when the user hits the spacebar. You might consider making both `Circle` and `Polygon` subclasses of an `abstract` class `Shape`.
- Test for bullets hitting asteroids; make the asteroids disappear when they are hit *You might eventually consider using this class hierarchy if you implement bullets* and/or stars



Asteroids

- Randomize the shapes and sizes of the asteroids
- Make the large asteroids split into smaller asteroids when they collide -- either by another asteroid or by a bullet or ship

Ship

- Add a zero-gravity acceleration**** effect so the ship flies as it does in the real game
- Make the ship visually collapse/explode when it hits an asteroid

Appearance/Other

- Add stars (also a subclass of Circle) in the background
- Add a timer; end the game if the player fails to destroy all the asteroids before the time runs out
- Add the alien ship that shoots at the real ship, as in the real game

Or anything else you can think of! You can even feel free to change the gameplay/game itself. Be creative.

Hint:

This will require one new member variable representing an acceleration vector. It is the rate of increase in x and in y at each time step. It can be enlarged with the following magical math:

```
public void accelerate (double acceleration) {
   pull.x += (acceleration * Math.cos(Math.toRadians(rotation)));
   pull.y += (acceleration * Math.sin(Math.toRadians(rotation)));
}
```