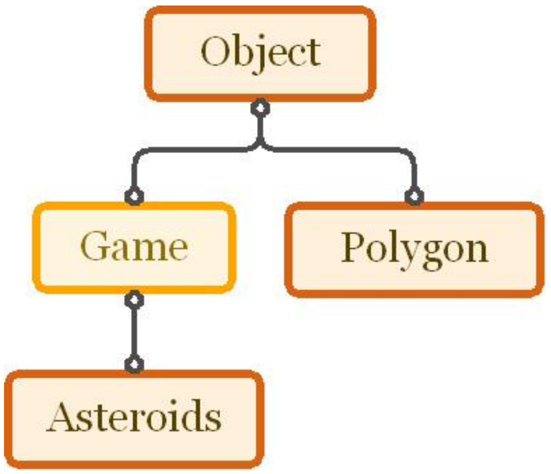
Asteroids

You will be reinvigorating the 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.

Prologue

First thing, if you've managed to spend your years without ever having played the game you're about to make, I expect you'll Google it and give it a try.



Initial class hierarchy: gold classes are abstract, orange are concrete.

We'll be charting the class hierarchy as it develops.

Second, you'll need to get comfortable with the starter code, so have yourself a perusal and come back when you're done.

Did that? I bet you've noticed a few curious things: Yes, *painting* on a *canvas* requires a *brush*. Yes, Java's coordinate system has its origin is at the leftmost, *topmost* point of a window. And, yes, you get a glorious *empty black window* when you run the starter code.

Well, now that you've met the brush and canvas – let's paint! `Asteroid` will call its `paint(Graphics brush)` method every tenth second – that way you can draw the next frame of the game's animation. Painting is done by calling the `Graphics` methods using the brush like so: `brush.fillPolygon(…)` or `brush.drawString(…)`. (See `java.awt.Graphics` for details.) As we develop our code, you'll write `paint(Graphics brush)` methods for your own classes; a sensible idea is to call those functions from inside `Asteroids.paint(Graphics brush)`, passing along that same brush!

Admittedly, what follows might drive some people nuts: it's a script for you to follow while coding. The truth is, you need not follow it exactly, the intent is for you to grasp the concepts it focuses on: class identity and inheritance design. As long as you've completed the assignment with all the details and are accountable for your class design, you've done what is asked. Be inventive, and enjoy!

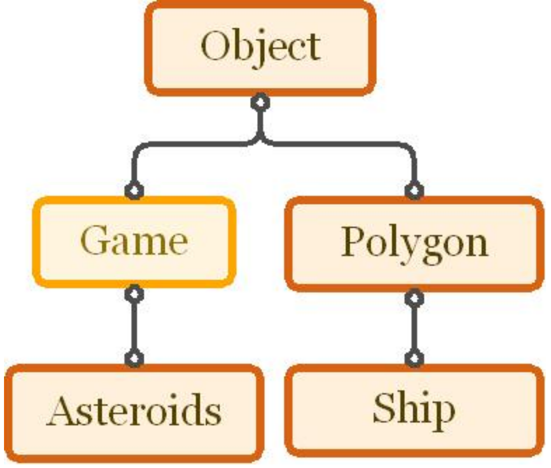
Following this script in exact terms is likely the easiest way to finish this assignment, though.

Characters

First up is the ship. Create a subclass of Polygon called Ship, implementing its constructor and the aforementioned `public void paint(Graphics brush)` draws the ship's polygon. Now make sure the ship starts at the screen's center.

How will the ship

know the screen dimensions?



Ship is a Polygon.

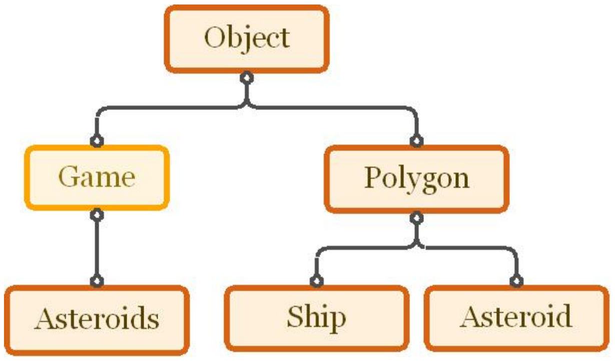
Experience has shown that keeping boolean variable for each button, toggled by KeyPressed() and KeyReleased() leads to smoother performance than using KeyTyped().

Now we'll get the ship moving. Create another method `public void move()` that increases the ship's `position.x` and call this function in `Asteroids.paint()`. Now we need the position values to wrap around when they gets out of bounds. After, you'll implement an interface called `KeyListener` that will allow our ship to respond to key presses. It requires you to implement three methods: `KeyPressed(...)`, `KeyReleased(...)`, `KeyTyped(...)`. I suggest leaving `KeyTyped(...)` blank.

Of course all the math isn't too hard to figure out, but let's just stick to programming here.

At this point, you've got a linear moving ship class. You press forward, it goes forward. You stop, it stops. What we'll need next is to create the zero-gravity acceleration effect: 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)));
```



Having trouble? The answer is **contained** within.

```
}
```

And now that you've got our protagonist in fighting form, he'll need a worthy opponent. Create an `Asteroid` class much in the same vein as the `Ship`. Create an array of `Asteroid` and paint them all.

How does adding paint(...) and any other methods you're considering adding change the meaning of what Polygon represents?

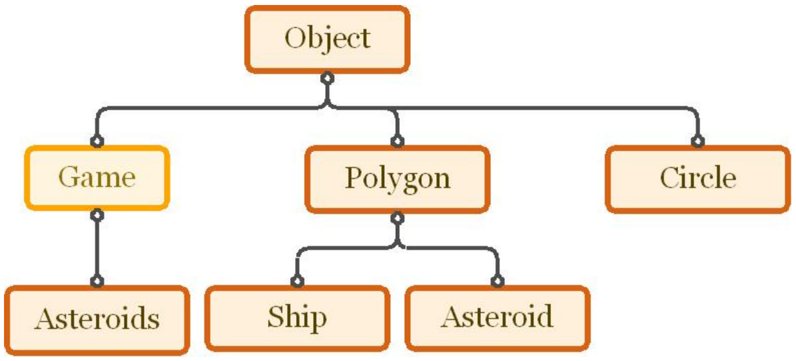
One detail remains: collisions. We need a method to determine whether a polygon intersects with another. The test aught to return true of either polygon overlaps the other – and let's put such a `boolean` method in our `Polygon` class. You're left to imagine how this might work.

Setting

Consider the `Star` and `Asteroid` classes. How are they similar? What abstractions can be made that apply to both – and more importantly, does that apply to their mutual abstraction: `Polygon`? Consider any similar methods, loops, and variables; do these properties belong to all polygons? Move as much as makes sense up into the `Polygon` class. At the least, `paint(...)` belongs in Polygon not in each and every subclass.

Remember that solving these design problems has immense impact on future work. You aught to be prepared to defend your decisions: Why do these methods belong here and not there? With which classes do the member variables best fit? What is the identity of the classes you've made? Are they consistent and reasonable real-world-like?

Conflict



Because bullets and stars aren't polygons.

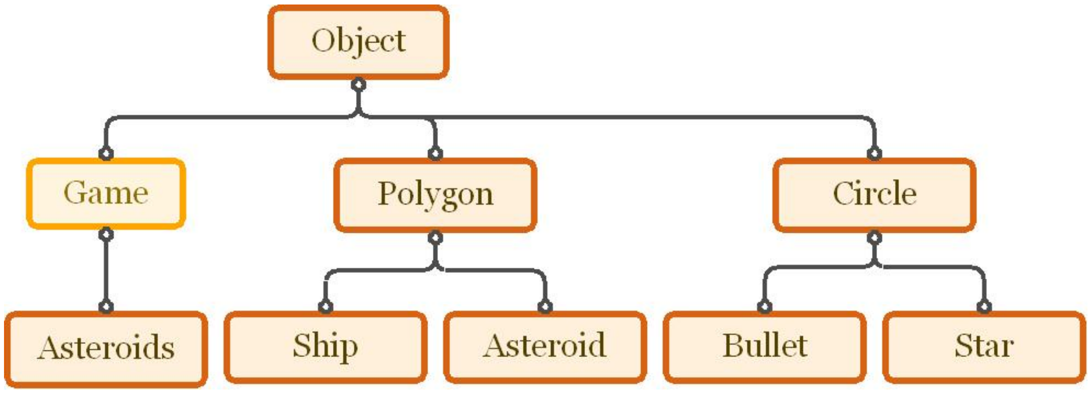
What's left? Oh, our ship has no bullets and our sky is an awfully lonely place. Now, everybody, what do stars and bullets have in common? Neither is a polygon! Both are... both are circles. That means we'll need a `Circle` class, but don't fret because it won't be nearly as hairy nor mathy as `Polygon` even though it will serve a parallel purpose.

We'll need a public `boolean contains(Point)` for intersections, a `paint(Graphics)` and that's it. So that's not bad at all. (Well, except, later on, it might come in handy to have implemented a method that returns a bunch of points lying on the boundary of our circle. Consider that a hint.)

Showdown

Now, much like with `Ship` and `Asteroid` earlier, we will subclass `Circle` to create `Bullet` and `Star`. I'll leave the details to you.

Consider making the stars twinkle or move toward or away from the ship. Consider making bullets that fade away or come in different colors or sizes.

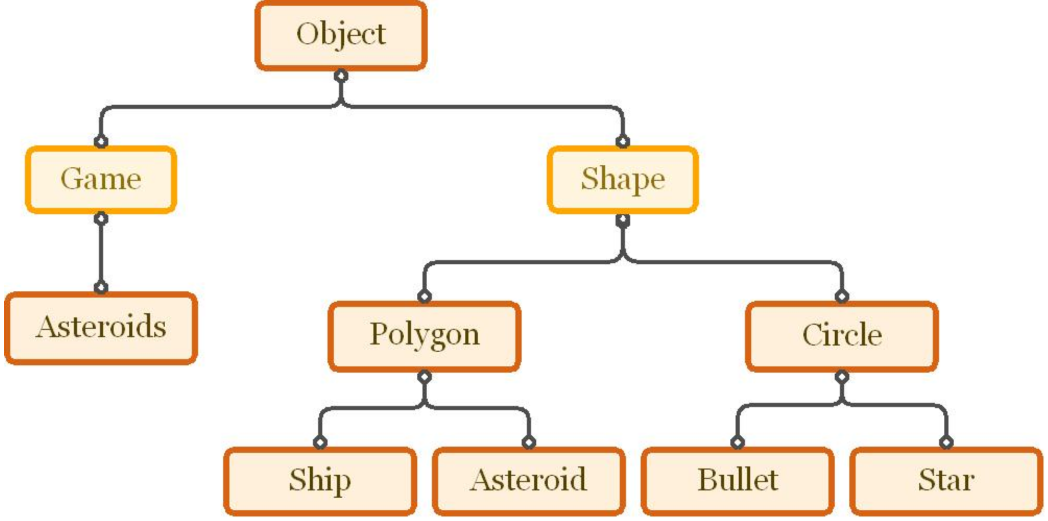


What's left? One major issue: how do we know a bullet has hit something? Our intersection-testing method is only for `Polygon`s!

We've hit on something here: we have a generalized. Instead of creating another method to repeat the test with slight variation for each possible shape, let's just talk about shapes in general!

You're about to create a super class for `Polygon` and `Circle`, we'll call it `Shape`. What are shapes? They may `contain(Point)`s, they may `intersect(Shape)`s, and you can `getPerimiter()`s – the perimeter is calculated differently based on type (in `Shape`, it should be an `abstract` method). Mixing abstract and implemented types means that there's no such thing in our world as a `Shape` for its own sake, it must be of some specific sort.

Shape thereby becomes its own



The glorious finalized hierarchy.

Denouement

Now we've got quite a beefy class hierarchy: abstract classes, implemented interfaces, lots of inheritance. This is your last opportunity to see that it all fits. Can you do better? Are the class identities still consistent and sensible?

Conclusion

Now we do the actual game fixins: scores, timers, and controls. Remember you can set the `Game`'s `on` member variable to `false`, which makes it stop `paint(...)`ing.

Here you're free to consider all the fun additions you might make: add sound, levels, difficulty, asteroids that explode into smaller asteroids, create explosive animations, use images for backgrounds or each ships. Do anything you like, it's your game!