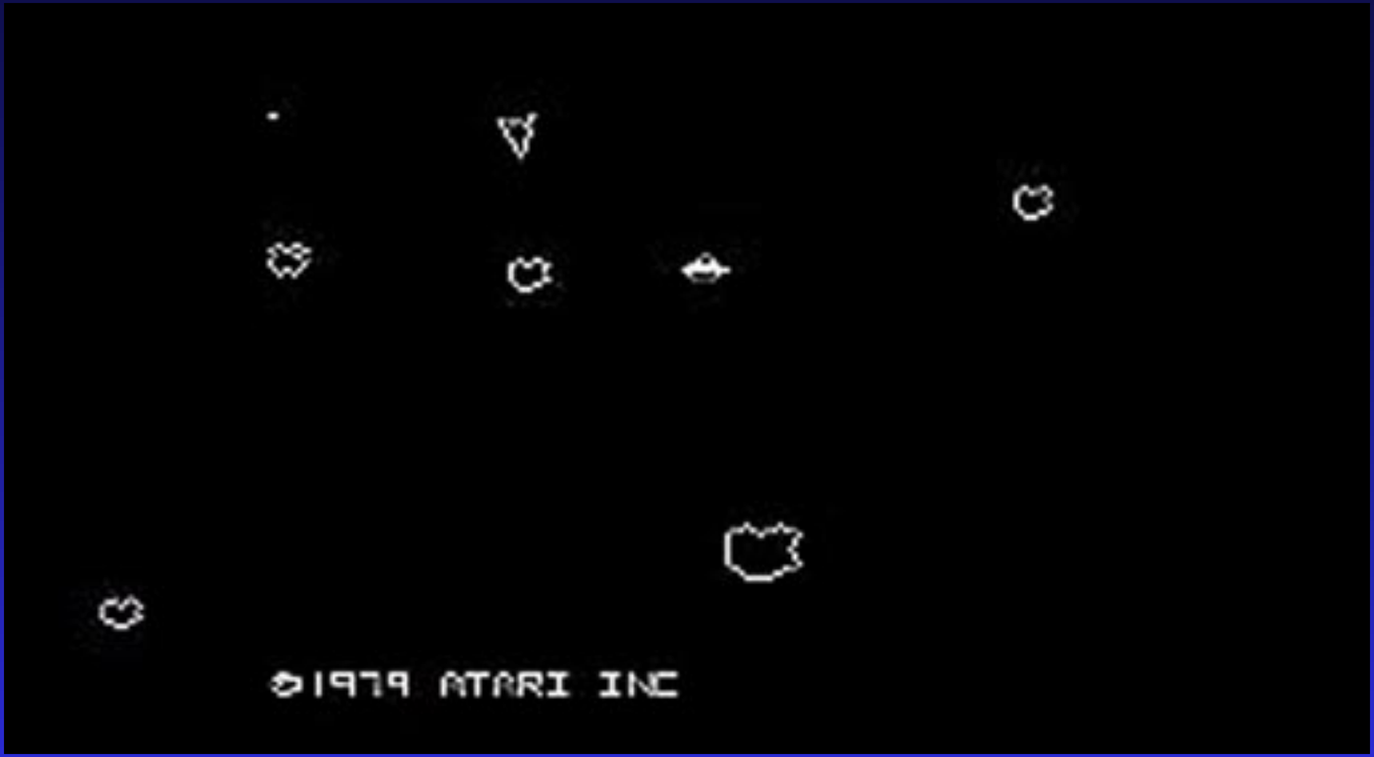
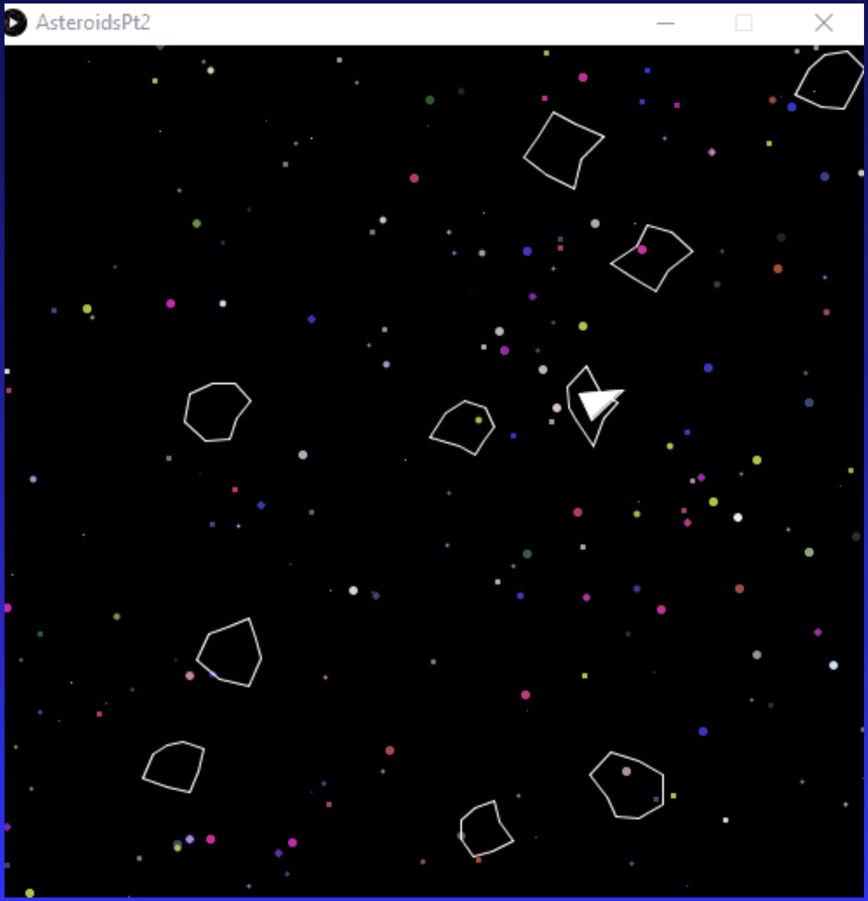
Unit_9c_AsteroidsProject Part 2



- We'll finish the Asteroids arcade game in parts 2 and 3

Asteroids Part 2: Adding the Asteroids

- Notice that the Asteroids move differently than the ship
- They rotate (or **`turn()`**) while they move



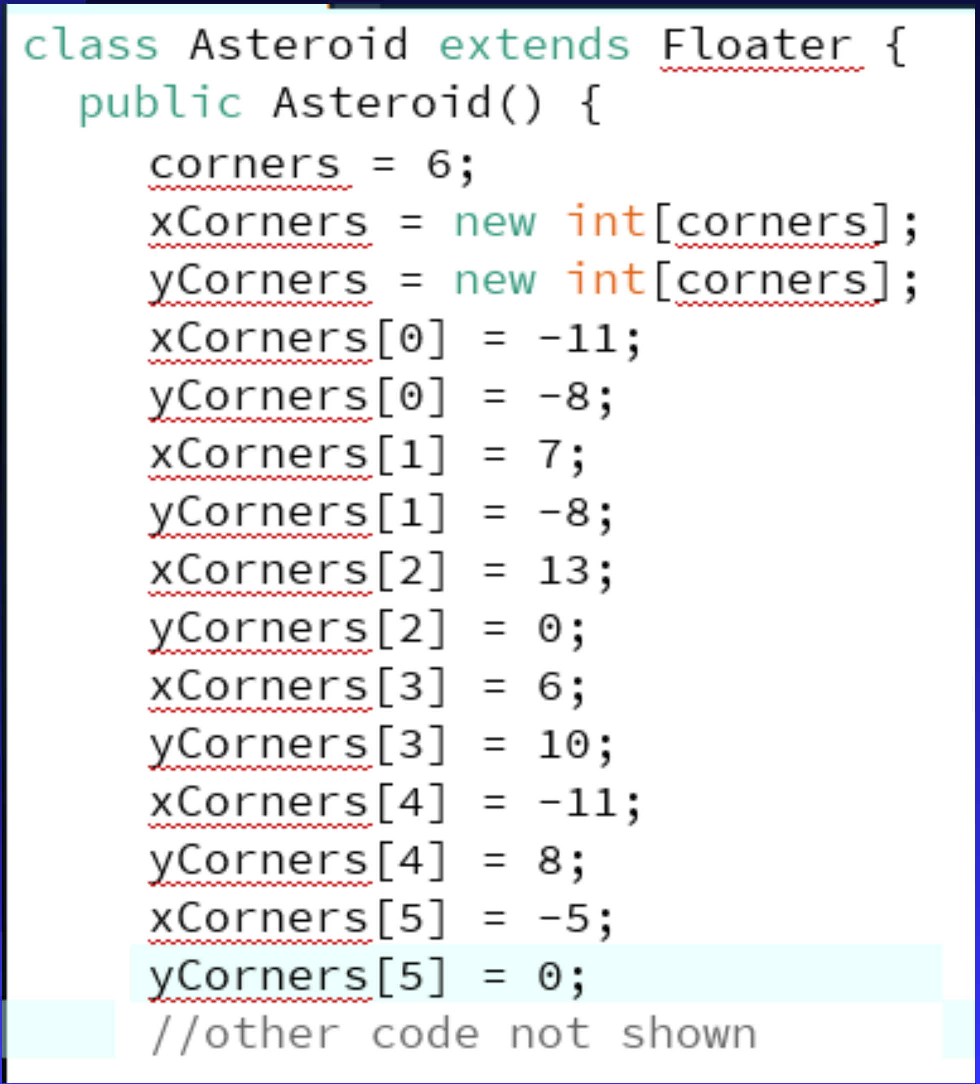
Asteroids Part 2

- In part 2 of Asteroids, we will create an **`Asteroid`** class
- You'll need a new member variable, constructor, **`move()`** and some getter and/or setter functions
- You will also need to encapsulate the class

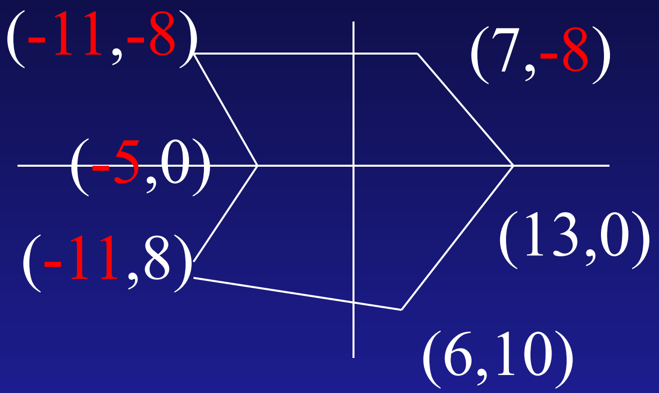
```
class Asteroid extends Floater
{
    ?? double rotSpeed; //randomly + or -
    ?? Asteroid(){ /*code not shown*/}
    ?? move(){ /*code not shown*/}

     //other getters and/or setters
     //may be necessary as well
```

Constructing an **`Asteroid`**

Note that half of your coordinates should be negative



```
class Asteroid extends Floater {
  public Asteroid() {
    corners = 6;
    xCorners = new int[corners];
    yCorners = new int[corners];
    xCorners[0] = -11;
    yCorners[0] = -8;
    xCorners[1] = 7;
    yCorners[1] = -8;
    xCorners[2] = 13;
    yCorners[2] = 0;
    xCorners[3] = 6;
    yCorners[3] = 10;
    xCorners[4] = -11;
    yCorners[4] = 8;
    xCorners[5] = -5;
    yCorners[5] = 0;
    //other code not shown
```



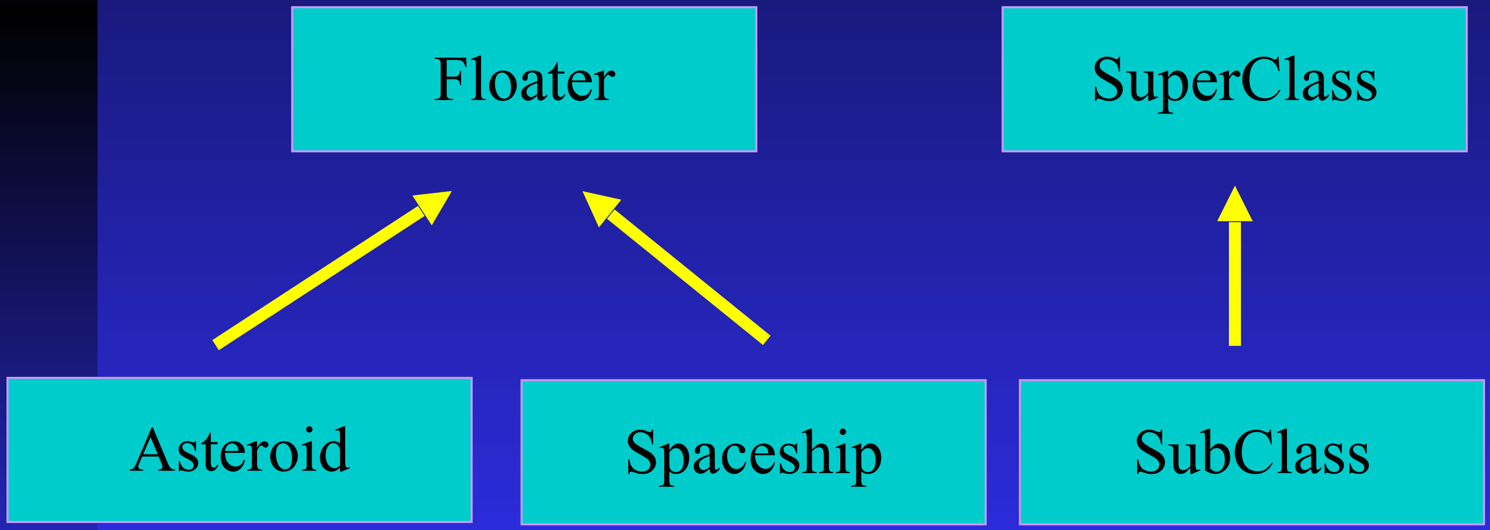
Important Vocabulary: Super and Sub classes

```
class Spaceship extends Floater
```

- `Spaceship` is the ***sub*** class and `Floater` is the ***super*** class
- Other less important vocabulary:
  - `Spaceship` is the ***derived*** class and `Floater` is the ***base*** class
  - `Spaceship` is the ***child*** class and `Floater` is the ***parent*** class

Inheritance Diagrams:
The arrow is important and always points up to the super class

```
class Asteroid extends Floater
class Spaceship extends Floater
class SubClass extends SuperClass
```



Note: the arrow designates the *is-a* relationship. A **`Spaceship`** *is-a* **`Floater`**

`super()`
`super.`

- The Java keyword **`super`** has *two* meanings:
  1. With *parenthesis*, it calls the super class *constructor* (e.g. **`super()`**)
  2. With a *dot*, it calls the *member methods* of the super class (e.g. **`super.move()`**)

`super()` vs. `super.`

1. By itself, **`super()`** calls the base class constructor. It must be used on the *first line* of the sub class constructor. If it's not called explicitly, there is an "invisible call" to the default no argument **`super()`**
2. **`super.method()`** calls the methods of the super class. You can use the methods in the super class to gain access to private variables that would otherwise be inaccessible

Error!

```
public void setup()
{
   Asteroid pete = new Asteroid(3);
   System.out.println(bob.getNum());
}
class Floater
{
   private int myNum;
   public Floater(int num){myNum = num;}
   public int getNum(){return myNum;}
}
class Asteroid extends Floater
{
   public Asteroid(int num){myNum = num;}
}
```

super() fixes the problem

```
public void setup()
{
  Asteroid bob = new SubClass(3);
  System.out.println(bob.getNum());
}
class Asteroid
{
  private int myNum;
  public Floater(int num_){myNum = num_;}
  public int getNum(){return myNum;}
}
class Asteroid extends Floater
{
  public Asteroid(int num_){super(num_);}
}
```

Overriding **`move()`**

- Asteroids turn (revolve) in addition to moving

```
?? void move()
{
    ??



    //other code not shown
```

Overriding `move()`

- Asteroids need to turn AND move

```
?? void move()
{
    turn(??);



    //other code not shown
```

Overriding **`move()`**

- Turn and then move just like a normal floater

```
?? void move()
{
    turn(rotSpeed);
    ??

    //other code not shown
```

Overriding **`move()`**

- You could copy and paste the **`move()`** code from **`Floater`**

```
?? void move()
{
   turn(rotSpeed);
   myCenterX += myXspeed;
   myCenterY += myYspeed;

   //wrap around screen
   if(myCenterX >width){
      myCenterX = 0;
   }
   //and so on. . .
```

But wait! There is a better way!
What could we put here to avoid copying and pasting code?

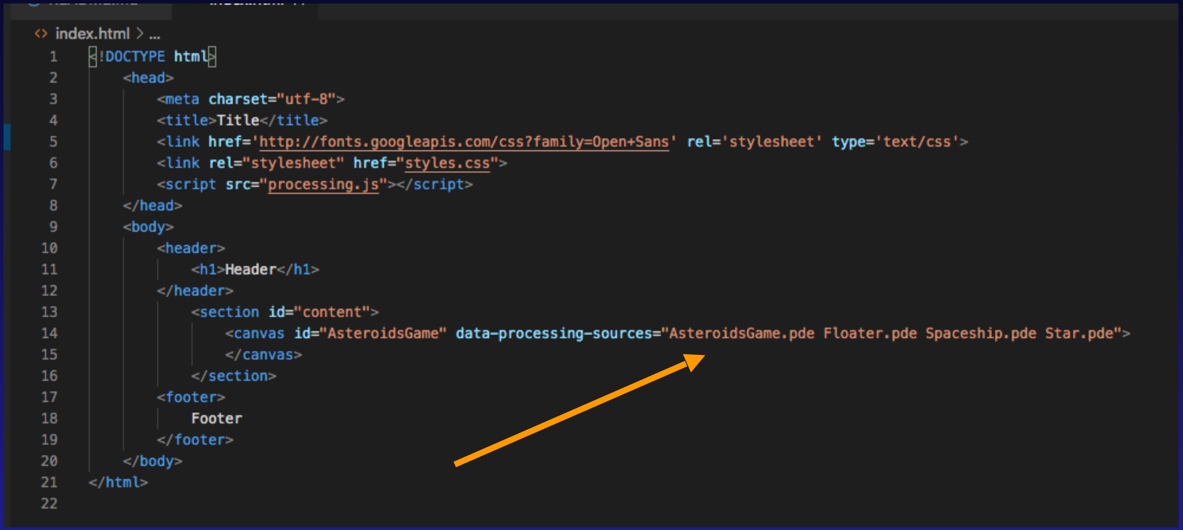
```
?? void move()
{
    turn(rotSpeed);
    ??
    //call move() in Floater
```

`super.move()`

- We could just tell java to use the **`move()`** function in the **`super`** class (**`Floater`**)

```
?? void move()
{
    turn(rotSpeed);
    super.move();
    //just 2 lines of code!
    //that's it!
}
```

Modifying index.html to load AsteroidsGame.pde


index.html > ...

```
1   <!DOCTYPE html>
2       <head>
3           <meta charset="utf-8">
4           <title>Title</title>
5           <link href='http://fonts.googleapis.com/css?family=Open+Sans' rel='stylesheet' type='text/css'>
6           <link rel="stylesheet" href="styles.css">
7           <script src="processing.js"></script>
8       </head>
9       <body>
10          <header>
11              <h1>Header</h1>
12          </header>
13              <section id="content">
14                  <canvas id="AsteroidsGame" data-processing-sources="AsteroidsGame.pde Floater.pde Spaceship.pde Star.pde">
15                  </canvas>
16              </section>
17          <footer>
18              Footer
19          </footer>
20      </body>
21  </html>
22
```

- On line 14 of index.html add **`AsteroidsGame.pde`** to the list of files in **`data-processing-sources`**
- The canvas tag should now look like:

```
 <canvas id="AsteroidsGame" data-processing-
sources="AsteroidsGame.pde Floater.pde Spaceship.pde
Stars.pde"> </canvas>
```

Overriding means replacing an inherited function

- If you wrote a different **`show()`** in **`Oddball`** you were ***overriding*** (replacing) the **`show()`** function inherited from **`Ball`**

```
class Oddball extends Ball{
  public void show(){
        ellipse(myX,myY,30,30);
  }
}
```

Vocabulary: Polymorphism

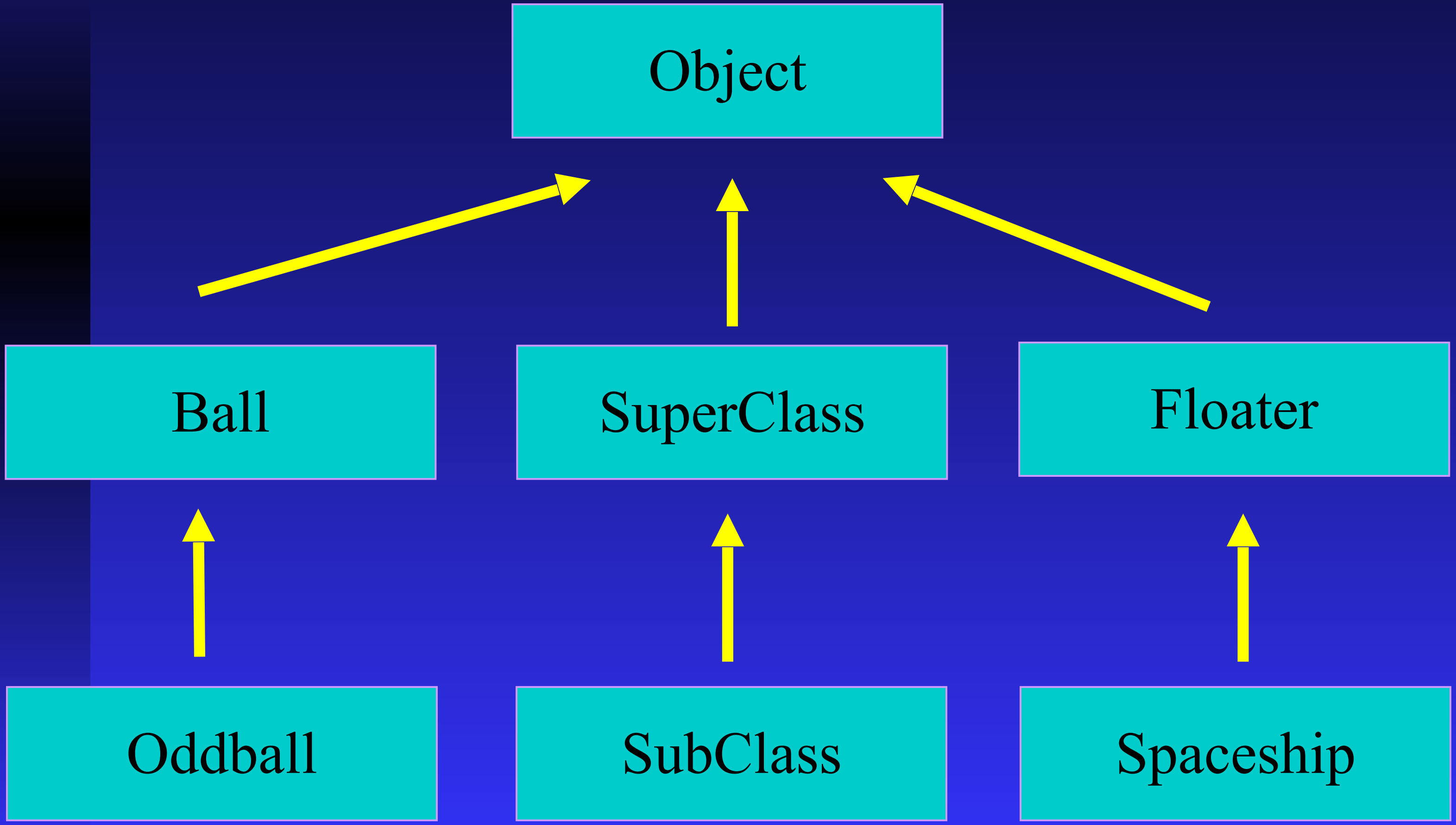
- Polymorphism means ***same name different behavior!***

```
Ball[] balls = new Ball[2];
balls[0] = new Ball();
balls[1] = new Oddball();
for(int i=0;i<balls.length;i++)
    balls[i].show();
```

- The red code is an example of Polymorphism
- `show()` has a *different meaning* for **`Oddball`** than for **`Ball`**

All classes `extend Object`

- The `Object` class is the parent class of all the classes in java by default. In other words, it is the topmost class of java
- `AnyClass instanceof Object` will always evaluate to `true`



`public` member variables

- 99% of the time, member variables are **`private`** and member functions are **`public`**
- Sometimes, though, you might do it the other way around
- Constants are "locked" variables
  - `public final static int LIFE_MEANING = 42;`
- It's fine to make a make constant **`public`** because it can't be changed or "messed up"
- Note that constant variable names are usually ALL CAPITALIZED (Not "mixedCase")

`private` functions

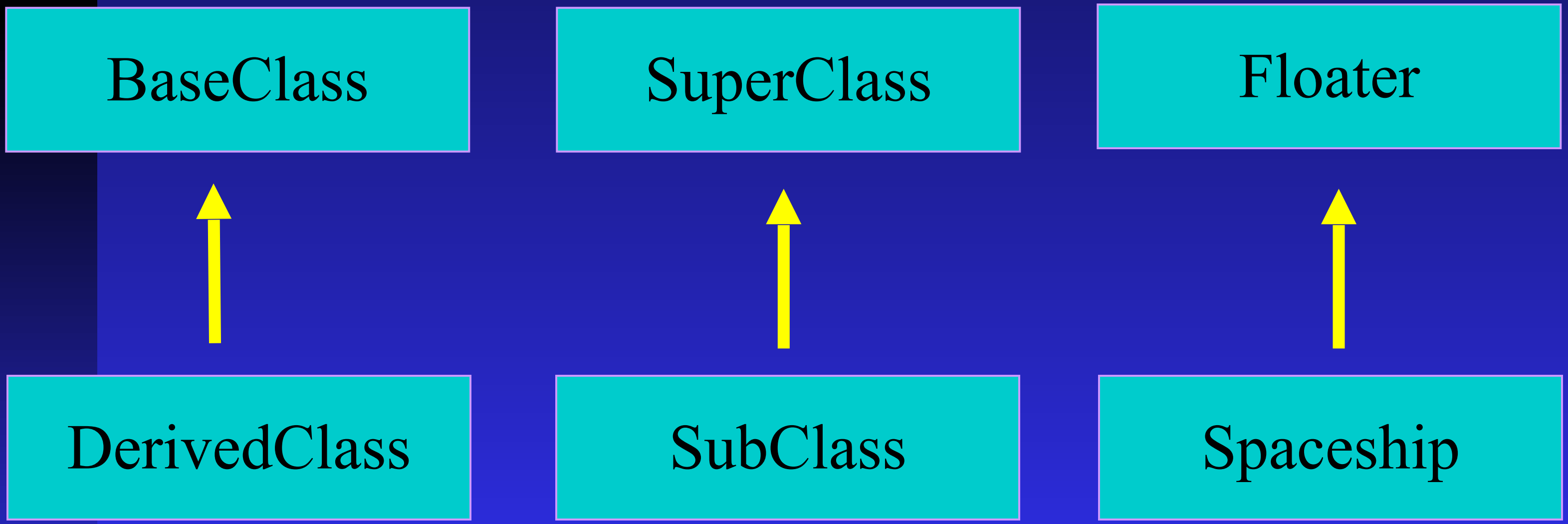
- Functions are **`private`** when the programmer doesn’t want any client programmers (think “team members”) to use them
- A “cheat” might be a good example
- For testing purposes, I might make a function that instantly destroys all the asteroids, but I want to restrict who can use it
- As an option write a private function to do just that and test it!

Inheritance: `super` and sub classes

- The Java keyword **`extends`** is used to create inheritance
- Inheritance means that there are two classes: **`class SubClass extends SuperClass`**
- The sub class inherits everything that the super class has *except constructors*
- The ***most common inheritance mistake is to copy or redefine variables or methods*** that the sub class inherits

Inheritance Diagrams: The arrow is important and always points up

```
class DerivedClass extends BaseClass
class SubClass extends SuperClass
class Spaceship extends Floater
```



Note: the arrow designates the *is-a* relationship. A spaceship *is-a* floater

Super and Sub class constructors

- Constructors are NEVER inherited
- But there is a relationship between the sub class and super class constructors
- The default, *no argument* super class constructor is "invisibly" called when you create a **`new`** instance of the sub class
- Huh?

Making a **new** instance of Subclass

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  public SuperClass() {
     myNum = 3;
     System.out.println(myNum);
  }
  public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

Subclass extends SuperClass

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  public SuperClass() {
     myNum = 3;
     System.out.println(myNum);
  }
  public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

Default no argument constructor runs

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  public SuperClass() {
      myNum = 3;
      System.out.println(myNum);
  }
  public SuperClass(int num1, int num2){
      myNum = num1 + num2;
      System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

Then SubClass constructor runs

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  public SuperClass() {
     myNum = 3;
     System.out.println(myNum);
  }
  public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

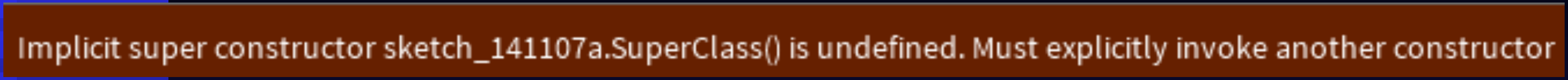
Output is 3 and then 5

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  public SuperClass() {
     myNum = 3;
     System.out.println(myNum);
  }
  public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

What if no default SuperClass constructor?

```
public void setup(){
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  //public SuperClass() {
  //   myNum = 3;
   //  System.out.println(myNum);
  }
 public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

Error!

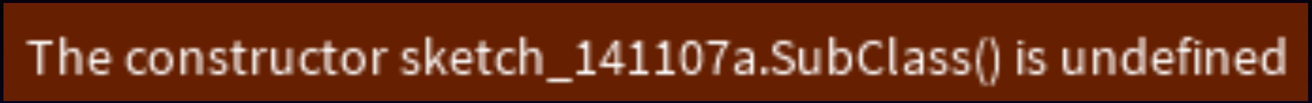

Implicit super constructor sketch_141107a.SuperClass() is undefined. Must explicitly invoke another constructor

```
  SubClass bob = new SubClass(5);
}
class SuperClass{
  protected int myNum;
  //public SuperClass() {
  //   myNum = 3;
  //   System.out.println(myNum);
  //}
  public SuperClass(int num1, int num2){
     myNum = num1 + num2;
     System.out.println(myNum);
  }
}
class SubClass extends SuperClass{
    public SubClass(int num) {
     myNum = num;
     System.out.println(myNum);
    }
}
```

Does the SubClass inherit the default constructor?

```
public void setup() {
  SubClass bob = new SubClass();
}
class SuperClass {
  protected int myNum;
  public SuperClass() {
    myNum = 3;
    System.out.println(myNum);
  }
  public SuperClass(int num1, int num2) {
    myNum = num1 + num2;
    System.out.println(myNum);
  }
}
class SubClass extends SuperClass {
  public SubClass(int num) {
    myNum = num;
    System.out.println(myNum);
  }
}
```

NO!


The constructor sketch_141107a.SubClass() is undefined

```
public void setup() {
  SubClass bob = new SubClass();
}
class SuperClass {
  protected int myNum;
  public SuperClass() {
    myNum = 3;
    System.out.println(myNum);
  }
  public SuperClass(int num1, int num2) {
    myNum = num1 + num2;
    System.out.println(myNum);
  }
}
class SubClass extends SuperClass {
  public SubClass(int num) {
    myNum = num;
    System.out.println(myNum);
  }
}
```

```
public void setup(){
  BaseClass bob = new BaseClass(4,5);
  DerivedClass notBob = new DerivedClass(6);
}
class BaseClass{
  protected int myNum;
  public BaseClass() {
    myNum = 3;
    System.out.println(myNum);
  }
  public BaseClass(int num1, int num2){
    myNum = num1 + num2;
    System.out.println(myNum);
  }
}
class DerivedClass extends BaseClass{
  public DerivedClass(int num) {
    myNum = num;
    System.out.println(myNum);
  }
}
```

Practice Quiz Question: Find 3 lines of output

Hint: `new DerivedClass(6)` prints ***2 lines*** of output

```
public void setup(){
            SuperClass bob = new SuperClass(5);
            SubClass sue = new SubClass();
            System.out.println(bob.myInt);
            System.out.println(sue.myInt);
            System.out.println(sue.mySecondInt);
}
class SuperClass{
      int myInt;
      SuperClass(int nInt){
            System.out.println("Building a Super");
            myInt = nInt;
      }
}
class SubClass extends SuperClass{
      int mySecondInt;
      SubClass(){
            super(3);
            System.out.println("Building a Sub");
            mySecondInt = 4;
      }
}
```

What is the output?

More on `super()`

- Every time a new instance of a *sub* class is created, we start by building a *super*

```
class SubClass extends
  SuperClass{…}
```

```
SubClass bob = new SubClass ();
```

More on `super()`

- On the very first line of the **`SubClass`** constructor, its as if there is an "invisible" call to `super();`

```
class SubClass extends SuperClass{
   public SubClass(){
         super(); //invisible
         //lots more java
}
```

More on `super()`

- The call is to the default, no argument constructor
- If we want to call a different version of the constructor, one with arguments, we need to do it on the very first line of the constructor

```
class SubClass extends SuperClass{
    public SubClass(){
        super(4);//must be first!
        // any other code after
}
```

What happens if a sub class doesn’t have a constructor?

- Does it inherit the constructor from the super class?

What happens if a sub class doesn’t have a constructor?

- Does it inherit the constructor from the super class?
- NO NO NO!
- Java generates an "invisible" constructor that calls ***super()***

```
class SubClass extends SuperClass{
   public SubClass(){//invisible
        super();
   }
   //lots more Java
}
```

Every class `extends Object`

- In Java, every class has an "invisible" **`extends Object`**
- **`Object`** is the ultimate "base" or "super" class for all other objects
- That means, all other classes are “derived" or "sub" classes of **`Object`**
- This "generic" data type lets us write methods that take any sort of Object as an argument

Why Java doesn't allow `super.super`

- You can only "go up one level" with **`super`**. Java won't let you use **`super.super`**
- For **`someClass`** what class would **`super`** refer to?
- What class would **`super.super`** refer to?

```
class someClass extends Object  //invisible
{
  //code not shown
}
```

What is the output?

```
public void setup()
{
   SubClass bob = new SubClass();
   bob.mystery();
   System.out.println(bob.getInt());
}
class SuperClass
{
   protected int myInt;
   public SuperClass(int nInt) {myInt = nInt;}
   public void mystery(){myInt *= 2;}
   public int getInt(){return myInt;}
}
class SubClass extends SuperClass
{
   public SubClass() {super(2);}
   public void mystery(){
      myInt++;
      super.mystery();
    }
}
```