CS 2113
Software Engineering

Java 4: Class Organization, Abstraction

Use IntelliJ to “Check out from Version Control” this git repo: **https://github.com/cs2113f18/template-j-4**

Click **Yes/Next** until the project opens
Open code for **drawing.MyDrawing**
Run and edit the code
Play.

Professor Tim Wood - The George Washington University

This Time...

- Project 1: how was it?
- More OOP Concepts
  - Abstract Classes
  - Polymorphism
  - Introspection
  - Interfaces
- Also
  - Project 2

So Long CodeAnywhere…

- Sadness? Tears? of joy?

IntelliJ IDEA

- Integrated Development Environment (IDE)
- Will make some parts of your life easier
- Can be a bit overwhelming
- Allows us to build more interesting programs
  - Can create windows, play sounds, send data over network...
- Teaches you about a more realistic development environment
  - Your future job may use something different, but the principles will be the same

* This is not actually a quiz

Java Quiz*!

- **Put code in the `animals` package!**
- Store two types of pets---cats and dogs
  - When you create a pet, constructor takes a name
  - All pets have a printName() function that prints the name
  - All pets have a makeNoise() function
    - Cats say "meow" and dogs say "woof"
- Your main method should:
  - Create two dogs named Fido and Spot
  - Create three cats named Fluffy, Mowzer, and Pig
  - Use **ONE** ArrayList to store all 5 pets
  - Print the names of all pets
  - Call the makeNoise function on all the pets

Files and Collections

- Let's:
  - Read all lines in a file
  - Add each line to an Array List
  - Print out a random entry from the array list
- Modify **files.RandReader.java**

Hierarchies and Abstraction

Use the benefits of OOP

- Use a super class to store common functionality

- Why?

Use the benefits of OOP

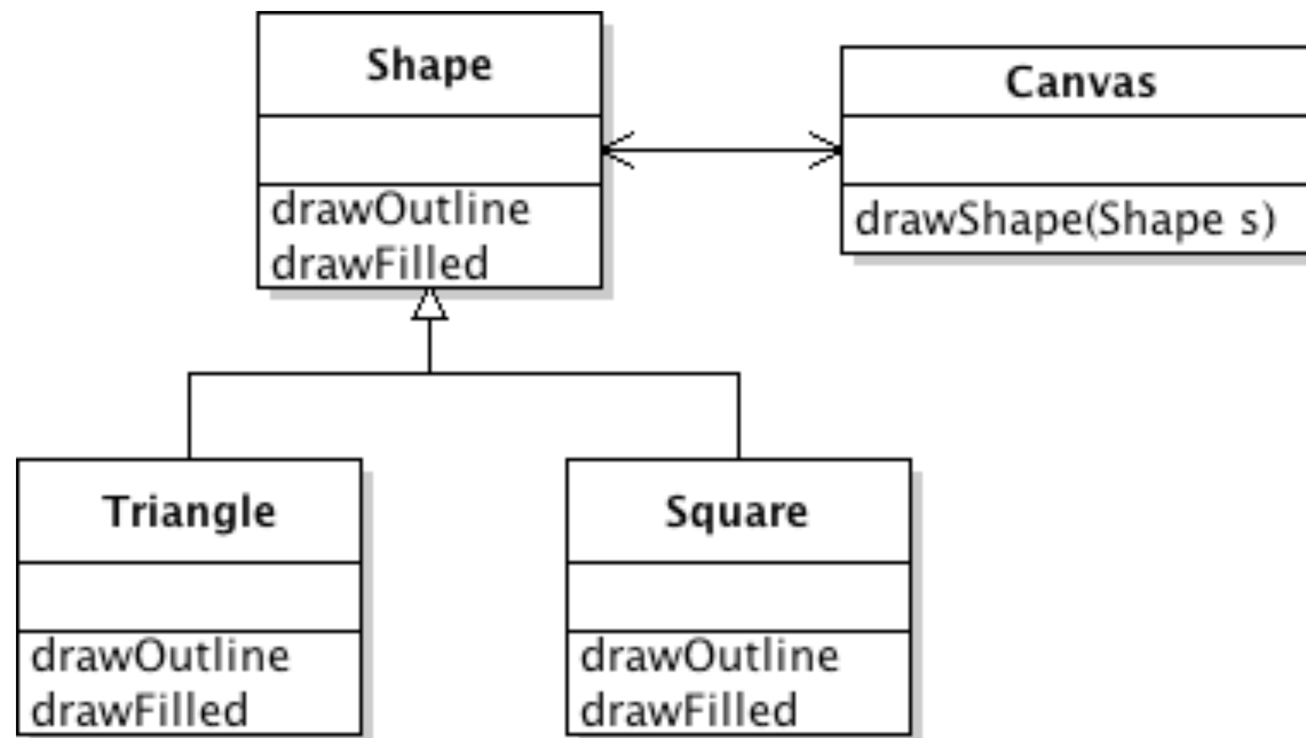
- Use a super class to store common functionality
- Why?
  - **Code reuse** - no copy/paste
    - What if you need to add an "age" field to all pets?
  - **Polymorphism** - treat similar objects the same way

```
ArrayList<Pet> list = new ArrayList<Pet>();
list.add(new Cat("Fluffy", 9));
list.add(new Dog("Fido"));

for(Pet s: list) {
   s.makeNoise();
}
```

Abstraction

- Sometimes it doesn't make sense to implement the functions in a class
  - Would we ever want to instantiate a Pet object?



- What would go in Shape's functions?
- **Abstract classes** define the structure of a class, but not its actual implementation

Abstract Classes

- Mark class and methods with abstract keyword
  - No function body for abstract methods
  - Class can still have some real data and methods
- Child classes must implement all abstract methods
- You can never instantiate an abstract class

```
public abstract class Shape {
  public abstract void drawOutline();
  public abstract void drawFilled();
}
```

```
public class Triangle extends Shape {
  public void drawOutline() { ... }
  public void drawFilled() { ... }
}
```

Drawing Shapes

- Look at the **shapes** package
  - What is the class hierarchy?
- Create an ArrayList and put a Circle, Rectangle, and Square into it
- Draw the filled version of each shape to the screen with a for loop
  - Get each shape out of the list and then call its drawFilled()
- Add some more shapes to create a beautiful work of art

Class Hierarchies

- **Look at the "dumbshapes" package**

- Why is this dumb?

Polymorphism

- Why does this work?

```
ArrayList<Shape> list = new ArrayList<Shape>();

list.add(new Circle(10, 10, 5, Color.blue));
list.add(new Rectangle(10, 5, 3, 6, Color.RED));
list.get(0).drawFilled();
```

- but not this?

```
ArrayList list = new ArrayList();

list.add(new Circle(10, 10, 5, Color.blue));
list.add(new Dog("Fido"));
list.get(0).drawFilled();
```

Java is "strongly typed"

- The JVM knows the type (class) of each object
- It enforces rules based on those types

- At ?????? time it will decide if your code calls functions that a type does not support

```
ArrayList list = new ArrayList();

list.add(new Circle(10, 10, 5, Color.blue));
list.add(new Dog("Fido"));
list.get(0).drawFilled();
```

- The array holds items of type Object
  - That class doesn't have a drawFilled function!

Casting

- Casting objects does let us get around type rules:

```
// In package dumbshapes
ArrayList list = new ArrayList();

list.add(new Circle(10, 10, 5, Color.blue));
list.add(new Rectangle(10, 5, 3, 6, Color.RED));
list.add(new Square(4, 6, 5, Color.GREEN));

((Circle) list.get(0)).drawFilled();
((Rectangle) list.get(1)).drawFilled();
((Square) list.get(2)).drawFilled();
```

- What happens if we cast to the wrong type?

```
((Circle) list.get(2)).drawFilled();
```

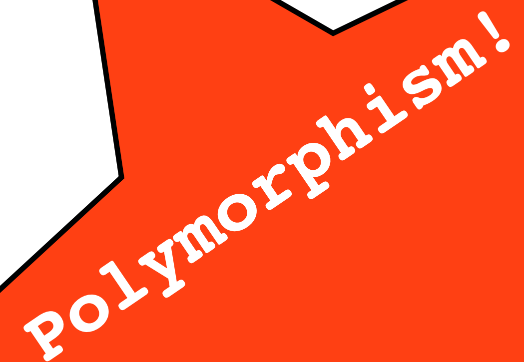
But Remember:

- An object can do everything that its parent can do!

```
ArrayList<Shape> list = new ArrayList<Shape>();

list.add(new Circle(10, 10, 5, Color.blue));
list.add(new Rectangle(10, 5, 3, 6, Color.RED));
list.get(0).drawFilled(); // OK since Circle's are Shapes
```

- What about the opposite?
  - Is a Shape a Circle?
  - Is a Square a Rectangle?
  - Is a Rectangle a Square?




```
ArrayList<Shape> list = new ArrayList<Shape>();
list.add(new Circle(10, 10, 5, Color.blue));
int r = list.get(0).radius; // Is this OK?
```

Introspection

- Polymorphism must know class of each object
- Introspection allows you to ask questions about an object or class
- **`instanceof`** operator asks if an object is part of a particular class

```
for(Shape s: shapes) {
 if(s instanceof Rectangle){
    s.drawOutline();
 }
 else {
  s.drawFilled();
  if(s instanceof Circle) {
     r = ((Circle)s).radius;
 }
}
```

What happens if I have a Circle, Rectangle, and Square?

Organizing a zoo

- Suppose we have a program about animals...
  - Cats, dogs, wolves, bears, lions, unicorns, etc
- They do things:
  - eat
  - roam
  - make noise
- What classes and functions do we need?
  - How would you organize them?

Consider these animals...

GrizzlyBear

PandaBear

CutePuppy

- How would they fit into a class tree?
- Pandas and Puppies are both cute… :(

Multiple Inheritance

- What if it makes sense for a class to inherit from two parent classes?
  - Java does not allow you to extend multiple classes
- Use an **Interface**
  - Looks like an abstract class
  - List of **functions** that must be implemented
  - **Cannot** include data!

```
public interface Cuddly {
  public void snuggle();
}
```

```
public class PandaBear extends Bear implements Cuddly {
  public void snuggle() { ... }
  // ...
}
```

Why use an interface?

- You can only have one parent
  - But you can **`implement`** many interfaces
- Useful when:
  - Some subclasses **do not** implement a function
  - Objects from several classes **do** implement a function
- **`Vegetarian`** interface implemented by:
  - Brontosaurus (child of Dinosaur)
  - Koala (child of Marsupial, also implements Cuddly)
  - Hindu (child of Human)

Animals and pets

- Support as many of these animals as possible:

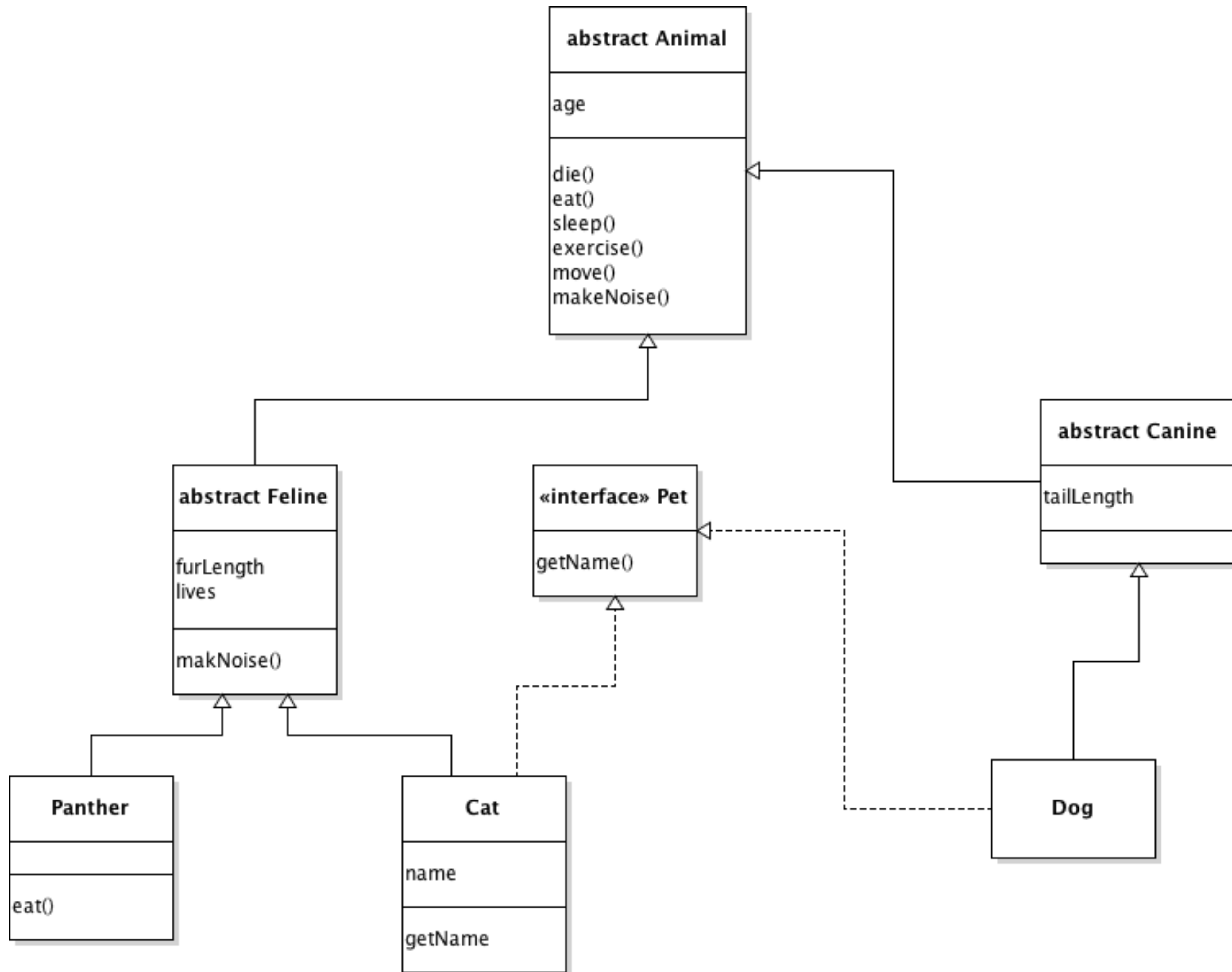
```
Cats, dogs, wolves, bears, lions,
unicorns, parrots, grizzly bears,
panda bears, pigeons, cuddly puppies,
panthers, horses, talking bears.
```

- Add code to the **animals** package
- Make them do interesting things
  - Pets have names
  - Cuddly animals snuggle
  - Felines all roar
  - What else?

```
In groups of at
least 2!
```

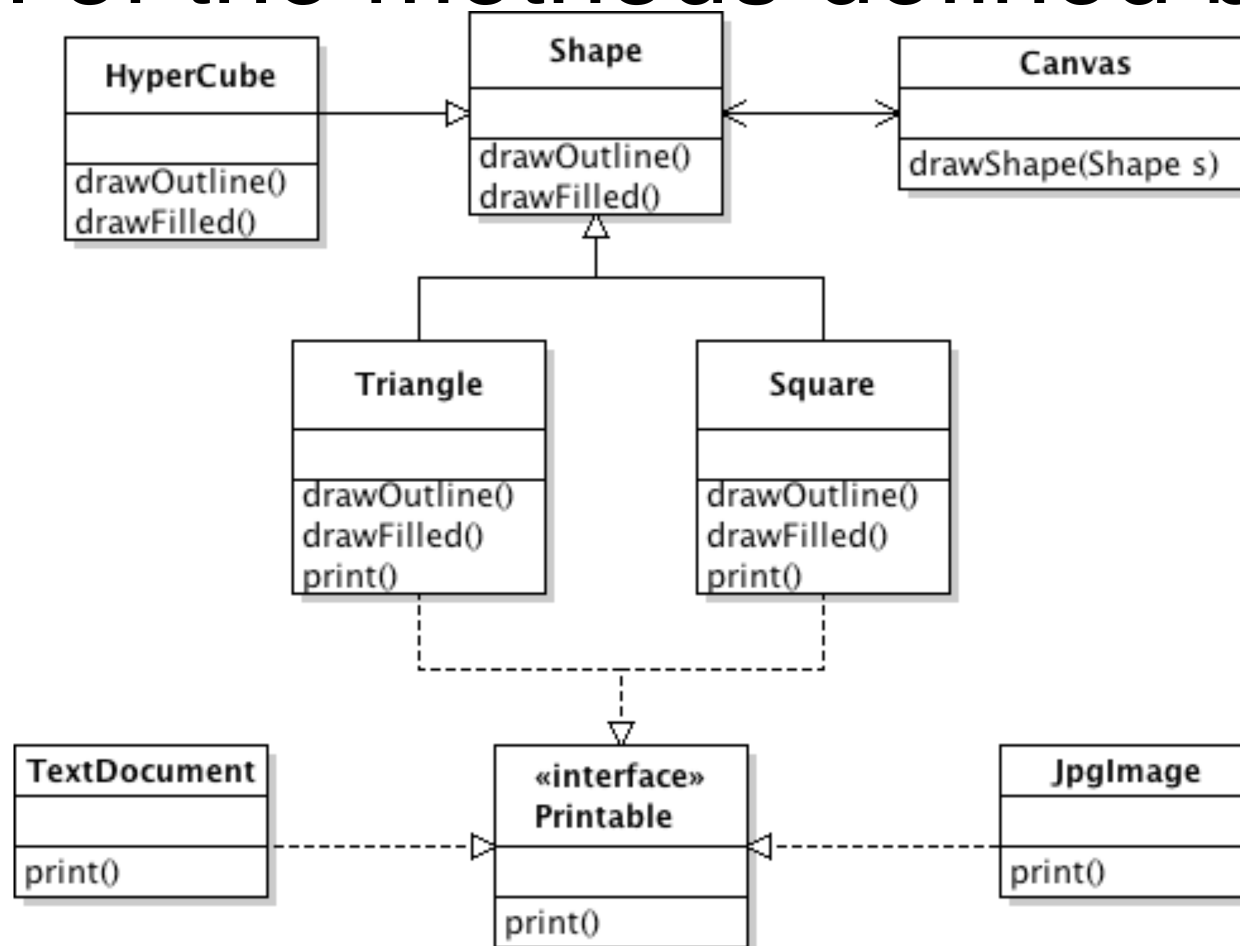
- **Abstract classes?** **Interfaces?**

Some animals



An Interface is a Contract

- If you implement an interface, you promise to support all of the methods defined by the interface



- Why is this useful???

Interfaces & Polymorphism

- Polymorphism lets us treat all classes that:
  - Implement the same interface
  - Are children of the same parent
- As if they *are* that parent or interface

```
public void  main()
{
  ArrayList<Printable> printme = new ArrayList<Printable>();
  printme.add( new Triangle() );
  printme.add( new Square() );
  printme.add( new PdfDocument() );
  printme.add( new TextDocument() );
  for (Printable p : printme) { p.print(); }
}
```

Abstract and Interface

- Question 1:
  - Can an abstract class have data members? Can an interface?
- Question 2:
  - Can you include the body of a function in an abstract class? In an interface?
- Question 3:
  - What happens if a subclass does not implement one of the methods in an abstract parent or an interface?
- Question 4:
  - Can you instantiate an object of an abstract type? an interface?

Interfaces for Sorting

- Sorting is a very common requirement
- How do you sort:
  - Numbers
  - Letters
  - Names
  - Animals
  - Customers
- Basic operation in any sorting algorithm:
  - Is element **A** higher or lower than element **B**?

Comparable Interface

- Implement the **`Comparable`** Interface to define how to compare instances of a class
- Allows you to use a generic sorting function

```
List<Name> names = new ArrayList<Name>();

// add elements to list

Collections.sort(names);
// list is magically sorted!
```

- Must implement the CompareTo(b) function
  - Return 0 if identical
  - Less than 0 if `this < b` or greater than 0 if `this > b`

Sorting Students

- Look at the code in the "interfaces" package
- **Student**: stores name and GPA
- **StudentSort**: adds a few names to a list, tries to sort
  - Uses **Collections.sort()**

- To allow a list of Names to be sorted, you must implement the **Comparable<Student>** interface

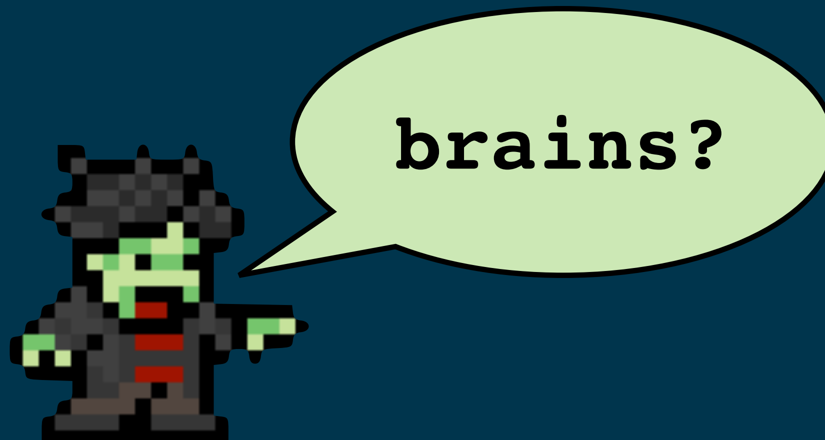
- Add code to implement **CompareTo<Student>**
  - Sort students by GPA
  - Challenge: Use last name and then first name as tie breakers
    - String already supports the compareTo() function, so you can use that as a base!

Summary

- Abstract classes
  - Define structure of subclasses and force them to implement complete behavior
- Interfaces
  - Define a list of functions that the implementor of an interface must include
  - One class can implement multiple interfaces
- Ways to group similar classes and enforce what they define

Project 2...

Zombie Infestation Simulator



Zombie Sim Structure

- ZombieSim
  - main()
  - instantiates city
  - loop: update city and draw
- City
  - private Walls[][]
  - update
  - draw
  - populate()
  - what else to add???

Tips/Best Practices:

- Think carefully about class structure and the data and functions in each one
- Think carefully about the "is a" versus "has a" relationship when designing your classes
- It is always better to have a class interact with another using an API (functions) instead of directly accessing data
- Use classes to encapsulate both data and functions. A City class should be responsible for everything to do with the city and a Cat class would be responsible for everything to do with cats, etc.