Polymorphism

- One of the aspects of object-oriented languages which makes it more than just structured programming is the ability to extend existing things to create new classes.
- A term that comes up often in describing some of the benefits and properties of polymorphism is "generic" (referring to data types as well as functions).
- There are different categories of polymorphism that can be discussed, and different object-oriented languages support polymorphism in a variety of ways.
Java Interfaces

• The first thing we will consider is the notion of an "interface" in Java.
• In Java, an interface establishes (for example) a list of methods that any class which implements that interface must contain.
• If we implement several classes which implement the same interface, we get an added ability; we can create a reference using the interface's name and have it point to any of the classes that implement that interface.

Interface: Animal.java

```java
public interface Animal {
    public String getName();
    public void setName(String s);

    public String makeSound();

    public String toString();
}
```
Class: Cat.java

```java
public class Cat implements Animal {
    private String animalName;

    public Cat(String nameIn) {animalName=nameIn;}
    public String getName() {return animalName;}
    public void setName(String s) {animalName=s;}

    public String makeSound() {return "meow";}
    public String toString() {return animalName;}
}
```

Designing Interfaces

- Once agreed upon, this is a VERY costly thing to change, so make sure it is well thought out.
- Think long-term and make sure to only put things that REALLY should be required in there.
- In addition to method signatures, interface definitions can contain public static constants.
- NOTE: When creating a new class, it can implement more than one interface.
A Heterogeneous Array

- **Cats** and **Dogs**, living together?
- Yes, if we create an array of **Animal** references and then allocate a **Cat** object or a **Dog** object as desired.

Methods not in the interface…

- What if we wanted our **Cat** class to have methods not defined in the interface?
- What if we wanted our **Dog** class to have methods not defined in the interface?
- What if the extra methods in the **Cat** class and the extra methods in the **Dog** class aren't the same as each other?

- Casting!
Object super class

- All classes in Java are automatically based on the Object super class.
- This allows us to be able to generically pass any reference into a method as a parameter by making the parameter of type Object.
- In order to access anything other than the methods that are defined within Object, we would need to cast the parameter to the correct type.
  - Note: You can ask any object to identify its type via the getClass() method that is automatically created for every class we create.

Comparable

- Any class implementing the comparable interface must correctly implement:
  
  int compareTo(Object);

- If we are going to use things like Arrays.sort() on things we should probably make sure our objects implement Comparable.

- Let's create a ComparableCat class...

  ```java
  public class ComparableCat implements Animal, Comparable {
    ...
  }
  ```
equals and compareTo

- The "official" signatures of equals and compareTo are:
  ```java
type compareTo(Object);
boolean equals(Object);
```

- Other Java code might be expecting methods that have these signatures.
  - It's why Comparable needed it.
  - It's why AssertEquals wouldn't work on our self-created types (until now).

Wrappers

- For flexibility, wrappers were created for each primitive data type in Java, such as Integer, Float, Double, etc.
- We have written wrappers of sorts too (such as the ComplexNumber class) which we could go back and "fix" to be in compliance on things like equals and have it implement the Comparable interface (and write the appropriate compareTo).
- We can either add functionality or restrict functionality via wrappers. What would the MutableInteger wrapper do?
Numeric Wrappers

• For things like numbers, if you have a group of them, you can ask questions like "which is the smallest?".

• All of our primitive numeric types have "wrapped" versions available to us.

• We could write a single generic method called `minimum()` that takes an array of comparable numbers (or any other comparable objects for that matter) and returns the smallest in the array.