CMSC131

Polymorphism and Interfaces

Polymorphism

We've seen that one of the aspects of object-oriented languages which makes it more than just structured programming is the ability to have general-use data structure classes like the **`ArrayList<T>`**.

A term that has come up in describing some of the benefits and properties of parametric polymorphism is "generic" when referring to data structure types.

There are different categories of polymorphism that can be discussed, and different object-oriented languages support polymorphism in a variety of ways.

Homogenous Data Structures

With the **`ArrayList<T>`** data structure, one of the limitations that we have discussed is that although we can ask for an **`ArrayList`** for any object type we choose, the references that we add to the list must match that type.

This means that if we were to create an **`ArrayList<Rational>`** we cannot add a reference to a **`CubicPoly`** or to a **`Long`** to it.

Heterogeneous Lists

- How often would you say you make or use lists that are heterogeneous?
- Do you think there would ever be a logic to having a Rational and CubicPoly in the same list?
- Do you think there would ever be a logic to having a Rational and Long in the same list?

Java Interfaces

We now look at something called *subtype polymorphism* as we consider the concept of an **interface** in Java.

In Java, an `interface` is used to establish (for example) a set of `public` methods that any class saying it `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 an object of ***any*** of the class types that implement that interface.

Example: `Comparable<T>`

An interface that is defined by the standard Java libraries is called `Comparable<T>`.

A class that implements this interface has to provide a public method `int compareTo(T otherObject)` that behaves the "appropriate" ways.

We've seen this method before, and it can be part of any class, but if the class states that it implements **Comparable**, it has to and we can get advantages…

Arrays.sort(array of objects)

Java has a utility class named **Arrays** that contains a variety of general-purpose methods for manipulating arrays, assuming they either contain primitive types or objects with certain properties.

One such potentially useful method is **sort()** that takes any array of an object type as long as that type implements the **Comparable** interface.

Making `Rational` a Comparable type

```
public class Rational
             implements Comparable<Rational> {
   :
   :
   @Override
   public int compareTo(Rational otherObject) {
       return
        this.subtract(otherObject).getNumer();
   }
}
```

That might be all we need to do! Is it? →

Try it out with your Lab06 later…

```
public static void main(String[] args) {
     Rational[] list = new Rational[7];

     list[0] = new Rational(7,11);
     list[1] = new Rational(8,11);
     list[2] = new Rational(9,11);
     list[3] = new Rational(10,11);
     list[4] = new Rational(7,10);
     list[5] = new Rational(7,9);
     list[6] = new Rational(7,8);

     Arrays.sort(list);

     for (Rational var : list) {
            System.out.print(var + "   ");
     }
     System.out.println();
}
```

Creating our own interface…

We might want to support a certain amount of heterogeneous behavior among a set of classes we are designing.

The easiest way to support this might be by creating an interface of our own.

Imagine having a variety of animals to support, yet wanting to be able to have a single **`ArrayList`** or array that contains different animals within it…

Interface: Animal.java

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

   public String makeSound();

   public String toString();
}
```

Class: Cat.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;}
}
```

Use of `@Override`

A somewhat common convention with methods connected to an interface being implemented is to make use of the Java "annotation type" text **`@Override`** above it.

- It is not required and causes no differences behind the scenes in terms of bytecode.
- It is a way to avoid careless typos since it will cause a compilation error if the signature of the method being written doesn't match the signature of a previously-mentioned method.
- You may have noticed this annotation in some code we provided (you will see it more in 132).

Designing Interfaces

Once agreed upon, changing an interface can be a fairly time-costly process, so make sure it is well thought out.

- Think long-term and make sure to only include things that really should be ***required*** to be in there.

In addition to method signatures, interface definitions can contain **`public static`** constants (note: you don't actually use the words **`final`** or **`static`** in the interface definition).

When creating a new class, it can implement more than one interface if you want to.

A heterogeneous array or ArrayList

Could we have **`Cats`** and **`Dogs`** together in the same array or **`ArrayList`**?

Yes, we can, if we create an array or an **`ArrayList`** of **`Animal`** references and then have each position refer to an allocated **`Cat`** object or **`Dog`** object as desired.

Methods not in the interface…

What if we wanted our **`Cat`** and/or **`Dog`** classes 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?

Answer: Casting! We can use **`(type)`** casting to specify what the object's actual data type is and then we can dereference it using that to gain access to non-interface methods.

Which of the following would be legal?

0% 1. Animal x = new Animal("Lucky");
0% 2. Animal x = new Cat("Lucky");
0% 3. Animal x = new Dog("Lucky");
0% 4. Cat x = new Cat("Lucky");
0% 5. Cat x = new Dog("Lucky");

Response Counter

Which types of object could be passed into an Animal parameter?

0% 1. Animal
0% 2. Dog
0% 3. Cat

Response Counter

Comparable Animals?

What if you wanted all of your animals to also be comparable? In Java, a newly-defined interface can extend an existing one.

We could change the Animal interface

```
public interface ComparableAnimal extends
                Comparable<ComparableAnimal> {
```

Then each class that implements this would need the appropriate **`compareTo`** method.

Implementing Multiple Interfaces

What if you only wanted *some* of your animals to also be comparable? Have the **`Animal`** interface we had at the start of the slide set and define **`ComparableCat`** as:

```
public class ComparableCat
  implements Animal, Comparable<Animal> {
  …
  @Override
  public int compareTo(Animal other) {
    return
     this.getName().compareTo(other.getName());
  }
}
```

Changes that happened in Java 8

One of the big changes that was introduced in Java 8 is being allowed to have a default implementation of a method within the interface definition itself.

- Previously, when it came to methods, the interface was entirely about setting requirements for classes that wanted to implement that interface.

One advantage to this is that if you decide to add a new method to the list of required ones, you can also provide a default action so older classes will still compile and run.

- The challenge here is being able to write a default that "makes sense" for any classes that implement the interface but not that particular method.

Interface: `AnimalJ8.java`

```
public interface AnimalJ8 {
   public String getName();
   public void setName(String s);

   default public String makeSound(){
      return "um";
   }

   public String toString();
}
```

Class: CatJ8.java

```
public class CatJ8 implements AnimalJ8 {
   private String animalName;

   public CatJ8(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;}
}
//Nothing really changes other than our naming
//   of things with J8.
```

Class: MartianJ8.java

```
public class MartianJ8 implements AnimalJ8 {
   private String animalName;

   public MartianJ8(String nameIn) {animalName=nameIn;}
   public String getName() {return animalName;}
   public void setName(String s) {animalName=s;}

   //Note that we didn't implement the makeSound() method.
   //  It will use the default version.

   public String toString() {return animalName;}
}
```

What will be printed?

```
AnimalJ8[] pets = new AnimalJ8[4];
pets[0] = new CatJ8("Neko");
pets[1] = new DogJ8("Fluffy");
pets[2] = new CatJ8("Crookshanks");
pets[3] = new MartianJ8("Marvin");

AnimalJ8 temp;
for (int i=0; i<pets.length; i++) {
     temp = pets[i];
     System.out.println(temp.getName() +
          " says " + temp.makeSound());
}
```

Things interfaces still can't do…

Enforce that constructors are being written at the class level by the implementing class (since the name of the constructor is the same as the name of the class).

Define *instance fields within the class*.

Define *private* static fields within the class.

The `Number` Class

In CMSC132 you might see that Java provides something called `Number` that other numeric classes can extend (all of the Java numeric wrappers do this). This is different than interfaces but is part of subtype polymorphism.

If we wanted `Rational` to as well, we would need to do this as well and provide the proper methods to convert the rational value to **byte**, **double**, **float**, **int**, **long**, and **short**. We could then have an **ArrayList<Number>** that held a **Rational** and a **Long**.