CMSC131

Interfaces, Object, and Wrappers

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 aspect of polymorphism we will consider is the notion of an "interface" in Java.

In Java, an **`interface`** establishes (for example) a set of **`public`** 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**

```
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;}
}
```

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: 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

`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?

Answer: Casting! We can use **`(type)`** casting to specify the actual data type, and then dereference it.

Which of the following would be legal?

1. `Animal x = new Animal();`
2. `Animal x = new Cat();`
3. `Animal x = new Dog();`
4. `Cat x = new Cat();`
5. `Cat x = new Dog();`

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

1. Animal
2. Dog
3. Cat

`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 (this method 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`**.

To create a **`ComparableCat`** class…

```
public class ComparableCat implements
  Animal, Comparable {
…
}
```

equals and **compareTo**

The "official" signatures of **equals** and **compareTo** are:

```
int compareTo(Object);
boolean equals(Object);
```

Other Java code might be expecting methods that have these signatures.

- It's why **Comparable** needed that signature.
- It's why **AssertEquals** is a potential problem with self-created types (until now).

Why wrappers?

- For flexibility, **`Object`**-based 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 **`Rational`** 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 a **`MutableInteger`** wrapper do?

Numeric Wrappers

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

All of our primitive numeric types have **`Object`**-based "wrapped" versions that implement **`Comparable`** available to us.

We could write a single generic method called **`minimum(Comparable[] arr)`** that takes an array of **`Comparable`** values and returns the smallest in the array (as long as we make sure the data types are valid to compare).