Thursday, September 6, 12
1

Announcements

- Lab 0 is due on Sunday at 8pm.
- Lab helper hours are posted on the website and on all the lab doors.
- If you have trouble with your account, see Chris Mohler in King 125 TODAY.
- Prelab 1 is due on Monday by the beginning of class (hand in in class.)

I have office hours today from 3:30-4:30 if you want to stop by and introduce yourself.

Friday 1 is on Monday by the beginning of class (hand in class.)

If you have trouble with your account, see Chris Mohler in King 125 TODAY.

Lab helper hours are posted on the website and on all the lab doors.

Lab 0 is due on Sunday at 8pm.

CS 151
Suppose we want classes to describe various shapes. Every shape has a color:

```java
public class Shape {
    private Color c;
}
```

Every shape should have methods to calculate its area and perimeter. We can't add their implementations to the `Shape` class, though, because their calculations depend on the specific shape:

```
4. match exact signature in implementation (otherwise you're not overriding)
3. abstract methods must be public
2. abstract classes (X interfaces only)
1. IS-A relationship holds (X implements Comparable, X implements Comparable)
```

We can add the methods to the `Shape` class, have them return a bogus value. We would have to make sure that the `Square` and `Circle` classes both override these methods, otherwise we'll return the bogus value.

But what if someone forgets to override the methods? Not good. We want to force them to override.

We could remove the method from the `Shape` class and hope that `Square` and `Circle` remember to implement those methods. What's wrong with this solution?

We can't have, for example, an array of shapes, then call `area` on each one.

```
5. cannot have abstract class to compile (previous program errors)
```

**Abstract class** - a class with at least one abstract method

```
abstract class -- a class with at least one abstract method
-- abstract classes cannot be instantiated (Shape s = new Shape();)
abstract methods cannot be implemented (Shape s = new Shape();)
```

**Abstract method** - use abstract keyword with method signature only to indicate that the method is just a placeholder; must be implemented by all child classes

```
abstract method -- use abstract keyword with method signature only to indicate that the method is just a placeholder; must be implemented by all child classes
```

When a class implements an interface:

```
1. IS-A relationship holds (X implements Comparable, X IS-A Comparable)
2. instanceof works (X instanceof Comparable)
3. interface methods must be public
4. match exact signature in implementation (otherwise you're not overriding)
5. cannot have abstract class to compile (previous program errors)
```

**Interface** - “the ultimate abstract class”

```
1. all interface methods are public abstract methods, by default
2. all fields are public static final, by default
public interface Comparable {
    int compareTo(Object other);
}
```

Suppose class `X` implements the `Comparable` interface. Then:

```
1. IS-A relationship holds (X implements Comparable, X IS-A Comparable)
2. instanceof works (X instanceof Comparable)
3. interface methods must be public
4. match exact signature in implementation (otherwise you're not overriding)
5. cannot have abstract class to compile (previous program errors)
```
What are Generics?

Do you remember ArrayLists and their strange, strange syntax?

ArrayList<String> stringList = new ArrayList<String>;
ArrayList<Point> pointList = new ArrayList<Point>;

That is, you could create an ArrayList out of any reference (non-primitive) type. Then, you could use all the ArrayList methods to manipulate the objects of your chosen type:

stringList.add( "blah blah blah" );
pointList.set(0, new Point(0,5));
Point p = pointList.get(0);

So, somehow, ArrayLists were written to work with any given reference type.

When you construct an ArrayList, you have to specify its type. From that point onwards, you have to use that type (or, derived types) for that list's methods.

How Can We Do This Ourselves?

We can use this technique in our own classes, with some extra syntax.

public class GenericData<AnyType> {
    private AnyType data;
    public AnyType getData() { return data; }
    public AnyType setData(AnyType data) { this.data=data; }
}

Restricting the Placeholder

What if we want to restrict the placeholder a little?

public class DinoData<AnyType extends Dinosaur> {
   DinoData<Velociraptor> dd = new DinoData<Velociraptor>();
   DinoData<Point> dd = new DinoData<Point>();
}

For example, we can restrict AnyType to derive from the Dinosaur class:

But this will not:

When we write a new class, we define a generic class parameter:

We can use this technique in our own classes, with some extra syntax.

Restricting the Placeholder

We may need to restrict things even further:

public interface Comparable<AnyType> {
   public int compareTo( AnyType other );
}

So, suppose our Dinosaur class implements Comparable<Dinosaur>:

public class Dinosaur implements Comparable<Dinosaur> { .. }

public class Data<AnyType extends Comparable<AnyType>> {
   // A lot of sorting methods are defined on types that implement comparable;
   // consider the common and comparable interface:
}

A lot of sorting methods are defined on types that implement comparable;

Consider the commonly used Comparable interface:

Then our Comparable class extends Dinosaur:

Dinosaur extends Animal;

We can use this technique in our own classes, with some extra syntax.

What are Generics?
Restricting the Placeholder

So, suppose our Dinosaur class implements Comparable\<Dinosaur\>.

```java
public class Dinosaur implements Comparable\<Dinosaur\> {}
```

Then our Velociraptor class, that extends Dinosaur, also implements Comparable\<Dinosaur\>.

Then consider the following Data class:

```java
Then the following statement will not work:

Data\<Velociraptor\> vData = new Data\<Velociraptor\>;
```

The problem is that Velociraptor extends Comparable\<Dinosaur\>, not Comparable\<Velociraptor\>. But, if you can compare Dinosaurs, you can certainly compare Velociraptors, because a Velociraptor IS-A Dinosaur.

```java
public class Data\<T extends Comparable\<? super T\>> {}
```

So in fact, we want our Data class to accept any class T that implements the Comparable interface on any superclass of T. We can specify this as follows:

```java
Data\<Integer\> myData = new Data\<Integer\>();  // OK
Data\<int\> myPoints = new Data\<int\>();        // not OK
AnyType myVar = new AnyType();               // not OK
AnyType[] array = new AnyType[10];           // not OK
AnyType[] array = (AnyType[]) new Object[10]; // OK
```

However, you can always use the provided wrapper classes.

```java
Data\<Data\<Integer\>\> myData = new Data\<Data\<Integer\>\>();  // OK
```

1. Cannot use generics for primitive types.
2. Cannot construct instances of a generic type.
3. Cannot create an array of generics.

However, you can create an array of primitive types.

```java
int[] array = new int[10];  // OK
```

This will give a compiler warning that you can fix by adding

```java
@SuppressWarnings("unchecked")
```

right above the method body (and after the javadoc comments).