CS 151

Inheritance Review & Generics

slides posted on course website
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.
The Object class is the parent of all non-primitive / reference types. What methods does every Object have?

- `toString()`, `equals()`, `hashCode()`, `clone()`, [others]

Suppose you have a class named `Foo`, and a `Foo` variable, `foo`. What is the default behaviour of `foo.equals(f)`?

It checks whether the addresses of `foo` and `f` in memory are the same. Usually this is not the desired behaviour. What would you want it to do?

```java
public boolean equals(Object rhs) { // tests if rhs equal to ourself
    return rhs == this;
}
```

Usually has the following tests, when implemented in a subclass / child class:

1. `rhs is instanceof this Class, cast rhs as Class`
2. `all primitive fields are equal via ==`
3. `all reference fields are equal via equals()`
Student s;
Person p = s;
s = p;

Suppose have following definitions in Person and Student class.

isOlder( Person p, Person p ) // Do we need more defs?
higherGPA( Student s ) // Only defined on students

Will the following compile and run?

Person p2 = new Student() // Yes. Polymorphism at work
Student s2 = new Person() // No. s2 won’t be a Student
s.higherGPA( p ) // No. p must be Student
s.higherGPA( s2 ) // Yes. s2 a Student
p2.higherGPA( s ) // No. compile error
((Student)p2).higherGPA(s) // Yes. compiler happy now
((Student)p).higherGPA(s) // No. May compile, won’t run

Can use instanceof to check whether casting will work to avoid runtime error.
p.toString(): Program determines at run time whose method will be called.
Suppose want classes to describe various shapes. Every shape has colour:

```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 Shape, though, because depends on WHAT Shape; the calculations are different for a Square and a Circle.

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

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

Could just remove the method from Shape class and hope that Circle and Square remember to implement those methods. What’s wrong with this sol?

Can’t have, for ex., array of Shapes, then call area on each one.
Abstract Methods & 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

    public abstract class Shape {
        private Color c;
        public abstract double area();
    }

Abstract class – a class with at least one abstract method
- abstract classes cannot be instantiated (Shape s = new Shape())
- must be declared abstract to compile (prevents programmer errors)
Sometimes a class wants to extend from 2 or more parent classes. When could this be a problem?

Suppose have Employee class, and want

```java
public static StudentEmployee extends Person, Employee

studentEmp.toString(); // Whose toString to call?
// And whose super() is called from inside StudentEmployee?
```

Interface – “the ultimate abstract class”

1. all interface methods are public abstract methods, by default
2. all fields are public static final, by default

```java
public interface Comparable {
    int compareTo( Object other );
}
```

How does this solve our problem?

You can inherit from multiple interfaces, no conflicting behaviour.

```java
public class Shape implements Comparable, AnotherClass {
    // must override / implement compareTo
}
```
When X implements an interface...

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. can’t have conflicting return types
6. if not all methods are implemented in subclass, then class is abstract and must be declared as such.
7. interfaces can themselves extend [multiple] interfaces.
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. When we write a new class, we define a generic class “parameter”. E.g.

```java
public class GenericData<AnyType> {
}
```

When a user creates an instance of the class `GenericData`, they will have to specify a reference type as the generic parameter:

```java
GenericData<Integer> myData = new GenericData<Integer>();
GenericData<Point> myPoints = new GenericData<Point>();
```

Inside our generic class, we can use our generic type parameter anywhere we want to, as a type:

```java
public class GenericData<AnyType> {
    private AnyType data;
    public AnyType getData() { return data; }
    public AnyType setData(AnyType data) { this.data=data; }
}
```
What if we want to restrict the placeholder a little?

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

```java
public class DinoData<AnyType extends Dinosaur>
```

And now the following will work:

```java
DinoData<Velociraptor> dd = new DinoData<Velociraptor>();
```

But this will not:

```java
DinoData<Point> dd = new DinoData<Point>();
```
We may need to restrict things even further:

Consider the commonly used **Comparable** interface:

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

A lot of sorting methods are defined on types that implement/extend the `Comparable` interface, because the `compareTo` method is supposed to define an ordering between to objects.

**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
public class Data<AnyType extends Comparable<AnyType>>
```
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
public class Data<AnyType extends Comparable<AnyType>>
```

Then the following statement will not work:

```java
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 Velo IS-A Dino.

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
public class Data<T extends Comparable<? super T>>
```
Limitations of Generics

1. Cannot use generics for primitive types.
   
   Data\<int\> myPoints = new Data\<int\>(); // not OK

   However, you can always use the provided wrapper classes.
   
   Data\<Integer\> myData = new Data\<Integer\>(); // OK

2. Cannot construct instances of a generic type.
   
   AnyType myVar = new AnyType(); // not OK

3. Cannot create an array of generics.
   
   AnyType\[] array = new AnyType[10]; // not OK

   However, you can create an array of
   
   AnyType\[] array = (AnyType\[]) new Object[10]; // OK

   This will give a compiler warning that you can fix by adding
   
   @SuppressWarnings("unchecked")

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