SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2017

Topic Notes: Inheritance

Our next topic is another that is fundamental to object-oriented design: *inheritance*.

Inheritance allows a programmer to take a Java class and extend or modify its functionality without changing the original class.

You have seen at least one example of inheritance this semester. In Lab 3, the `HelloApplet` class had this class header:

```
public class HelloApplet extends JApplet
```

This means that the class `HelloApplet` "inherits" the functionality of the class `JApplet`, and can then modify it as appropriate.

(There seem to be dozens of short answer exam questions just screaming out from these topics, doesn't there?)

Overloaded Functions

Before we get into the details of inheritance, we will first consider the concept of *overloading* of constructor and method names.

Overloading is when a program has multiple constructors or methods of the same name. Java will determine which of the overloaded functions is called based on the *signature* (*i.e.*, the parameter list).

This kind of overloading is known as *ad hoc polymorphism*.

You have likely used this in your programs, and we saw overloaded constructors in an example earlier this semester:

See Example: Point

Here is a similar example that overloads the method `distance` to compute the distance from this `Point` to another `Point`, or the distance from this `Point` to a given x,y-coordinate pair, or the distance from this `Point` to the origin, all using the method name `distance`.

See Example: PointOverload

Inheritance

Inheritance allows a Java programmer to write classes that build upon the functionality of others.

Some terminology:

- We say that the *subclass* or *extended class extends* the *superclass* or *base class*.
- A *derived class* of a base class is any class which has the base class as an ancestor. That is, there is some series of superclass/subclass relationships that connect the base class with the derived class.
- Inheritance defines an *IS-A association* between the subclass and its superclass. If `A extends B`, then `B` "IS-A" `A`.

As a simple example, we extend the `Point` class from the previous example to include a color.

See Example: ColorPoint

The `ColorPoint` class has three data members: the `x` and `y` values that are inherited from `Point` and `color` which is added by `ColorPoint`. It also inherits methods from `Point`.

Note that the `toString` method is *overridden* by `ColorPoint`.

Note that we needed to change the protection of the `x` and `y` fields from `private` to `protected` to allow them to be available in the `toString` method of `ColorPoint`.

`java.lang.Object`

Every Java class has a unique superclass hierarchy that ends with `java.lang.Object`. Every class inherits from `Object` implicitly (*i.e.*, no `extends` is needed).

Some programming languages allow *multiple inheritance*, where one class can explicity extend multiple other classes, but Java disallows this. Every class (except `java.lang.Object`) extends exactly one other class. That superclass is `Object` unless otherwise specified.

Multiple inheritance can be a useful tool to programmers, but it significantly complicates the compilers and run-time systems of languages that support it. The designers of Java chose not to include this feature.

`protected` and default

Our recent discussion of packages and current discussion of inheritance also allows us to consider more of the differences among Java's qualifiers for access control on variables and methods:

`public` members can be accessed by anyone

`private` members can be accessed only within the class where defined

`protected` members can be accessed within the class where defined, by other classes in the same package, and by any derived classes

Default (no qualifier) members can be accessed within the class where defined and by other classes in the same package

---

UML and Inheritance

The superclass/subclass relationship is shown in a UML class diagram by a hollow arrow from the subclass to the superclass. This indicates the "IS-A" relationship. We previously saw that a regular arrow indicates a "HAS-A" relationship.

Since the superclass is also shown in the diagram, there is no need to replicate the inherited members in the diagram of the subclass.

Subtypes

A subclass is a *specialization* of its base class. However, objects of the subclass still are and can be treated as objects of the base class.

We say that the subclass determines a *subtype* of the type of the superclass.

An object of a subtype can be used anywhere that an object of its supertype is expected.

Java will perform a *type conversion* when object types are not precise.

- A *widening* conversion converts a subtype to one of its supertypes.
- A *narrowing* conversion converts a supertype to one of its subtypes. This is sometimes referred to as *downcasting*.

Let's examine *subtype polymorphism* through a series of examples.

Suppose we have a base class `Student` and two derived classes `Undergraduate` and `Graduate`:

```
class Student { ... }
class Undergraduate extends Student { ... }
class Graduate extends Student { ... }
```

We can store references to either subtype in a variable of the base class type:

```
Student s1, s2;
s1 = new Undergraduate();  // valid widening
s2 = new Graduate();       // valid widening
```

Now suppose we have a variable of one of the subclasses:

```
Graduate s3;

s3 = s2;  // compilation error, cannot guarantee downcast
s3 = (Graduate) s2;  // OK, since s2 is a Graduate
s3 = (Graduate) s1;  // Compiles, but run-time cast failure
```

We can avoid a potential run-time failure by checking the type of an object with the `instanceof` keyword:

```
if (s1 instaceof Graduate) {
  // perform safe downcast
  s3 = (Graduate) s1;
}
```

Overriding Functions

Inheritance brings up a variety of issues that can be powerful and in some cases, potentially problematic.

We saw an example above where a derived class (`ColorPoint`) provided a `toString` method that was intended to *override* the `toString` method provided by its base class (`Point`). Let's look at that idea a bit more closely next.

A method in a subclass will *override* a method inherited from its superclass if it has the same method prototype: the same name and the same signature (*i.e.*, parameter list).

Note that this is different from *overloading*, where the same class provides multiple methods (or constructors) of the same name but with different signatures.

We looked earlier at the concept of *subtype polymorphism*. In the example above, variables of type `Student` could be used to refer to either `Student` objects, or objects of types that are derived from `Student`: `Undergraduate` and `Graduate`.

Let's extend that example a bit, by adding a few more classes, `Freshman`, which is an extension of `Undergraduate`, and `Phd`, which is an extension of `Graduate`, and providing a method `getName` that prints out the student's name, annotated with the type of student, where applicable:

See Example: Overriding

Even though all of the `getName` method calls are using a reference of type `Student`, the `getName` method that gets called is determined by the type of the object, not of the reference. Java uses *dynamic binding of method calls* to accomplish this.

In some cases, this is straightforward: the method is defined in the class definition corresponding to the object itself. However, in cases like the object of type `Freshman`, there is no `getName` method defined in that class. So it must locate and execute an appropriate `getName` method further up the class hierarchy. When we call the `getName` method of the object of type `Freshman`, it first checks its direct superclass, `Undergraduate`. It finds the method there, and uses it. If `Undergraduate` did not override the `getName` method, Java would then continue up the class hierarchy, and use the `getName` method in class `Student`.

The `@Override` Annotation

It is not required, but is considered good programming practice to include the `@Override` annotation on each method that is intended to override a method of an ancestor in its class hierarchy. Refer to the previous example to see it in use.

The main advantage of using the `@Override` annotation is that it can help detect programmer errors such as method name misspellings and method signature mismatches. For example, suppose we mistakenly called our method `getname` in the `Phd` class. The `@Override` annotation would trigger a compile error (try it, it's fun!). Without the `@Override` annotation, `getname` would be defined as a new method of `Phd` and it would inherit its `getName` method from `Graduate`. The mistake would need to be detected at run time.

You will be expected to include `@Override` annotations as appropriate in your programs.

Access Control with Overriding

What should we do about the access control of a method in a derived class that overrides a method in its base class? That is, if the method was `public` in the base class, must it be `public` in the derived class? Yes, in that case, but Java does allow the method in the derived class to have a *less restrictive* access control than the method it overrides from the base class. However, a `private` method in the base class cannot be overridden, since it is not visible to any code except the class in which it is defined. A method with the same name and signature in the derived class would simply be a method of that class, and would not override the `private` method in its superclass.

Try it out:

See Example: OverrideAccessControl

Covariant Return Types

Normally, we would expect the return type of a method in a subclass that overrides a corresponding method in a superclass to be the same as the return type in the superclass. We are allowed to modify the return type of the overriding method, as long as it is a subtype of the original return type. This is called a *covariant return type*.

This can help reduce the amount of type checking (with `instanceof`) and casting in some circumstances.

See an example at `http://javapapers.com/core-java/covariant-return-type-in-java/`

The `final` Keyword

You have most likely used the `final` keyword in Java when defining constants:

```
private static final int BOARD_SIZE = 100;
```

or local to a method:

```
final String WIN_MESSAGE = "You win!";
```

In this context, `final` indicates that the value of the name being defined cannot be changed subsequently. A statement that tries to assign a new value to such a name will trigger a compile error.

You are less likely to have encountered the `final` keyword as a qualifier on a formal parameter:

```
public void move(final double dx, final double dy) {
   ...
}
```

Normally, you can use parameters as local variables and change them as you see fit in the body of the method, but adding the `final` qualifier disallows this.

Note however, that this only means that the variable cannot be changed. It does not mean that the variable's value cannot be used to change something. For example, consider this method:

```
public void f(final ArrayList<Integer> a) {

    // this would be an error:
    // a = new ArrayList<Integer>();

    // but this is permitted:
    a.add(27);
}
```

Even though `a` is `final`, and it cannot be modified to refer to a different `ArrayList`, there is nothing stopping that reference from being used to modify the `ArrayList` to which it points.

`final` and Inheritance

The `final` keyword can also be used to place restrictions on inheritance and function overriding.

If the `final` qualifier in a class header, it means that the class cannot be extended:

```
final class A { }

class B extends A { }
```

This results in a compile error.

`final` can also be used on an individual method to prevent that method from being overridden in any subclass:

```
class A {
```

```
    public final int x() {

        return 1;
    }
}

class B extends A {

    public int x() {

        return 2;
    }
}
```

This also results in an error at compile time.

Declaring a class as `final` is useful in cases where objects of the class are intended to be *immutable*, like `java.lang.String`.