SIENAcollege
Computer Science

Computer Science 225
Advanced Programming
Siena College
Spring 2021

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.

Overloaded Methods

Before we get into the details of inheritance, we will first recall 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. The `SimpleLinkedList` we have worked with has overloaded methods.

`PointOverload` has a class that overloads its constructors and 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`.

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. Come back to take Programming Languages to find out more.

`protected` and default

We now also need to consider more completely 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 (a topic we'll discuss in more detail soon), 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

---

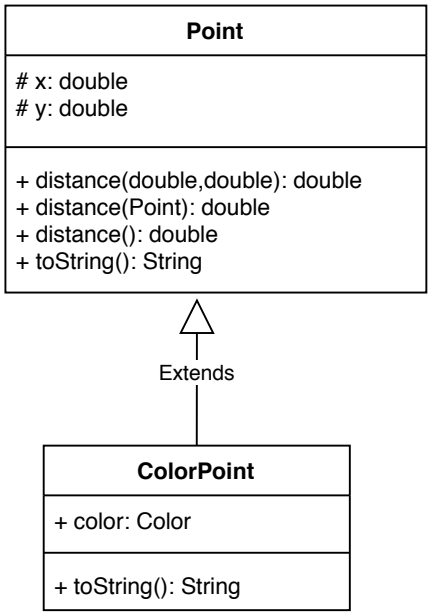
UML and Inheritance

We will at times this semester be using diagrams to design and visualize class hierarchies. The *Unified Modeling Language (UML)* (see **On the web:** UML on Wikipedia at `https://en.wikipedia.org/wiki/Unified_Modeling_Language` for a summary, **On the web:** the complete UML specification at `https://www.omg.org/spec/UML/2.5.1/PDF` for almost 800 pages of details) is a software engineering modeling language that includes a specific diagram format for *class diagrams* (**On the web:** Wikipedia at `https://en.wikipedia.org/wiki/Class_diagram`).

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.

For our earlier example, this is the UML class diagram:



Subtypes

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

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:

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`.

In class, we will draw a UML class diagram for this example.

The `@Override` Annotation

As we discussed in our examples of Java interfaces, the `@Override` annotation not required, but is considered good programming practice for 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 (and please point out when you notice one missing in a class example).

Abstract Classes

So far, we have seen interfaces and "regular" classes, which we will call *concrete classes*. There is a level between these called an *abstract class*, which has some method definitions like a regular class and some method signatures that must be implemented by subclasses.

For example, in our example with the animals, we could make the `Animal` interface into an abstract class and include the complete functionality related to animal weights at that level.

`AnimalAbstract`

Key points about an abstract class:

- The class header is preceded by the keyword `abstract`.
- Methods that are not implemented by are required to be implemented by subclasses are also preceded by the `abstract` keyword.
- Abstract classes can declare instance variables and use them in its method definitions.
- An abstract class can implement zero or more interfaces and may extend one abstract class or concrete class.
- Like interfaces, abstract classes cannot be instantiated.

It is possible for a class that extends an abstract class to override methods defined in the abstract class, in case there is a more efficient way to do some of these things when an actual implementation is developed.

A frequent use of an abstract class is to "factor out" implementation of methods that happen to be the same for multiple implementations of an interface.

Another use of an abstract class is to provide default implementations of methods that can be implemented in terms of other methods.

Concrete classes extend exactly one concrete or abstract class (that class is `java.lang.Object` if not specified).

Additional abstract class example building on our `Student` example from before: `AbstractClass`

Case Studies

We will examine the class hierarchy of Bailey's Structure Package as it implements list structures and graph structures.