Brief review

Last lecture we discuss overriding and overloading. This concerned methods that have multiple definitions. These multiple definitions could either be within a class (overloading only) or they could be between two classes (overriding or overloading, depending on whether the method signatures match or not). That discussion concerned compile time only, the methods as they appear in the class definitions.

I spent a few minutes reviewing it at the beginning of the lecture, but do not repeat the discussion here. The key points were:

- what it means for one class to **B** to extend another class `A`
- the difference between overloading and overriding
- if you have a method call of the form `v.m(a)` where `v` is a reference variable, `m` is a method that the referred object can invoke, and `a` is an argument passed to that method, then the program will only compile without error if the type of `a` is consistent with a method `m` that is well defined by the declared type (i.e. class) of `v`. This is referred to as *type checking.* More on that below...
- classes can be summarized with a *class diagram* and relationships between classes can be summarized with an *inheritance diagram.* See the textbook (GT p. 74) for an example. These terms did not appear in the previous lecture notes, even though class diagrams and inheritance diagrams were drawn.

Polymorphism

Suppose you have some reference variable. It is is declared to be a certain type. You might assume that this means that the variable can point *only* to objects that are of that type[1] However, this would too be limiting a requirement. For example, Java allows us to write:

```
Dog   mydog = new Beagle(``Buddy'');
```

even though we have two types in the above definition. It makes sense to allow this definition because `Beagle` is a subclass of `Dog`. Any object that belongs to class `Beagle` also belongs to class `Dog`. In particular, an object of class `Beagle` inherits all fields and methods of class `Dog` (except of course those methods that are overridden).

We say that the variable `myDog` is *polymorphic* (from Latin: poly means "many" and "morph" means forms). It can sometimes refer to objects of class `Dog` and sometimes refer to objects that belong to a subclass of `Dog`. (I do not give a formal definition of polymorphism here. You will see more of this in COMP 302, 303, 304.)

Several issues arise. First, why not just declare `myDog` to be of type `Beagle` ? The reason is that it could happen that later we may want `myDog` to point to a different dog. e.g. `Buddy` might runs away from home or die, and we might replace him, e.g.

```
mydog = new   Doberman(``Dooby'');
```

[1]By the "type" of an object, I just mean the class that is specified when the object was constructed e.g. `new ClassName()`.

Note that by declaring `myDog` to be of type `Dog`, we are not allowing `myDog` to invoke methods that are in subclasses of `Dog`. For example, beagles chase rabbits but many other dogs don't. So if class `Beagle` had a method `chaseRabbit()`, then we would not be allowed to say `myDog.chaseRabbit()`. The Java compiler will not allow it since `chaseRabbit()` is not defined in class `Dog`.

A similar issue arises if we were to write:

```
Beagle   mydog = new Dog(``Buddy'');
```

The compiler would give an error here. The reason is that, since `Beagle` is a subclass of `Dog`, there may be fields and methods in class `Beagle` that are not in class `Dog` and it would not make sense to reference such fields or invoke such methods on the `Dog` object which is constructed by this statement. (For example, the method `catchRabbit()` is not defined in class `Dog`.) Java protects the programmer from such mistakes happening, by given a compiler error.

One last example: What would happen if we declared `myDog` via

```
Object myDog = new Beagle(``Buddy'');
```

This would be legal. However, it would not be as useful as declaring the type of `myDog` to be `Dog`. The reason is that there are methods in the class `Dog` that we would like to invoke which are defined for all dogs, but which are not defined for all objects. If we define `myDog` to be of type `Object`, we would not be allowed to invoked the `Dog` methods.

Dynamic dispatch

Let's now turn to the question of what method is invoked when the program is running (assuming it has compiled without error). In Java, the method that is actually used is the method that is defined by the actual object referred to. This choice is called *dynamic dispatch*. There are a few different cases to be aware of here.

The first case is what I discussed above: when you invoke an object's method, you need to know what is the type of that object and you use the method that corresponds to the actual object's type. (Recall that the object's type is well defined when the object is constructed, and the type of the object doesn't change.) So, for example:

```
Object  it;
   :
it = new float[23];
   :
it = new Dog();
   :
System.out.print(it.toString());
```

At compile time, we cannot say which `toString()` method will be used. We can only say at runtime, when the last instruction above is executed and `it` references some particular object. The object belongs to some class which either has its own `toString()` method or has inherited the `toString()` method from an ancestor.

Here is a more complicated example. Suppose `myDog` and `yourDog` are defined by

```
Dog  myDog, yourDog;
```

and later we have a method invocation

```
myDog.reactsTo(yourDog)
```

where `reactsTo( )` is a method defined in class `Dog`

```
public void reactsTo(Dog  otherDog){ .. }
```

and may be overridden or overloaded in various subclasses of `Dog`. For example, with the class `Beagle` we might have methods:

```
public void reactsTo(Dog       otherDog){ .. }   // overriding
public void reactsTo(Beagle    otherDog){ .. }   // overloading
public void reactsTo(Doberman  otherDog){ .. }   //   "
```

At runtime, the version of `reactsTo` that is invoked by `myDog.reactsTo(yourDog)` is determined by the type of the object that `myDog` references – as in the *it* example above – and it also depends on the type of object that `yourDog` references, since there are several versions of the method `reactsTo` that are available.

Example (not discussed in class)

Let's try to understand a statement like:

```
System.out.println(myDog.toString());
```

If you look up the class `System` in the Java API, you'll see it is defined `public final class System`. It extends the class `Object`. The class `System` has a field `out`, that is, `System.out` is a reference variable which is a (static) field in class `System`. Because it is a static field, there is no object created and so we are refering to this variable `out` using the class name, rather than an object name.

If you look it up in the Java API, you'll find that the type of the reference variable `System.out` is `PrintStream` which is just another (static) Java class. The class `PrintStream` has a (static) method `println()` which you are familiar with. This method is overloaded within the class `PrintStream`. One version of `println` has a single `String` parameter and so, glancing at the code, you would assume this is the version used here.

However, this is not necessarily the case. Because `myDog` is polymorphic, you don't necessarily know what class `myDog` is at the time the method is invoked. Of course, you expect `myDog.toString()` will return a string. However, if `myDog` were to reference an object of a new class `BadlyDefinedDog` in which `toString()` is overridden and now (because the programmer is not paying attention to what he/she is doing) returns say a `float` rather than a `String`, then the version of `println()` which expects a float parameter would be invoked.

I give you this example, not as a way of torturing you, but rather to emphasize the mechanisms involved here. In particular, it is important to distinguish `type checking`, which is done at compile time, versus dynamic dispatch which is done at runtime. Type checking ensures that statements of the program make sense in terms of how variables and method parameters are declared. (The above statement passes the type check, since `myDog` is declared to be of type `Dog` which would either have its own `toString` method or would inherit this method. Type checking still leaves ambiguity, however, because of polymorphism. During runtime, the method that is used is based on the object that is actually present.