Polymorphism

In the last few lectures, we have seen that the declared type of a reference variable does not entirely determine what classes of object the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, or it can also reference an object that is a subtype of the variable's declared type. This property, that the object type can be narrower than the declared type, is called *polymorphism*[1].

Last lecture we concentrated on the type checking that is done by the compiler. We looked at when implicit casting was good enough and when we needed to do explicit casting. We also looking at the `instanceOf` operator which is used for "dynamic type checking", namely checking the class of an object at runtime. Let's now assume a program has compiled fine, and let's focus on which method is invoked at runtime. We will see that *the method used is the one defined by the object that invokes it.* There are a few different cases to be aware of here.

The first case is what I discussed above: when an object invokes a method at runtime (i.e. we are talking about an object and objects only exist at runtime), the method is determined by the class that the object belongs to. How does the Java Virtual Machine running the program know what class the object belongs to? (The answer, as we'll see below, is that this information is stored in the object in the `class` field that references the class descriptor. This field can be accessed with the `Class getClass()` method, which the object inherits from the `Object` class.)

Consider, for example:

```
boolean  b;
Object  obj;
   :                      //  some code not specified here
if (b)
   obj = new float[23];   // an array of floats
else
   obj = new Dog();
System.out.print(obj);   //  invokes the toString() method (*)
```

The compiler cannot say for sure which `toString()` method should be used, since the compiler doesn't know for sure what the value of `b` will be when the `if (b)` condition is evaluated. Rather, the `toString()` method must be determined at runtime, when (*) is executed and the variable `obj` references either a `float[]` or a `Dog`. In each case, there will be a `toString()` method used which is appropriate for the object. (Recall that every class has a `toString()` method.)

Class descriptors and objects

When you define a class by typing some ASCII code into a `.java` file, your class definition includes various things: the name of the class, a list of fields and types, a list of methods and their signatures, and the instructions of each method, visibility modifiers such as `public`, and other info. When you compile the class, the compiler makes a `.class` file, which is stored in a directory on your computer. That directory is determined by the package name that you write in the first line of the `.java` file.

[1]from Latin: poly means "many" and "morph" means forms

When a program uses a particular class, the information about that class is stored as an object in the program, and the object is called a *class descriptor*. This class descriptor is different from a `.class` file generated by a compiler, although the information in a class descriptor is basically the same (for our purposes, anyhow) as the the information in a class file. In particular, class descriptors contain the name of the class, an array of fields and types, an array of methods including the instructions of each method, a reference to the superclass, etc.

I emphasize: class descriptors are the objects in your running program, just like the objects that are generated by `new` commands. What class do these class descriptor objects belong to? Answer: the `Class` class. [ASIDE: In my opinion, it might have been better if the good people who created Java had called this the "`ClassDescriptor`" class, since saying "an instance of class `Class`" is more confusing than saying "an instance of class `ClassDescriptor`".]

Also, note that the class descriptor of the `Object` class, and the class descriptor of the `Class` class are also class descriptors. Hence they are instances of the `Class` class. So, we have objects that are instances of the `Object` class; we have objects (class descriptors) that are instances of the `Class` class; and we have objects that are instances of lots of other classes.

You may be wondering how is this useful, and why am I torturing you with this? The answer is that, although the `Class` and `Object` classes are strange when you first meet them, once you understand what they are for, you will see that they make beautiful sense. Moreover, it is nearly impossible to understand how inheritance and polymorphism work in Java, without understanding what these classes are for. (See the slides for illustrations of what I am discussing here.) So let's explore this more to help get you there.

In lecture 30, I mentioned several methods in the `Object` class, such as `hashCode(), toString(), equals()`. One that I did not mention is `getClass()`. This method returns a reference to the class descriptor of the class that this object belongs to, that is, the class whose constructor invoked this object. So, the return type of `getClass()` is `Class`, since this method returns a reference to a class descriptor. For example, `myDog.getClass()` would return a reference to the `Dog` class descriptor, which is an object of type `Class`.

The `Class` class has several methods, in particular, `getSuperClass()` whose return type is `Class`. The method `getSuperClass()` returns a reference to the class descriptor of the (direct) superclass. So, if `myDog` were a `Beagle`, then `myDog.getClass()` would return a reference to the `Beagle` class descriptor, and `myDog.getClass().getSuperClass()` would return a reference to the `Dog` class descriptor.

Notice that when we consider class descriptors as objects, we have to think of objects referencing each other in two different ways:

- "instance of" - an object is an instance of a class. This defines a reference from an instance object to a class descriptor object i.e. `getClass()`.
- "subclass of" - a class descriptor will have a reference to its superclass descriptor, i.e. `getSuperclass()`. Note that the `Object` class does not have a superclass, so the following returns `null`.

```
Object obj = new Object();
System.out.println(obj.getClass().getSuperclass());
```

Reference variables, Objects, and the Call stack

I ended the lecture by reminding you that when you are run a program, you call the main method of some class. The main method is put on the call stack, and then instructions in the main method cause objects to be created and call other methods. Each time a method is called, a new frame goes on the call stack, and each time a method returns, the frame of that method gets popped from the call stack.

Recall that the stack frames have local variables. These variables are explicitly declared in the methods. The stack frame also keeps track of the parameters that are passed to it, and other information. One variable that is implicit in each stack frame is `this` which references the object that called the method. (Some methods are `static` and aren't called by an object, so they don't have a `this`.) Anyhow, these variables in the call stack can be either primitive types or they can be reference types, in which case they reference objects. Please see illustrations in the slides at the end of this lecture for how the call stack is related to the objects and to the class descriptors that I described above.

In particular, I walked you through the example:

```
Dog    myDog = new Beagle();
       myDog.bark()
```

Note that I neglected to mention the step where the constructor `Beagle()` also gets pushed onto the call stack and then popped.

Finally, please see the last slide where I reminded you where various important things are located, such as the code for running the methods in a class (in the class descriptor), the local variables in a method (in the stack frame) and the objects that are instantiated by class constructors.