Abstract Methods and Classes

Suppose we have a Pet class (a base class) and then Dog and Cat classes that inherit from Pet (Dog and Cat are derived classes):

```
public class Pet { }                // content of these classes doesn’t matter right now
public class Dog extends Pet { }
public class Cat extends Pet { }
```

Furthermore, suppose we want to have the Dog and Cat classes have the ability to “speak” (make a sound, like a bark or a meow). We can easily add a speak() method to these classes.

```
public class Pet { }
public class Dog extends Pet {
   public void speak() { System.out.println(“Woof”); }
}
public class Cat extends Pet {
   public void speak() { System.out.println(“Meow”); }
}
```

And now suppose we have an ArrayList of Pets, and we want all the pets to speak:

```
ArrayList<Pet> pets = new ArrayList<Pet>();
pets.add(new Dog());
pets.add(new Cat());
for (Pet p : pets) {
   p.speak(); // doesn’t work!
}
```

We know through polymorphism that creating the ArrayList of Pets works fine, and adding a Dog and a Cat to the ArrayList also works fine. However, calling p.speak() will not work (the program will not even run), because the Pet class does not have a speak() method; that method lives in the Dog and Cat classes. And we know with polymorphism, Java has to determine *before* the program runs if it is “legal” to call a method on an object, and all it knows about this object “p” is that it is a Pet, so it might not be able to speak (even though **we** know that all the Pet subclasses have this ability).

How do we fix this?

One solution is to cast the variable p to either a Dog or a Cat before calling speak on it. Java knows that both of those classes have a speak()method, so this will work. We can use instanceof to determine if p is a Dog or a Cat, then do the appropriate cast. However, this idea is not optimal, because it does not scale up to having more kinds of pets. If we had, say, ten kinds of pets, we would need ten different if tests to determine what kind of pet it was, do the appropriate cast, then call speak().

Another solution is to add the speak()method to the base class (Pet). This would solve the problem without adding any if tests with instanceof. However, this introduces another wrinkle: what should the speak() method in Pet do? How does a generic “Pet” speak?

This is a common situation that arises with designing a hierarchy of objects. We often have a situation where we want some behavior to be shared among all objects, but the base class represents a concept that is generic enough that it’s unclear what that behavior at the base class level should look like.

Enter abstract methods!

An abstract method allows us to solve this problem. This concept allows us to declare a method in a base class that *must* be overridden in a derived class because it is, in some sense, too generic (or abstract) to be implemented at the base class level. In other words, it represents the concept of "All subclasses have this behavior in common, but they all do it differently, and therefore the superclass can't provide an implementation for it."

```
public abstract class Pet {
   public abstract void speak();          // notice there is no body for this method
}
public class Dog extends Pet {
   public void speak() { System.out.println("Woof"); }
}
public class Cat extends Pet {
   public void speak() { System.out.println("Meow"); }
}
```

When a method in a base class is marked as abstract, two things then happen. First, the method will not have a body. This makes the entire class an abstract class, and the `abstract` keyword must also be added to the class definition itself (notice the two `abstracts` above).

Abstract classes cannot be instantiated (you can't create new objects of this class). In other words, a line of code like this is now illegal:

```
Pet p = new Pet();  // illegal!
```

The reason for that is because if this were legal, and we were to call `p.speak()`, there is no code for this method, so Java wouldn't know what to do.

Second, when a method in a base class is marked as abstract, any derived class must either override the method (to provide an implementation with code), or the derived class itself must be marked as abstract (and the same instantiation caveat as above would apply).

Note: just because we can't instantiate a new object of an abstract class doesn't mean we can't declare a variable of that class. For instance, this is fine:

```
Pet p = new Cat();      // legal
p.speak();              // legal
```

This code works fine because now the `p.speak()` method is a part of the `Pet` class (even though it's abstract), so Java can determine before the program runs that the `p.speak()` line above will work fine.

Similarly, our original code from earlier:

```
ArrayList<Pet> pets = new ArrayList<Pet>();
pets.add(new Dog());
pets.add(new Cat());
for (Pet p : pets) {
   p.speak(); // doesn't work!
}
```

will work fine now for the same reason. Before the program runs, Java knows the `p.speak()` line will work fine, because all `Pets` have a `speak()` method. However, Java does not know until the program begins running *which* `speak()` method will be called by that line of code; in fact, because of the loop above, the first time through the loop Java will call Dog's `speak()` method, and the second time, it will call `Cat`'s `speak()` method.