I began by reviewing the `Dog` class and `Progression` class and their subclasses, which I discussed last lecture. (Discussion not repeated here.)

The class "`Object`"

Java allows a class to (directly) extend at most one other class. The definition of a class is of one of the two forms:

```
public class MyClass
```

or

```
public class MyClass extends ExistingClass
```

where `extends` is a Java keyword. If you don't use the keyword word `extends` in the class definition then Java automatically makes `MyClass` extend a default class, which is the `Object` class. So, the first definition above is equivalent to

```
public class MyClass extends Object
```

The `Object` class contains a set of methods that are useful no matter what class you are working with. In Java, regardless of how a class is defined, it is always a description of things which we are calling objects, and so any instantiation of any class is always some object, and so we can safely say that it belongs to class `Object`. See the API:

```
http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html
```

As stated there, "Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class."

What is the intuition here? In our everyday lives, when we talk about particular instances of classes e.g. particular rooms or particular dogs, it always *makes sense* to ask certain types of questions, for example, "is this object the same as some other object?" We can ask whether two rooms are the same e.g. "is this week's tutorial in the same room as last week's tutorial", or we can ask if two dogs are the same "is that the same dog I saw you with yesterday?" Asking if two objects (of any given class) are the same is something we may want to do in Java as well. Similarly, we often may want to ask for a description of an object.

In Java, the class `Object` has several methods that it can invoke (see the Java API URL above) which are of this general nature. One commonly used method is `equals()` which tests whether the invoking object is the same object as some other object, that is, whether the reference variables for the two objects are aliases. This method does the same thing as the operator `==`.

For example, suppose `a` and `b` are the names of objects, and suppose they are of class `Object`.

```
Object a = new Object;
Object b = new Object;
         :
```

If later we had the expression `a.equals(b)` then this expression would have the value true if and only if `a` and `b` were aliases, i.e. they pointed to the same object.

[ASIDE: One student observed out that, in COMP 202, you were told that to check for aliasing you use the operator `==`, and that `equals()` means something other than aliasing. In particular, you were told that, for the class `String`, the method `equals()` is used to check if a type `String` variable points to a particular string (sequence of characters). So, you might have:

```
String s = "initial";     // initialize a variable
  :
  :                       //  .. some code...
  :
if (s.equals("initial")){   //  check if the variable's value
                            //  has changed since it was initialized.
  :
```

This observation is correct. *However,* it is not the whole story. Yes, the operator `==` does check for aliasing, but it only does so when the two operands (reference variables) are of the same type, (You get a compiler error when the operands are of different type.) The `equals` method from the `Object` class does not have this requirement. It compares *any* two reference variables, and returns true if the referenced objects are the same.

Technically, we would say that, for the `String` class, the method `equals()` is *overridden*. See next section below.]

Another commonly used method in class `Object` is `clone()`. This method creates a different object, which is of the same class as the invoking object and which has fields that have identical values to those of the invoking object (at the time of the invocation). So,

```
Object c = a.clone();
```

would create a new object whose fields are identical to those of `a`. Note that `c.equals(a)` would be false, since we have two different objects.

A third method of class `Object` is `toString()`. You are familiar with such a method from COMP 202. I will discuss it below, as it is another example of "overriding".

Overriding

The discussion above uses the term "override" but didn't define it, so let's do that now. It often happens that we wish to use the same method name in a subclass as we use in the superclass. In particular, we may want to change the method in some way for the subclass, and so we would need to redefine the method in the subclass. We can do so by either changing just the method body (but not the signature), or we can change the method body *and* the signature. In the case that we change just the method body, we say that the method of the subclass *overrides* the method of the superclass. The `equals()` method for class `String` was one example. Let's briefly consider two more examples.

Example 1: the `Progression` class

Last lecture we looked at the `Progression` class from the textbook and we saw that this class and its three subclasses all had a method `nextValue()`. In each case, there are no parameters in the method and so the signatures are trivially all the same. (You should go back to that example in the text.)

Example 2: `toString()`

Another common example of overriding occurs in the `toString()` method of `Object` class. As you saw in COMP 202, this method is commonly used to write out a description of the state of the class (namely the values of its fields). The author of the class is free to define `toString()` however he/she wishes. As you can verify from the Java API, the signature is

```
public String toString()
```

i.e. the return type is String. This is an example of overriding, since we are replacing the `toString()` method of the `Object` class with a different version of the method in the subclass, but the signature is (trivially) the same.

Overloading

Another way of having different methods with the same name is to *change the signature*, namely to change the type and/or number of parameters. Whenever we have a multiple method definitions either in the same class or in a sub- and super-class such that the methods have the same name but different signatures, we say that the method is *overloaded*.

Overloading within a class

Let's first consider a method that is defined more than once *within* a class. You should have seen examples of this in COMP 202, but I will briefly review it here. We take the example of a constructor method. When a class has multiple fields, these fields are often initialized by parameters specified in the constructor. One can make different constructors by having a different subset of fields. For example, if I want to construct a new `Dog` object, I may sometimes know the dog's owner and name and sometimes I might just know its owner, and sometimes neither, so I use different constructors in each case.

```
public  Dog(String name, String owner){
       :
}

public  Dog(String name){
       :
}

public  Dog( ){
}
```

The last of these constructors is the default constructor which has no parameters. In this case, all numerical variables (type int, float, etc) are given the value zero, and all reference variables are initialized to `null`.

Note that overloading doesn't only happen with constructors, though. It can be done with any method (except `main`*).*

Overloading between classes

What if we have a method that is defined in a subclass and in a superclass such that the signature differs between classes. This is *also* called *overloading*. It is easy to see how such a situation might arise. The subclass will often have more fields than the superclass and so you may wish to include one or more of the these new fields as a parameter in the method's signature in the subclass. Or one of these new fields might replace one of the fields in the signature from the superclass.

Constructor chaining, and the keyword `super`

A related issue is how to define constructors of subclasses. Subclasses inherit the fields of their superclass automatically (as well as the fields of the superclass of the their superclass, etc). So, when an object of a subclass is constructed, the fields of this subclass are created and, in addition, the fields[1] of the superclass are also created, etc. This is called *constructor chaining*. How is it achieved ?

The first line of any constructor is

```
super(...);   //  possibly with parameters
```

and if you leave this line out (as you have done in the past!) then the Java compiler puts in the following (with no parameters):

```
super();
```

This causes the statements in the superclasse'ss constructor to be executed, which typically sets the fields of the superclass to some value. (Note that the superclass has its own `super(..)` statement, and so on, which causes the fields of *all* the ancestor classes to be initialized.)

What happens when the superclass has more than one constructor ? In this case, you choose which of the superclass'es (overloaded) constructors you want, based on the signature. You do so by setting the signature of the `super` command so that it matches the signature of the constructor you want in the superclass. In the following example, it is assumed that the class `Dog` has string fields that specify the name and owner and that the class `Beagle` merely inherits these fields (but doesn't redefine them). So, the `Beagle` constructor might be:

```
public  Beagle(String name, String owner){
      super(name, owner);
       :
}
```

[1] Only the non-`private` fields of the superclass are created.

```
public  Beagle(String name){
      super(name);
       :
}
```

A few final notes (not discussed in class, but worth mentioning):

- Java does not allow you to choose constructors from the superclass of the superclass. i.e. you cannot write `super.super`
- It doesn't make sense to talk about a subclass constructor overriding a constructor from a superclass, since a constructor is a method whose name is the same as the class in which it belongs and this is obviously different from the name of the superclass.

Everything we have discussed today concerns relationships between classes, which are defined at compile time. Next class, we will look at *polymorphism* which is concerned with how methods in subclass and superclasses are chosen when a program runs.

Hint: I strongly suggest that you get your hands on a basic Java programming book (not the course textbook), so that you can see lots of examples. There are many basic Java books available for loan in the Schulich library.