Lecture 26

MacDonald's farm

(Polymorphism and dynamic method binding)

```
public class Animal
{  public Animal(String aBreed)
   {  breed = aBreed;
   }

   public String getBreed()
   {  return breed;
   }

   public void talk()
   {  System.out.print("");
   }

   protected String breed;
}
```

(`Animal` could be an abstract class; `talk` could be an abstract method)


```
public class Cow extends Animal
{  public Cow()
   {  super("cows");
   }
   public void talk()
   {  System.out.print("Moo");
   }
}
public class Pig extends Animal
{  public Pig()
   {  super("pigs");
   }
   public void talk()
   {  System.out.print("Oink");
   }
}
public class Aardvark extends Animal
{  public Aardvark()
   {  super("aardvarks");
   }
   public void talk()
   {  System.out.print("Snaffle");
   }
}
```


Assignment

- Suppose I define some variables of type `Animal`. Then I can construct an `Animal` object and assign it to one of the variables. But I can also construct a `Cow` and assign it to one of the variables.

```
Animal a1, a2;
a1 = new Animal("mute swans");
a2 = new Cow();
```

- I.e. assigning from narrower to wider type is allowed
  - e.g. assigning an `int` into a `double`
  - e.g. assigning a subclass object into a superclass variable

Polymorphism

- *Polymorphism* is the ability to take several forms.
- Specifically in object-oriented languages, a variable of a particular type may refer ('point') to objects of different classes (the subclasses)
- More examples:

```
Publication p =
   new ConferencePaper(...);

Shape s = new Circle(0, 0, 5);
```


Method binding

- But what happens here?

```
Animal a = new Cow();
a.talk();
```

Which version of `talk` is run?

- *The decision is made on the basis of the class of the object, not the type of the variable that is referencing the object*
- I.e. the object is of class `Cow` so it is `talk` in `Cow` that is executed.

Dynamic method binding

- Method binding: deciding which of several methods of the same name to run
- At what point does Java decide?
  - at compile-time? (static method binding)
  - at run-time? (dynamic method binding)
- Modern object-oriented languages such as Java do it (mostly) at run-time because only then will you know exactly what class of object the variable is pointing to
- There is, however, a small performance degradation in doing it this way. (In fact, Java mixes static and dynamic binding; it uses the former in rare cases when it is possible, in order to gain efficiency)

Illustrating all this

```
Animal a = new Cow();
a.talk();
a = new Pig();
a.talk();
System.out.println(a.getBreed());
a = new Animal("Centipede");
a.talk();
System.out.println(a.getBreed());
```

Although variable `a` is of type `Animal`, we didn't repeatedly execute methods in `Animal`. During the program, `a` references various different classes of object, and, when we sent messages, the method that was executed depended on what `a` was referring to.

The power of polymorphism and dynamic method binding

- One exceptionally useful exploitation of polymorphism is to have a data structure, such as an array, where the elements are declared to be of some general class, and then objects constructed from subclasses are assigned to the elements
- Hence, we get a data structure of mixed kinds of objects
- Then we can work through the elements of the array, invoking methods on them, knowing that, by dynamic method binding, the right methods will get invoked

An array of `Animal`s

```
public class SingAlong
{
   private static Animal[] livestock;

   public static void main(String[] args)
   {  livestock = new Animal[3];
      livestock[0] = new Cow();
      livestock[1] = new Pig();
      livestock[2] = new Aardvark();
      // continued...
```

`livestock` is a polymorphic data structure: it can hold objects of different forms: objects that are any subclass of `Animal`

Teaching the world to sing

```
   for (int i = 0;
        i < livestock.length; i++)
   {  System.out.print(
         "Old MacDonald had a farm. ");
      System.out.println(
         "Eee Eye, Eee Eye, Oh!");
      System.out.print(
         "And on that farm he had some ");
      System.out.print(
         livestock[i].getBreed() + ". ");
      System.out.println(
         "Eee Eye, Eee Eye, Oh!");
      for (int j = i; j >= 0; j--)
      {  chorus(livestock[j]);
      }
      System.out.print(
         "Old MacDonald had a farm. ");
      System.out.println(
         "Eee Eye, Eee Eye, Oh!");
      System.out.println();
   }
}
```



```
   private static void chorus(Animal a)
   {  System.out.print("With a \"");
      a.talk();
      a.talk();
      System.out.print("\" here, and a \"");
      a.talk();
      a.talk();
      System.out.println("\" there.");
      System.out.print("Here a \"");
      a.talk();
      System.out.print("\", there a \"");
      a.talk();
      System.out.println("\".");
      System.out.print("Everywhere a \"");
      a.talk();
      a.talk();
      System.out.println("\".");
   }
}
```


Discussion

- A complex program without an `if` in sight! Java makes the decisions about what to run for us
- Adding a new subclass of `Animal` is easy, and there's no `if` to extend with another case: so nothing really changes
- Widely useful:
  - a structure of employees where different subclasses have different ways of having their pay computed
  - a structure of graphics objects where each subclass has a different way of being drawn, being enlarged, etc.
  - a structure of documents where each subclass has a different implementation of a printing method