```
                    CS 4:  Lecture 24
               Wednesday, April 19, 2006
```

JAVA INTERFACES
===============
Suppose you write an implementation of the Nelder-Mead simplex algorithm, and you want to be able to use it to optimize many different objective functions. The simplex algorithm frequently needs to call a method that evaluates the objective function at a new vertex.   But if that method is coded in Java, how can you change it in the middle of a running program?

We solve this problem with the following steps.

- Declare a _Java_interface_ for calling an objective function, containing one
  or more method prototypes.

- Define a bunch of different classes that _implement_ the Java interface, each
  having a _different_ method for computing an objective function, but all
  having the _same_ prototype.

- The simplex algorithm takes in a parameter whose "type" is the interface.
  The actual parameter passed in is an object of one of those classes. The
  object tells the simplex algorithm which method to call to compute the
  objective function.

A "Java interface" is declared using Java's "interface" keyword, which refers to something quite different than the interfaces I discussed in Lecture 13, even though it's closely related. Henceforth, when I say "interfaces" I mean public fields, public method prototypes, and the behaviors of public methods. When I say "Java interfaces" I mean Java's "interface" keyword.

Here's an example of an interface, which is declared somewhat like a class, in a file named Optimizable.java.

```
  public interface Optimizable {
    public double objectiveFunction(double[] parameters);
  }
```

Notice that there is no implementation of objectiveFunction()! It's not allowed in a Java interface. However, any class that _implements_ Optimizable must provide an implementation of objectiveFunction() with the same prototype. Here's an example of a class that computes an objective function f(p) = |p|^2.

```
  public class Paraboloid implements Optimizable {
    public double objectiveFunction(double[] parameters) {
      double sum = 0.0;
      for (int i = 0; i < parameters.length; i++) {
        sum += parameters[i] * parameters[i];
      }
      return sum;
    }
  }
```

You could write a hundred other classes that all implement Optimizable but have different objective functions. Then you can tell the simplex algorithm what objective function to use by passing it an object of the appropriate class.

```
    public void nelderMead(Optimizable opt) {
      double[] p;
      ...
      value = opt.objectiveFunction(p);
      ...
    }
```

The magic here is that the parameter "opt" can be an object of _any_ class that implements Optimizable! It's not restricted to be of one particular class. How does Java know which objectiveFunction() method to call?

Dynamic Method Lookup

Take note of the following two definitions.
  _Static_type_: The type of a variable.
  _Dynamic_type_: The class of the object the variable references.

The static and dynamic types aren't always the same. We can declare a variable whose static type is Optimizable, but whose dynamic type is Paraboloid.

```
    Optimizable opt = new Paraboloid();
    value = opt.objectiveFunction(p);
```

How does Java execute the last line above? Java follows the reference "opt" and figures out the class of the actual object "opt" points at. Then, Java looks up the right objectiveFunction() method for that particular class at run-time. This is called _dynamic_method_lookup_.

Note that there is no such thing as an Optimizable object. If you try writing "new Optimizable()", Java will give you a compile-time error.

Often, I will just say "type" for static type and "class" for dynamic type. Java chooses methods based on an object's class, never on a variable's type.

INHERITANCE
===========
Suppose you have a class Vector that represents two-dimensional vectors.

```
    public class Vector {
      double x, y;

      public double x() {
        return x;
      }

      public double y() {
        return y;
      }

      public double length() {
        return Math.sqrt(x * x + y * y);
      }
    }
```

Suppose in _another_ class you have a method that computes the angle between a vector and the x-axis.

```
      public static double xAxisAngle(Vector v) {
        return Math.acos(v.x() / v.length());
      }
```

Now, suppose you decide you need a 3D vector class as well as a 2D class. You'd like to be able to reuse some of the code for vectors. In particular, some vector-manipulating algorithms like xAxisAngle() are independent of the dimension of the vector. Sometimes you can reuse them without changing them at all.

Java provides a mechanism called _inheritance_ by which one class can inherit the properties of another, then add more. For example, here is a Vector3D class that inherits all the properties of the Vector class. (It is declared in the a file called Vector3D.java.)

```
    public class Vector3D extends Vector {
      double z;

      public double z() {
        return z;
      }

      public double length() {
        return Math.sqrt(x * x + y * y + z * z);
      }
    }
```

Vector3D is a _subclass_ of Vector, and Vector is the _superclass_ of Vector3D. Vector3D has three fields: x, y, and z. It inherits x and y from Vector. Every Vector3D object has a field "x" and a field "y", even though they're not explicitly declared in the Vector3D class. The field z is explicitly declared in Vector3D.

Vector3D also inherits two methods from Vector: x() and y().

A subclass can modify or augment a superclass in at least three ways:
(1) It can declare new fields--like z.
(2) It can declare new methods--like z().
(3) It can _override_ old methods with new implementations--like length().

The most important thing to understand about inheritance is this.

```
    +-------------------------------------------+
    | Every Vector3D object is a Vector object!!! |
    +-------------------------------------------+
```

This has subtle implications. The first implication is that you can assign a Vector3D object to a variable of static type Vector.

```
    Vector v = new Vector3D();             // Okay.
```

However, _not_ every Vector object is a Vector3D object! If a Vector object represents a 2D vector, it's not a Vector3D. So the reverse assignment is not allowed.

```
    Vector3D v = new Vector();             // COMPILE-TIME ERROR!
```

Another implication is that if a method takes a parameter of type Vector, as xAxisAngle does, you can pass it an object of type Vector3D.

```
    Vector3D v = new Vector3D();
    v.x = 3;
    v.y = 3;
    v.z = 3;
    angle = xAxisAngle(v);                 // Okay.
```

In this case, not only does xAxisAngle takes in v; it computes the correct result. The result is correct for two reasons. First, the formula for the angle between v and the x-axis, theta = arccos(v.x / |v|), works for vectors of any dimension. Second, the Vector3D class _overrides_ the length() method, so that it computes |v| correctly for a 3D vector. There are two different length() methods--one for Vector and one for Vector3D--and Java needs to know which one to call. Java makes the decision by...

Dynamic Method Lookup

When we invoke a method that is overridden in a subclass, Java calls the method for the object's _dynamic_ type, regardless of the variable's static type.

```
  Vector v = new Vector();
  l = v.length();                  // Calls Vector.length()
  v = new Vector3D();
  l = v.length();                  // Calls Vector3D.length()
```

In the statement "v = new Vector3D()", the static type of v is Vector, and the dynamic type of v is Vector3D. The dynamic type controls which length() method is called.

Java doesn't care that "v" has type Vector, or that the line "l = v.length();" is exactly the same both times. It calls a different method the second time, because the variable v references a different object. That's dynamic method lookup in action.

Why is dynamic method lookup interesting? Consider what happens when you pass a "Vector3D" object to xAxisAngle(). That method calls "v.length()". Because Java uses dynamic method lookup, it checks the object that v points at, and discovers that it's a Vector3D object. So it calls Vector3D.length(), which correctly computes the length of v. If Java had called Vector.length() as usual, it would have ignored the z coordinate and computed the length wrong!

```
-------------------------------------------------------------------------------
| WHY DYNAMIC METHOD LOOKUP MATTERS             (Worth reading and rereading) |
|                                                                             |
| Suppose you've written a class that uses Vectors extensively.  By changing  |
| the constructors so that they create Vector3Ds instead of Vectors, many     |
| of the methods in your class automatically work on Vector3Ds as well,       |
| without change.                                                             |
-------------------------------------------------------------------------------
```