Today - let's look at a kind of case study in which interfaces, inheritance, and abstract classes are used in a realistic way. Take the graphing calculator example from before. We used this as an example of where using an interface made a lot of sense: The `GraphingCalculator` class can make a plot from any object representing a mathematical function, as long as it has the ability to compute a y-value for each x. This behavior or "capability" is represented by the simple interface `SVFunction`.

```java
public interface SVFunction {
    double yValue(double x);
}
```

Now, here's the scenario. The graphing calculator has been a tremendous success. With your bonus from the company's profits you've been able to buy a new Prius to toodle around Seattle with the other hipster programmers. Now, management is gearing up for a new release and comes in with a new feature request. Recall that each function we plot has one or more parameters that affect its values: For example, a linear function has a \textit{slope} and \textit{intercept}, a quadratic function has its three coefficients (often called "a", "b", and "c" in math books), and the sine function has an \textit{amplitude} and \textit{period}.

Management now wants us to add a new feature: each time a function is plotted, the plot window should display the name of the function and the values of its parameters or coefficients. As an example of what is desired, run `ugly.GCMain` and plot a linear function with slope 1 and y-intercept 0:
There is already a method in the plotter window for printing text on the plot, using a plotter method called `addText`. So, we just have to check which kind of function is being plotted and print its name and parameters.

In order to call the `getSlope()` and `getIntercept()` methods of the `LinearFunction` class, we have to downcast from `SVFunction` to `LinearFunction`, and in order to do that, we have to know in the first place whether it’s a `LinearFunction` type or not. Well, we can use the method `getClass()` or the `instanceof` keyword. The expression

\[
\text{x instanceof T}
\]

has value `true` if the type of `x` is a subtype of class/interface `T`. This idea leads to the code

```java
if (f instanceof LinearFunction)
{
    plotter.addText("Linear function");
    LinearFunction lf = (LinearFunction) f;
    plotter.addText("Slope: " + lf.getSlope());
    plotter.addText("Intercept: " + lf.getIntercept());
}
else if (f instanceof QuadraticFunction)
{
    plotter.addText("Quadratic function");
    QuadraticFunction qf = (QuadraticFunction) f;
    plotter.addText("Leading Coefficient: " + qf.getLeadingCoef());
    plotter.addText("Middle Coefficient: " + qf.getMiddleCoef());
    plotter.addText("Constant Term: " + qf.getConstantTerm());
}
else if (f instanceof SineFunction)
{
    plotter.addText("Sine function");
    SineFunction sf = (SineFunction) f;
    plotter.addText("Amplitude: " + sf.getAmplitude());
    plotter.addText("Period: " + sf.getPeriod());
}
```

This is **really ugly** code. Why is Steve so unhappy about this?? First of all, now in order to add new kinds of functions to our application, we have to go in and hack the `GraphingCalculator` code. Remember the original idea of defining the `SVFunction` interface: the `GraphingCalculator` really does not care what kind of function is being plotted, it just needs each function having the certain needed _behavior_, namely, being able to compute a y-value.

So let’s take a step back and see how we can redesign this. The point of polymorphism is that you should be able to invoke an operation on an object without knowing its exact type. Each kind of function being plotted (e.g. Linear, Quadratic, Sine, or whatever) should "know" what its _own_ parameters are and what they are
called, so that the GraphingCalculator can just ask the function object to display what it already knows about itself. Then all the GraphingCalculator has to do is to iterate over a list of parameters and print a description of each one, something like this:

```java
plotter.addText(f.getName());
for (Descriptor d : f.getParameters())
{
    plotter.addText(d.toString());
}
```

where a Descriptor would just be a simple data container for the name and value of one function parameter (like "Slope" or "Intercept"), e.g. something like this:

```java
public class Descriptor
{
    private double value;
    private String description;
    public Descriptor(double givenValue, String givenDesc)
    {
        value = givenValue;
        description = givenDesc;
    }
    // ... and so on
}
```

We can capture this needed behavior or capability in the improved better.SVFunction interface:

```java
public interface SVFunction
{
    double yValue(double x);
    ArrayList<Descriptor> getParameters();
    String getName();
}
```

So far so good. The GraphingCalculator class can now be written as above, to depend only on this interface, and does not need to ever check types, or downcast.

Then we start implementing classes such as LinearFunction to use this interface: We might first try it like this:

```java
public class LinearFunction extends SVFunction
{
    private double slope;
    private double intercept;

    public LinearFunction(double givenSlope, double givenIntercept)
    {
        slope = givenSlope;
        intercept = givenIntercept;
    }
```
@Override
public ArrayList<Descriptor> getParameters()
{
    ArrayList<Descriptor> params = new ArrayList<>();
    params.add(new Descriptor(slope, "Slope"));
    params.add(new Descriptor(intercept, "Intercept"));
    return params;
}

@Override
public String getName()
{
    return "Linear Function";
}

@Override
public double yValue(double x)
{
    return slope * x + intercept;
}

So far so good. But then we try doing the same thing for the SineFunction,

public class SineFunction implements SVFunction
{
    private double amplitude;
    private double period;

    public SineFunction(double givenAmplitude, double givenPeriod)
    {
        amplitude = givenAmplitude;
        period = givenPeriod;
    }

    public String getName()
    {
        return "Sine Function";
    }

    @Override
    public ArrayList<Descriptor> getParameters()
    {
        ArrayList<Descriptor> params = new ArrayList<>();
        params.add(new Descriptor(amplitude, "Amplitude"));
        params.add(new Descriptor(period, "Period"));
        return params;
    }

    @Override
    public double yValue(double x)
    {
        return amplitude * Math.sin((2 * Math.PI * x) / period);
    }
And we notice that most of the code exactly the same as in the LinearFunction class: we have to store the parameters, return a name, and return an arraylist of parameter descriptors. Only the names are different. This is a good opportunity to use inheritance to reduce the duplication. We take the code that is the same for both kinds of functions, and move it into a common superclass.

Take a look at the code for better.AbstractFunction. Here, we store the name and descriptor list, and each concrete subtype can just define its name and parameters during construction.

Note that since the common code can't actually compute a function's y-values, there is no sensible way to implement the yValue() method in the superclass. We'll declare the superclass to be abstract so that we can leave that function unimplemented. Note also the common practice for abstract classes of declaring the constructor to be protected, since only the subclasses can actually invoke it. We have also declared a couple of other methods to be protected, addParam() and getParameter(). These methods are not part of the public API, but are intended for the use of subclasses only.

Now take a look at better.LinearFunction - all it has to do in order to initialize itself is set up the name and parameter array list. Then look at QuadraticFunction: it looks exactly the same.

When we start to do SineFunction, we realize also that the cosine function, and tangent function, and cotangent function, will also look almost exactly the same - each has an amplitude and period as its parameters. So here is another opportunity to take the common code for SineFunction and CosineFunction and TangentFunction, and move it into a superclass we call AnyTrigFunction.

A common question at this point might be: if we have abstract classes, what's the point of having an interface? Isn't an abstract class "better", because it can contain actual code?

To try to answer this, notice that the interface and the abstract class served very different purposes here:

- we used an interface to describe a set of behaviors or capabilities that the GraphingCalculator requires. The purpose of using an interface is to reduce the potential coupling between GraphingCalculator and any particular classes that might come along to implement mathematical functions we might want to plot.

- we used inheritance to be able to reuse code among several different implementations of classes implementing the SVFunction interface. There still could be implementations (present or future) of SVFunction that aren't
based on our AbstractFunction implementation, and GraphingCalculator should still work with them.

Remember that a class can extend only one superclass, but it can implement many interfaces. Therefore, defining a type as an interface leaves the implementer much more flexibility.