Interfaces, Polymorphism, and Inheritance
• **PhoneContact** that *extends* contact

• *Inherits* the attributes and methods of Contact (the first name and last name; and the `getFirst`, `getLast` and `compareTo` methods).

• Adds new attributes and behavior (the *phone number* attribute and the `getPhoneNumber` method)

• *Overrides* the existing methods `toString` and `notify`.

• **PhoneContact** with the type **TextContact** that inherits the phone number attribute, but overrides the `notify` method

• Another extension of Contact called **EmailContact**.
Comparable interface

• All Contact objects have the "capability" of being ordered by a method called compareTo

• The declaration is Comparable<Contact> because the Comparable interface is declared to be generic: So you have to state the type to which you are going to compare your Contacts

```java
public class Contact implements Comparable<Contact>
{
}
```
**CompareTo method (String)**

- If they have different characters at one or more index positions, let \( k \) be the smallest such index; then the string whose character at position \( k \) has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string.

  *Return value: \( \text{this.charAt}(k) - \text{anotherString.charAt}(k) \)*

- If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string.

  *Return value: \( \text{this.length}() - \text{anotherString.length}() \)*
Observations

• The **notify** method can only be implemented when we know the concrete type of the contact

• Therefore we made the notify method **abstract**

• This forces us to declare the class itself to be **abstract**

• Contact serves as a "holder" for the code that is common to all types of contacts, but it can't actually be used on its own (**i.e. It can’t be instantiated**)
Graphing Calculator

• 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);
}
```
Functions and Modifications

- 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 \textbf{name of the function and the values of its parameters} or coefficients.
has value \texttt{true} if the type of \texttt{x} is a subtype of class/interface \texttt{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());
}
```
• To add new kinds of functions to our repertoire, we have to go in and hack the GraphingCalculator code
• 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());
    }
```
public class Descriptor
{
    private double value;
    private String description;
    public Descriptor(double givenValue, String givenDesc)
    {
        value = givenValue;
        description = givenDesc;
    }
    // ... and so on
public interface SVFunction
{
    double yValue(double x);
}

public interface SVFunction
{
    double yValue(double x);
    ArrayList<Descriptor> getParameters();
    String getName();
}
public class LinearFunction extends SVFunction {
    private double slope;
    private double intercept;

    public LinearFunction(double givenSlope, double givenIntercept) {
        slope = givenSlope;
        intercept = givenIntercept;
    }

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

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

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);
    }
}
Inheritance..

• 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.

• **AbstractFunction**: Here, we store the name and descriptor list, and each concrete subtype can just define its name and parameters during construction.
More on inheritance..

• 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.

• common practice for abstract classes of declaring the constructor to be protected, since only the subclasses can actually invoke it.

• 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.
• When we start to do \texttt{SineFunction}, we realize also that the cosine function and tangent 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 \texttt{SineFunction} and \texttt{CosineFunction} and \texttt{TangentFunction}, and move it into a superclass we call \texttt{AnyTrigFunction}
# Graphing Calculator

<table>
<thead>
<tr>
<th>Interface</th>
<th>Inheritance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe a set of <em>behaviors</em> or <em>capabilities</em> that the GraphingCalculator requires</td>
<td>Able to reuse code among several different implementations of classes implementing the SVFunction interface</td>
</tr>
<tr>
<td>Reduce the potential coupling between GraphingCalculator and any particular classes that might come along to implement particular types of functions we might want to plot.</td>
<td>Could be implementations (present or future) of SVFunction that aren't based on our AbstractFunction implementation, and GraphingCalculator should still work with them</td>
</tr>
<tr>
<td>Composition</td>
<td>Inheritance</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Has-a relationship</td>
<td>Is-a relationship</td>
</tr>
<tr>
<td>A Dog <strong>has a</strong> name</td>
<td>A Dog <strong>is a</strong> type of Pet</td>
</tr>
<tr>
<td>A PhoneContact <strong>has a</strong> phone number</td>
<td>A PhoneContact <strong>is a</strong> type of Contact</td>
</tr>
<tr>
<td>A Car <strong>has a</strong> gas tank.</td>
<td>A Car <strong>is a</strong> type of Vehicle</td>
</tr>
<tr>
<td>“has-a&quot; test</td>
<td>&quot;is-a&quot; test</td>
</tr>
<tr>
<td></td>
<td>class X extends Y</td>
</tr>
</tbody>
</table>
For example, "every car is a gas tank"
"every gas tank is a car"
This tell us that inheritance is *not* the way to implement the relationship between a car and a gas tank

"every car **has** a gas tank"
a car is *composed* of a gas tank and other attributes
Car, it will have an instance variable whose type is a gas tank, e.g.

```java
class Car
{
  private GasTank tank;

  // etc.
```