SLX 2 Status & Discussion

The following features are now implemented:

- subclasses
- abstract methods
- overridable methods
- interfaces
- "super" qualifier
- "protected" prefix

The following test programs are available:

File	Contents
inheritance.slx	A hodge-podge of basic tests
interfaces.slx	First cut at Java-style interfaces
protected.slx	Illustrates protected classes.
MAshapes	A test program contributed by [An SLX User]

Discussion Items

1. Currently, it's possible to instantiate a class that has abstract methods. If you attempt to invoke one of these methods for an instance of the class, you'll get a run-time error. Classes with abstract methods probably should be limited to use as base classes, i.e., instantiation should be disallowed.

2. [Unnamed SLX User] & I have different opinions on whether the "override" prefix should be required for concrete instances of abstract methods. My opinion is that the "override" prefix should *not* be required, since abstract classes are intended to be instantiated. An instantiation is the fulfillment of a contract, not an override. [The user's] position is as follows:

The "abstract" issue is not as clear cut as maybe it ought to be. I interpret an abstract method as being a virtual/overridable method that has no body – akin to what is termed a "pure virtual function" in C++. So in that sense, since the method has no body and must be overridden in order to instantiate sub-classes, an abstract method is also "required" - it must be overridden by a sub-class. By that definition, an "abstract" procedure has to be "overridden" explicitly.
The confusion comes when we look at methods defined as part of an interface. In Java, all methods are virtual by default, so although Java interface methods cannot be declared as "abstract", that is effectively what they are: they must be overridden (the closest Java comes to explicit overriding is the use of the @Override attribute from Java 1.5 onwards) in an implementing class, they are virtual and they have no body. By contrast, in C# – which SLX currently resembles more – interface methods are merely "required": they are not abstract and cannot be declared as such, and they cannot be overridden.
For example (C# code):

```
public class Color
{
}

public interface IColorable
{
   // setColor is required by implementing classes, but is not abstract
   void setColor (Color color);
}

public class Vehicle: IColorable
{
   // Non-virtual implementation of setColor – cannot be overridden by a sub-class.
   public void setColor (Color color)
```

```
    {
        // …
    }
}

public abstract class Thing: IColorable // Has abstract method, so class must be abstract too
{
    // Abstract implementation of setColor – must be overridden by a Thing sub-class.
    public abstract void setColor (Color color);
}

public class SomeThing: Thing
{
    // Overrides the abstract Thing::setColor method
    public override void setColor (Color color)
    {
        //…
    }
}

public class Ball: IColorable
{
    // Virtual implementation of setColor – may be overridden by a Ball sub-class.
    public virtual void setColor (Color color)
    {
        //…
    }
}

public class GolfBall: Ball
{
    // Inherits Ball::setColor.
}

public class Football: Ball
{
    // Overrides Ball::setColor.
    public override void setColor (Color color)
    {
        //…
    }
}
```

Given the syntax currently supported by SLX, the above is both a little complex (requires a differentiation between what it means to be "required" and what it means to be "overridable and required") and non-obvious. I believe it would be easier if we explicitly declare interface methods as abstract as we currently do, and require that we explicitly override them in implementing classes. The equivalent SLX code to the above would be:

```
private module ColorModule
{
    public class Color
    {
    }
    public interface Colorable
    {
        // setColor is abstract, must be overridden
        abstract method setColor (pointer (Color) color)
        {
        }
    }
```

```
    public class Vehicle
    implements (Colorable)
    {
        // Overrides and seals setColor – cannot be overridden by a sub-class.
        public sealed override method setColor (pointer (Color) color)
        {
            // …
        }
    }

    public class Thing
    implements (Colorable)
    {
        // Abstract implementation of setColor – must be overridden by a Thing sub-class.
        // No need to do anything – inherits abstract method.
    }

    public class SomeThing
    subclass (Thing)
    {
        // Overrides the abstract Colorable::setColor method
        public override method setColor (pointer (Color) color)
        {
            //…
        }
    }

    public class Ball
    implements (Colorable)
    {
        // Virtual implementation of setColor – may be overridden by a Ball sub-class.
        public override method setColor (pointer (Color) color)
        {
            //…
        }
    }

    public class GolfBall
    subclass (Ball)
    {
        // Inherits Ball::setColor.
    }

    public class Football
    subclass (Ball)
    {
        // Overrides Ball::setColor.
        public override method setColor (pointer (Color) color)
        {
            //…
        }
    }
}
```

If we treat abstract methods as being merely "required", rather than being "overridable and required", and issue warnings when they are overridden, then users will tend not to use "override" when defining implementations for abstract methods. (I certainly prefer my code to compile cleanly with no warnings or errors and suspect that others will do the same.) They will also not have the same protection that we currently enjoy for overridable methods: in the event that an abstract method is added to an interface or base class that clashes with the name of an unrelated method in an implementing class, the compiler will not issue a warning.

Use of an explicit "override" statement also clues anyone reading the code to the fact that the overridden method redefines an inherited method from a base class or interface, rather than merely being a new procedure.

The semantics of an overridable method and those of an abstract method are exactly the same, with the exception that the first has a default definition and the second doesn't.

3. [An SLX User] has raised questions about "protected" classes:

So in an SLX 2.0 world, our trunk models will contain all the base classes and interfaces required for our plug-and-play customizations to work with. One issue that I see coming is that all member variables in our base classes will need to be public for the sub-classes in different modules to have read-write access. In C++, the protected keyword allows sub-classes read-write access to base class members but hides the members to anything else. Aside from adding a protected keyword, perhaps it SLX 2.0 could automatically allow read-write access to private and read_only members in a base-class even when the sub-class is in a different module? It's not a huge deal, we can live with making all base-class members public but it would be nice to have more control if it's an easy (and sensible) thing for SLX to provide.

One difficulty about setting defaults for base classes is that the compiler doesn't discover that they're base classes until it encounters the first instance of their being subclassed. By the time that that happens, it may be too late to retroactively change defaults. The way I envision protected classes most often being used is with the "protected" prefix applied to the entire class, rather than individual members:

```
protected class widget
        {
        …
        };
```

If you have a bunch of such classes, the "protected" prefix can be applied to their containing module:

```
protected module MyClasses
        {
        …
        };
```

Finally, let me remind you of a recent addition to SLX. SLX has provided the ability to create and run ".rts" (run-time SLX) files for a long time. RTS files also can be written with security key protection so that only users who have a key with the right security code can run them. With one exception, source code from which an RTS file was built is no longer visible. The exception is that if an RTS file generates an error, e.g., division by zero, the source code in the immediate vicinity of the error is displayed. There is no back-door access through the SLX debugger.

The recent addition is that RTS files no longer have to be complete SLX programs. They can be a collection of SLX source files that lack a "main" procedure. RTS files can be used in "import" statements, e.g.,

```
import   MyLibrary.rts
```

For those of you who provide code for others to use, but wish to maintain the confidentiality of that code, wrapping the code into RTS files is a solution. Given the flexibility of abstract methods, overridable methods, protected classes, etc., we're moving in the direction of allowing some very sophisticated packaging.

4. To an extent, SLX 2 steals features from a variety of sources. Stealing from one source is called plagiarism. Stealing from many is called research. I hope that the end result is not a hodge-podge, but rather, a collection of the most useful features provided in other languages. Since many SLX users are not OOP gurus, it's important to keep things as simple, small, and general as possible.

Let's keep the discussion going. I've already gotten lots of good suggestions from a number of you.

Regards,
Jim