Using SLX Methods

1.0 What's an SLX Method?

An SLX method is an SLX procedure conceptually (if not physically) contained *within* an SLX class. In the following example, "inci" is a method of the class "widget":

```
class widget
    {
    int  i;

    method inci(int increment)
        {
        i += increment;
        return;
        }
    };
```

2.0 Advantage #1 of Using Methods

Within a method, elements of the method's containing class can be used without a qualifying pointer. This is possible for two reasons. First, SLX always tries to find the declaration for a variable by first searching the current scope and any outer, containing scopes. When the compiler sees "i" within "inci", it first looks for a declaration of "i" within "inci". Finding none, it searches the outer scope of "inci", which is the scope of "widget". Here, it finds a declaration for "i". Thus, SLX knows that "i" within "inci" is an element of "widget". Second, when a method is called, SLX passes a hidden zeroeth argument that is a pointer to the instance of the class that contains the method. This hidden pointer is used to access elements of the class.

The alternative would be to pass an explicit pointer argument identifying the object to be operated upon:

```
method inci(pointer(widget) w, int increment)
    {
    w -> i += increment;
    return;
    }
```

Such a procedure could be placed anywhere. It needn't be placed within class widget, since it makes no implicit references to elements of class widget.

3.0 Calling Methods

3.1 Calling a Method from Outside its Class

To call a method from outside its class, one must prefix the method name with a pointer or "dot" prefix that identifies the particular object on which the method is to operate:

```
procedure main()
    {
    pointer(widget)     w;
    widget              localw;

    localw.inci(2);

    w = new widget;
    w -> inci(3);
    }
```

In the above example, we see inci applied to a local instance of class widget, named "localw" and to a dynamically created instance pointed to by "w".

3.2 Calling a Method from Within its Class

When a method is called from within its class, one need not qualify the method name with a prefix, since SLX assumes the method is to be applied to the current instance of the class. Alternatively, you can qualify such a method call by using "ME". Assume that class widget included the following actions property:

```
actions
    {
    inci(5);
    ME -> inci(10);
    }
```

The two approaches shown above are functionally equivalent. In the unqualified approach, the passing of ME is implicit. In both cases, ME is passed as a hidden, zeroeth argument that does not appear in inci's definition.

4.0 Using the Same Method with More than One Class

Suppose you're building a model of a traffic grid, and your model contains classes for cars, trucks, and buses. Further suppose that you wish to construct an "EnterGrid" method for each of these classes. EnterGrid performs an identical function for each of the three classes, but because the objects entering the grid differ from one another, the implementations of each of the three versions of EnterGrid may differ considerably.

You can call such methods using either class-specific pointers or universal ("pointer(*)") pointers:

```
pointer(Car)    c;
pointer(Bus)    b;
pointer(Truck)  t;
pointer(*)      u;

c = new Car;
b = new bus;

c -> EnterGrid(…);
u = b;
u -> EnterGrid(…);
```

The first call of EnterGrid shown above is easy for SLX to interpret, since it's obvious that the version of EnterGrid to be called is the version defined for class Car. The second call is more difficult. SLX must examine the object to which "u" points, and select the version of EnterGrid for the object's class. SLX is able to do so in a fixed sequence of instructions. (Method selection is accomplished by a direct lookup, not a search.) Thus, the overhead of method selection is extremely low.

To use methods specified by universal pointers, the following rules apply:

A. The methods must be identically named.
B. The methods must have identical argument lists.
C. The methods must return identical types (or all be typeless).

The following errors can occur when calling a method specified by a universal pointer:

A. The pointer may point to a class that doesn't include the method.
B. The pointer may be NULL.

SLX traps both of these errors.

5.0 Advantage #2 of Using Methods

Specifying methods by means of universal pointers eliminates the need for manual method selection; i.e., you do *not* have to do the following:

```
If ("object is a Car")
    Call the Car method

else if ("object is a Truck")
    Call the Truck method
```

When you specify a method by using a universal pointer, you achieve an ideal notation: your intent is clear, while implementation details are hidden. When you look at

```
u -> EnterGrid(…)
```

You can see the intent without having to worry about details.

6.0 Defining Methods Outside Their Classes

SLX provides two ways of defining methods outside their classes. However, you should think of both approaches as temporarily placing the method being defined back into the context of the class in which it is conceptually, but not physically contained. In other words, even when a method is defined outside its class, it acts as if it were physically contained within its class.

The first approach is to use SLX's rarely used "**augment**", which allows you to add things to a class after the class has been defined. For example, if we had not included method inci in class widget, we could do so as follows:

```
augment widget
    {
    method inci(int increment)
        {
        …
        }
    }
```

The augment verb temporarily places your program back in the widget class. Note that augment allows you to do considerably more than add methods. For example, you can add elements to a class; you can add a property, e.g., initial; or you can append to or prefix an existing property.

The second approach is to use OOP-inspired notation:

```
method widget::inci(int increment)
    {
    …
    }
```

In this notation, "widget::" accomplishes a sufficient subset of "augment widget", using a notation known outside the wild, wonderful world of SLX.

Methods for classes defined within precursor modules must be included within the precursor module. In other words, once a precursor module has been compiled, extending its classes by adding methods outside the precursor module is not allowed.

7.0 Pseudo-Methods

To qualify as a true method, a method must include unqualified references to elements of its containing class. However, it's possible to code methods that include no such references. Such methods are pseudo-methods, since they look like ordinary methods and are called like ordinary methods, but have no "real" requirement for specifying which object of a given class they are to operate upon, because they contain no implicit references to their classes.

For historical reasons, SLX is lenient in its handling of pseudo-methods. It allows you to call them without pointer or "dot" qualifications; i.e., you can call them as ordinary SLX procedures.

8.0 Debugger Support

Within a method, the SLX debugger's Local Data (Dynamic Selection) window contains a pseudo-variable designated as "(*class name*)". For example, if you step into the inci method, the Local Data window will contain a pseudo-variable named "(widget)". By right clicking on "(widget)", you can explore the particular widget currently operated upon by inci.

9.0 An Example

Source code for the following example, methods7.slx, is available from Wolverine:

```
//*****************************************************************************
//          SLX Method Examples
//*****************************************************************************

public precursor module Methods1
        {
        class widget
                {
                int                     i;
                double x;

                actions
```

```
                        {
                        inci(5);
                        ME -> inci(10);
                        }
//                      method inci(int increment, int incompatibility)
//                      method inci(int increment[*])
//                      method inci(int increment, pointer(widget) w)
                        method inci(int increment)
                                {
                                i += increment;
                                return;
                                }

                        method fetch_i returning int
                                {
                                return i;
                                }
                        };
        }

module Methods2
        {
        class widget3
                {
                int     i, inci;
                };

        class widget2
                {
                int                     i;
                double x;

                method inci(int increment)
//              method inci(int increment, pointer(widget2) w2)   // compile-time error
                        {
                        i -= increment;
                        return;
                        }
                };

        method widget2::fetch_i returning int                                   // OOP-style
                        {
                        return i;
                        }

        augment widget2
                {
                method store_i(int j)
                        {
                        i = j;
                        }
                }

        method two() returning int
                {
                return 2;
                }
```

```
	method delay() returning int
		{
		yield;
		return 1;
		}

	method main()
		{
		pointer(widget)		w;
		pointer(widget2)	w2;
		pointer(*)		u;
		int			i;

		widget			localw;
		widget2			localw2;

		localw.inci(2);
		localw2.inci(2);

		w = new widget;

		activate w;
		yield;

		w2 = new widget2;

		w -> inci(3);
		w2 -> inci(3);

		u = w;
		w = u;
		u -> inci(3);
		i = u -> fetch_i();

//		u -> store_i(27);		// run-time error

		u = w2;
		u -> inci(3);
		i = u -> fetch_i();

//		u -> inci(delay());		// compile-time error

		u = NULL;
//		u -> inci(3);			// run-time error

		u = new widget3;
//		u -> inci(3);			// run-time error

		u = NULL;
//		u -> inci(two());		// run-time error
		}
	}
```