How to Pass Arguments to a Stored Function

Brian Willkie

In chapter 18, section 3.3, we looked at supporting different types of objects within the NRT_TimeFrame class, specifically SimpleNumbers and Functions. In this supplemental website extra, we look at ways to pass arguments to those functions using Object's performList method to expand an array into the arguments of a function.

1.1 Basic Problem

If the user provides a function for starttime or duration, s/he may want to pass arguments to that function.

```
a = NRT_TimeFrame.new(starttime: {arg x; x/2;});
a.starttime(7);
```

Figure 18.28

The problem is that we have no way of knowing how many arguments the user may want to pass. One solution is to simply pass all of the provided arguments as an array to the function.

```
starttime {arg ... args;
    ^ starttime.value(args);
}
```

Figure 18.29

This however requires the user to access the array to get to the arguments.

```
a = NRT_TimeFrame.new(starttime: {arg args; args[0]/2;});
```

Figure 18.30

1.2 Solution using performList

There is an alternative however. In addition to the value method, Object implements a method called performList. Given a method name and arguments, performList invokes that method with the given arguments. If the last argument is a List or Array, its elements are unpacked and passed individually to the given method.

```
a = {arg x, y; x * y;};
a.value(2, 3);
a.performList(\value, [2, 3]);
```

Figure 18.31

Since Object implements performList, everything in SuperCollider responds to it. Of course, whatever we pass performList to has to implement the given method (i.e. you can't expect to call **"Hello".performList(\bufnum);** and get a reasonable result, since String doesn't know anything about Buffer IDs). So with this method, we can write:

```
starttime {arg ... args;
    ^starttime.performList(\value, args);
}
```

Figure 18.32

We can pass ‘value’ as the method name to performList. We can then collect any arguments the user passes to the starttime method in an array (args), and pass that array as performList’s last argument. performList unpacks args and passes the individual arguments to starttime.value.

We can apply the same approach to duration.

```
duration {arg ... args;
    ^duration.performList(\value, args);
}
```

Figure 18.33

1.3 Special Considerations for endtime

The endtime method requires a little extra work. First of all, notice that we have to call this.starttime (which now takes a set of arguments) and this.duration (which takes a different set of arguments). This means that endtime has to take two sets of arguments. One way to handle this is with two arrays.

So now that we’ve put all this work into building starttime and duration methods that accept individual arguments (i.e. not arrays), inside the endtime method we only have the startArgs and durArgs arrays to pass. All is not lost, we can use performList to bail us out again. This time however, we do not want to call value, we want to invoke this.starttime and this.duration. Since NRT_TimeFrame implements performList (well, its parent, Object, does), we can call it with this.performList. Likewise, since NRT_TimeFrame implements methods called starttime and duration, we can pass those method names as the first argument to this.performList. Then we simply have to pass the appropriate variables (startArgs, or durArgs) to the respective calls to performList[i].

```
endtime {arg startArgs, durArgs;
    var start, dur;
    //use performList to call this TimeFrame’s method "starttime"
    // and pass in the array of arguments for starttime, but only if
    // there is one
    startArgs.notNil.if(
        {start = this.performList(\starttime, startArgs);},
        {start = this.starttime;}
    );
    //do the same for this.duration
    durArgs.notNil.if(
        {dur = this.performList(\duration, durArgs);},
        {dur = this.duration}
    );
    ^(start != nil).if({
        (dur != nil).if({
              //if the dur is inf, then endtime is nil
            (dur != inf).if({start + dur;}, {nil}
        }, {nil})
    //else start is nil
    }, {nil});
}
```

Figure 18.34

Now, putting all of that together, we have our new NRT_TimeFrame class definition:

```
NRT_TimeFrame {

	var >starttime, >duration;

	*new {arg starttime, duration;
		^super.newCopyArgs(starttime, duration);
	}

	starttime {arg ... args;
		^starttime.performList(\value, args);
	}

	duration {arg ... args;
		^duration.performList(\value, args);
	}

	endtime {arg startArgs, durArgs;
		var start, dur;
		//use performList to call TimeFrames's method "starttime"
		// and pass in the array of arguments for starttime, if there is one
		startArgs.notNil.if(
			{start = this.performList(\starttime, startArgs);},
			{start = this.starttime;}
		);
		//do the same for this.duration
		durArgs.notNil.if(
			{dur = this.performList(\duration, durArgs);},
			{dur = this.duration}
		);
		^(start != nil).if({
			(dur != nil).if({
				(dur != inf).if({start + dur;}, {nil})
			}, {nil})
		}, {nil});
	}

}
```

Figure 18.35

Now we can store the relationship, not just a value. In an abstract sense, we are just avoiding hard-coding in that we don't provide a specific value, but rather a way to derive a value, a way for the computer to fill in the blanks.

[i] See the SuperCollider help file, Syntax Shortcuts, for a more succinct way to call performList.