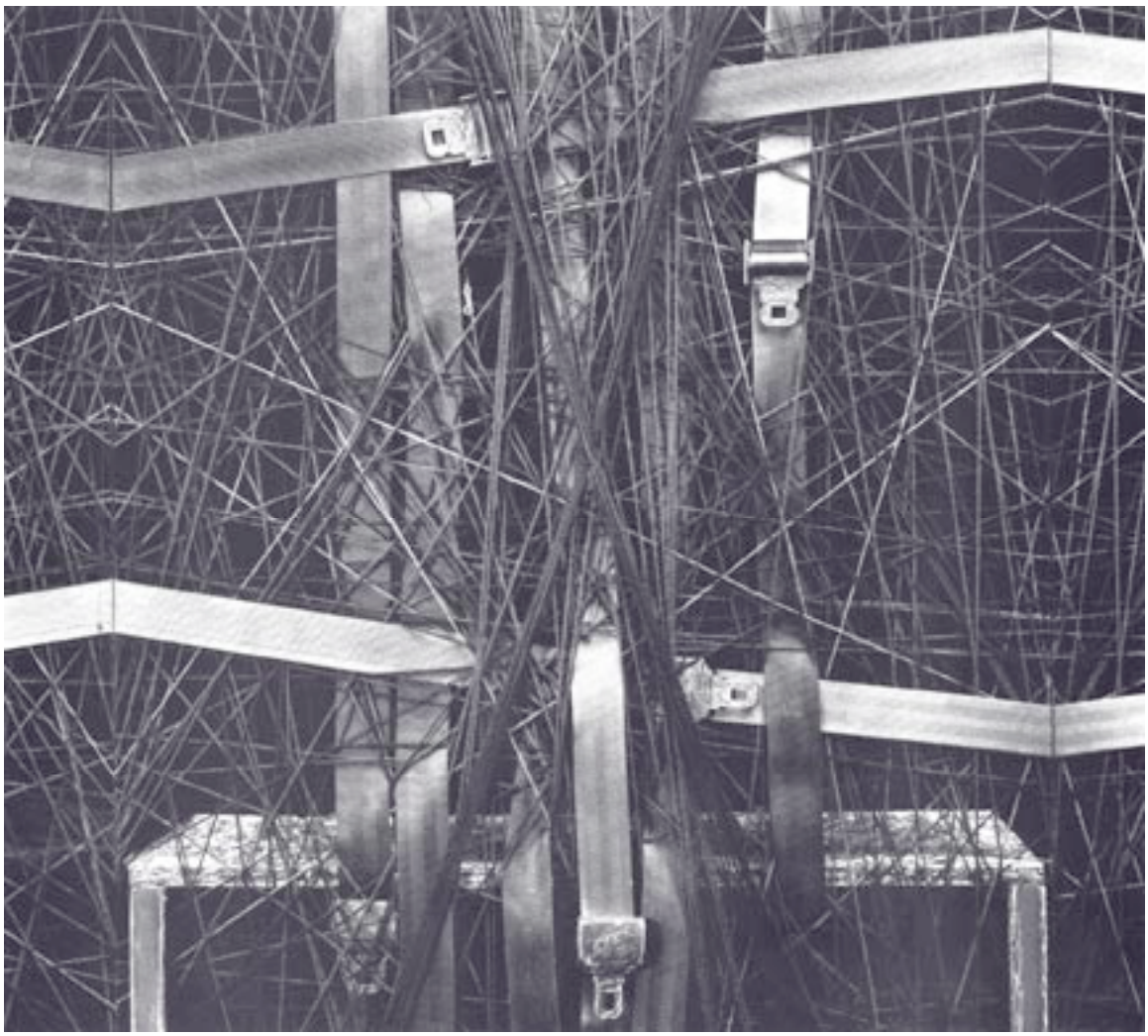
Max/MSP



Writing Max Externals in Java

V 0.3

by Topher La Fata
topher@cycling74.com

Table of Contents

Copyright and Trademark Notices

Credits

Content: Topher LaFata and Joshua Kit Clayton (Appendix A)

Cover Design: Lilli Wessling Hart

Graphic Design: Gregory Taylor

Overview

With the release of Max/MSP 4.5, the Java programming language has been made available to developers/users who wish to extend the functionality of the Max/MSP environment beyond what is provided by the Max visual programming language itself in the form of Max, MSP and Jitter externals.

Externals are code resources—generally written in the C programming language—that are loaded into the Max environment at runtime and are called upon by the Max application to provide some specific functionality to the user. For instance, the Max **print** object is a Max external written in C that lets a user print messages to the Max console from within a patcher. Most of the boxes you connect together in a typical Max patcher are externals written in C. By authoring your own externals in C or Java you are able to extend the core functionality of the Max visual programming language in whatever way suits your needs.

Why write a Max external in Java?

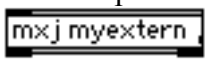
Java externals are cross-platform. You need only to develop and maintain a single binary distribution for both the Windows and OS X versions of Max/MSP.

Java, as a programming language, lends itself nicely to rapid prototyping. Java handles memory management for you. Thus, you do not need to worry about memory allocation issues present in C or crashing as a result of dereferencing bogus pointers.

Java is object oriented which often aids in software design and promotes code reuse. You can also take advantage of the feature rich class library that is standard with all Java distributions, and leverage a wealth of code available on the internet to quickly provide functionality that would have taken much longer to replicate in C. For example, Java has a rich networking Applications Programming Interface (API) that is in many ways easier to deal with than a C language equivalent.

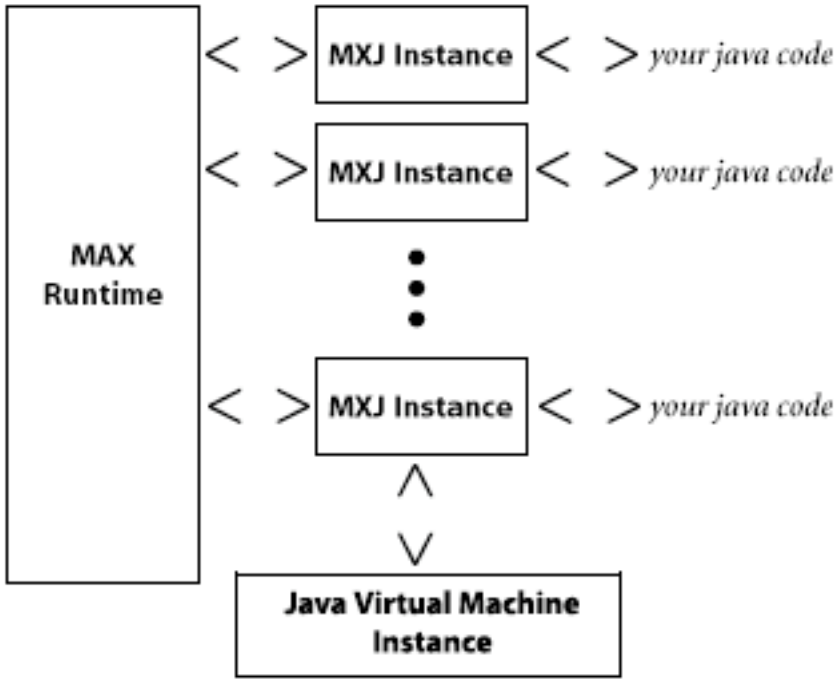
Cycling ‘74 also provides a simple integrated development environment that allows you to interactively build and compile Java externals without ever leaving the comfort of your Max patch. Changes can more or less be seen immediately—there’s no need to quit and restart Max if you want to change one line of code in the external object you’re developing.

Accessing the functionality of Java externals in a Max patcher is slightly different than accessing the functionality of a Max external written in C. C externals are loaded directly into the Max application environment at runtime, but Java externals are loaded into the Max environment and thereafter communicate with the Max environment through a proxy external object called **mxj**.

For instance, if you had an external named "myextern" written in Java you would need to access the functionality of "myextern" in your Max patch by entering the name of your external as an argument to the **mxj** object: 

instead of typing **myextern** into an empty object box as you would if your external had been written and compiled in C.

mxj is a Max external written in C that interacts with an instance of the Java virtual machine (JVM) within the Max application environment and provides the glue that allows calls from Max to be dispatched to your Java code through the JVM and vice versa. Only one JVM is ever created over the lifetime of your Max application and all **mxj** instances share this single JVM instance to execute their given functionality.



This document assumes that you have a basic knowledge of the Java programming language. It is possible that if you are familiar with another programming language such as C/C++ and Max that you may be able to glean all you need know about authoring Max externals in Java from this document. There are also many great Java programming resources on the web. The best place to start may be at Sun Microsystems Java website (http://java.sun.com).

If you are having problems and/or are curious about something not explained in this document, the Max/MSP list hosted by Cycling ‘74 can be an invaluable community-

based resource for your questions. It is also a great place to distribute and share all the cool work that you have done using **mxj**/Java with others in the Max community.

Basics

Setting Up Your Development Environment

Although Cycling ‘74 provides a simple environment for developing Java externals entirely from within Max you may prefer to use a third-party development environment to write your Java code. Here are pointers to a few:

Eclipse	http://www.eclipse.org
BlueJ	http://www.bluej.org
JBuilder Foundation	http://www.borland.com
XCode(OS X only)	http://www.apple.com/xcode

No matter which development environment you choose it is essential that your project CLASSPATH settings include the Cycling ‘74 specific library that will enable you to compile Java classes that can execute within Max and be loaded by the **mxj** proxy external. This notion of CLASSPATH may be expressed differently depending on which environment you choose.

The specific library that you need to add to your CLASSPATH is named **max.jar**. In Macintosh OS X this file is located at:

/Library/Application Support/Cycling '74/java/lib/max.jar

The default location for this library on Windows is:

c:\Program Files\Common Files\Cycling '74\java\lib\max.jar

The max.jar library implements the application programming interface (API) that allows your Java class to handle communication to and from the Max application environment. A set of html **javadocs** - documentation generated from the Java source code of the library - are located in the directory where you installed Max 4.5. These documents are an essential reference for developing Max externals in Java as they detail every function available to communicate with Max/MSP. You may want to bookmark them in your web browser since you will most likely be referring to them often.

The mxj CLASSPATH

For the **mxj** proxy to find Max Java classes that you develop you need to place your compiled Java class files in a directory that is within the **mxj** CLASSPATH. This is different from the CLASSPATH you set up in your development environment.

The **mxj** CLASSPATH is where the **mxj** proxy external searches to find Java classes that you wish to use within Max/MSP. In most instances the simplest place to put your compiled classes on OS X is:

    /Library/Application Support/Cycling '74/java/classes

and on Windows

    c:\Program Files\Common Files\Cycling '74\java\classes

This directory, by default, is always within the **mxj** CLASSPATH. To customize the CLASSPATH that **mxj** uses to locate compiled Max Java classes, have a look at the file max.java.config.txt located in the C74 Java support directory. It details various options on how you can customize the CLASSPATH which **mxj** uses to locate your Java classes. Various options for the Java runtime of Max/MSP are exposed via this config file as well such as additional options to pass to the JVM at startup.

Additionally, when loading a patcher file from a directory not in the CLASSPATH **mxj** will search for Java classes in the directory that contains the patcher and all of its subdirectories. It is worthwhile mentioning that the terms "Java externals" and " Java classes" can be used interchangeably since a Java external is simply a Java class (or set of classes) which accesses the functionality of the Max Java API in the max.jar library.

mxj quickie - Integrated Java external Development From Within Max

The simplest way to get up and running authoring Java externs for Max is by using the **mxj quickie** development environment provided by Cycling ‘74. For all but the most complex projects, this environment will be sufficient. When using **mxj quickie** all Java development is done from within Max itself. This provides a very fluid experience for coding up a Max Java external on the spot.

mxj quickie also provides immediate source-level access to any Java external living in the Max environment. If you find yourself in the middle of a project and you need to extend the functionality of a new or existing Max Java class, you can jump directly into that class’s Java source code and compile a new version with the required functionality.

Even if you do not use **mxj quickie** for the initial development of your Max Java external, it can still be an invaluable tool for quickly fixing a bug or adding functionality to an existing Java extern. **mxj quickie** was developed as a Java Max external itself and serves as a real-world demonstration of the possibilities of extending Max with Java.

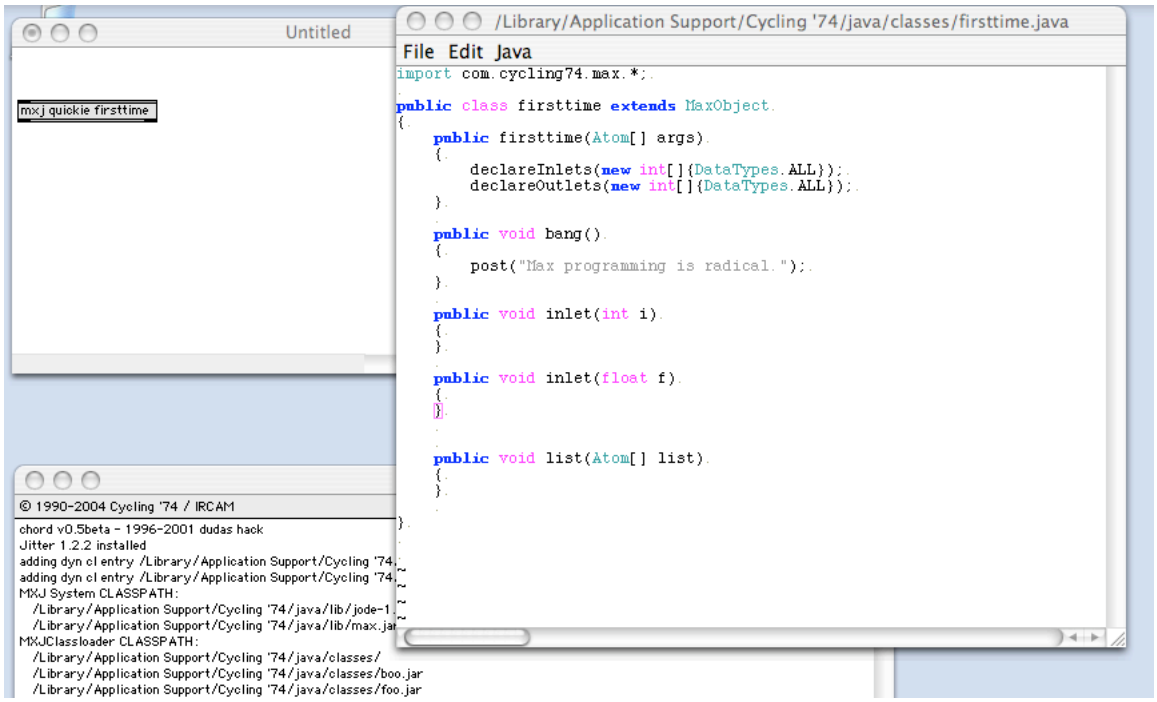
The most appropriate way to demonstrate the functionality of **mxj quickie** is to present a simple use case in which we will develop a fully functional Max Java extern. This Java external will print "Max programming is totally radical." into the Max console window upon receiving a bang message in its first and only inlet. The source-level details will be explored later – for now we are just presenting a step-by-step demonstration of how to use **mxj quickie** to compile a fully functional Java external called firsttime.

Start the Max Application.

Create a new Max patcher.

Make a new box and type **mxj quickie firsttime** into it.

Double click on the newly created **mxj quickie firsttime** box (or send a bang into its inlet).



At this point a text editor will pop up. It will be filled out with a default code template for developing Max Java externs. The first time you open the editor it may take a little while since the JVM needs to initialize its windowing subsystem. During this time the display

of the **mxj quickie firsttime** box in your patcher will cycle through some different colors to let you know it is working.

Notice that the title bar has the location of a file named firsttime.java. This Java source file will not actually be saved until you try to compile the editor buffer or save the contents of the editor buffer manually using File -> Save in the editor menu. Therefore if you accidentally typed **mxj quickie firsyime** into the object box, you can close the editor window, go back and type the correct class name **mxj quickie firsttime** and double click or bang the box again, and no extra files will have been saved to disk.

The code template in the buffer will have an empty method called bang. Defined as:

```
public void bang()
{
}
```

Within the body of the method type:

```
post("Max programming is radical.");
```

So that it looks like:

```
public void bang()
{
   post("Max programming is radical.");
}
```

Choose "**Open Compile Window...**" from the Java menu of the text editor. You will be presented with a dialog that enables you to compile the Java source buffer in the text editor. Look closely at the field titled "**Compiler Command:**". It should point to the location of the Java compiler you wish to use to compile the source buffer. If the command is wrong or the file it points to does not exist you can use the "**Browse...**" button to choose the location of the Java compiler you wish to use.

It is important to note on Windows machines that no Java compiler is installed by default. To install a Java compiler you should download the Java Developer Kit (JDK) for Windows from http://java.sun.com. Currently, **mxj** supports JDK versions of 1.4 or greater based on the Sun Microsystems JVM code base.

The "**Compiler Command:**" field is sticky. This means that you only have to set it once and **quickie** will remember to use the same command from that point forward. If you ever need to change it just choose "**Browse...**" again in the Compile Window and the new setting will be remembered.

Finally, press the "**Compile**" button in the Compile Window and your firsttime.java source file will be compiled and the resulting class file will be placed in the default **mxj** classes directory. The compile window will print out the command it is using to compile your source file and any resulting messages that occur during the compilation process.

If you happen to have mistyped something you may get an error in the message section of the Compile Window. In this case you can double click on the line number of the error and you will be brought back to that line the in source editor. Fix whatever erroneous syntax may have been introduced and open the compile window again.

Note that you can open the compile window from the editor using the keyboard shortcut Command-k in OS X or Control-k in Windows. You can also toggle back to the editor window from the Compile window by using the key shortcut Command-e in OS X and Control-e in Windows.

You have now successfully compiled your first Max Java external. You can close the editor window at this point. To see your newly created Java external in action create a new box in your Max patcher named [**mxj firsttime]** . Add a button to your patcher and connect it to the inlet of **[mxj firsttime]**. Click the bang widget. You should see the message "Max programming is radical." displayed in the Max application console window.

Any time you want to create a new instance of this external you can type [**mxj firsttime]** into a new object box. If you wish to further modify the **firsttime** external and add new functionality you can create an **[mxj quickie firsttime]** object and double click it. You will be presented with the last saved firsttime.java source file for further editing. If you remove firsttime.java and firsttime.class from the classes directory, after launching the **quickie** editor you would be presented with the default template once again.

NOTE: If you change your firsttime class and recompile it, the updated class needs to be reloaded off of disk before any changes will be reflected. That is, you will need to create a new instance of **[mxj firsttime]** in your patch, or force **mxj** to reload the latest version of your class by retyping the name of your class into the same **mxj** box. Any instances of the external created with the old class file will continue to operate with the old code.

Writing Java Externals

This document aims to provide an overview of key aspects of Max external development in Java as opposed to detailed survey of the entire Max Java API itself. For a detailed explanation of all functions and classes contained in the Max Java API please consult the javadocs in the Max application install directory.

Writing the Constructor

For the **mxj** bridge to instantiate an instance of your Java class, the class needs to be a subclass of `com.cycling74.max.MaxObject`. If you look at the javadoc for `MaxObject` you can see that it exposes a variety of functions that enable your object to communicate with the Max application environment. By subclassing MaxObject you have access to all these functions and can even override them with your own implementations.

```
public class scale extends com.cycling74.max.MaxObject
{
        private float scale_factor; //scale member variable of
                                    //the scale class
```

or the preferred method...

```
 /* import the supporting classes from the max.jar API so you
 can refer to them without the full package justification. */

import com.cycling74.max.*;
public class scale extends MaxObject
{
   private float scale_factor; //scale member variable
                               //of scale class
```

Although method overloading in Java externs is generally prescribed against for various reasons explained later, the constructor is one place where overloading is actually suggested and is useful. Let's assume that you want to force the user to explicitly provide a scale factor as an argument to your class when instantiating it via a Max box e.g [**mxj scale 1.2**].

```
public scale()
   {
      bail("(mxj scale) you must provide a default”+
           ” scale factor e.g. [mxj scale 0.5].");
   }
public scale(float scale_arg)
   {
      scale_factor = scale_arg;
```

```
    }
```

If a user types only **mxj scale** into a Max box the empty constructor will be called since no instantiation arguments were supplied. In this case the **bail** method of `MaxObject` is used to prevent instantiation of the object and print an error message to the Max console informing the user why instantiation failed. The unusable 'ghost' box created in the patcher window will have no inlets or outlets. This is often the desired behavior if the user does not supply you with enough information to make a functional instantiation of your class.

A more realistic example of constructor overloading would be something like the below class **foo** which has one required argument and two optional ones. In this case 'thename' is the required argument.

```
public class foo extends MaxObject
{
   //member variables
   String name;
   float velocity;
   int factor;
   public foo()
   {
      bail("(mxj foo) must provide a name for"+
           " foo e.g. [mxj foo cheesecake]");
   }
   public foo(String thename)
   {
      this(thename, 1.2, 40);
   }
   public foo(String thename, float thevelocity)
   {
      this(thename,thevelocity,40);
   }
   public foo(String thename, float thevelocity,
              int thefactor)
   {
      name    = thename;
      velocity = thevelocity;
      factor     = thefactor;
   }
}
```

In this scheme the following patcher boxes will all create valid instances of **foo** in your Max patcher:

[mxj foo goober]

[mxj foo goober 0.7]

[mxj foo goober 0.7 42]

While **[mxj foo**] would result in an error message and a 'ghost' box since the code makes 'thename' a required arg.

Constructors and Argument Coercion

Another point worth noting at this time is that the arguments passed into your constructor from the Max environment will be coerced into the types that you declare. If you declare your constructor as

public foo(float x, float y, float z)

and a user types **mxj foo 1 0 1** into a Max box, the arguments 1, 0 and 1 , although integers, will be automagically coerced into the floating point numbers 1.0, 0.0 and 1.0 before being passed into your constructor.

All of the primitive Java types are supported in this constructor coercion scheme, so all of the following are valid constructors:

```
public foo(int i, float f, String s)
public foo(short s)
public foo(boolean b1, boolean b2, byte b)
public foo(int[] intarray)
public foo(float[] floatarray)
```

etc..

Note that when your constructor takes an array primitive as an argument that it has to be the first and only argument to your constructor. Thus

```
public foo(int[] ia1, int[] ia2)
```

and

```
public foo(float f, int[] ia)
```

are not legal constructors for a Max Java external.

Declaring Inlets and Outlets

The next task while writing the constructor is letting the **mxj** bridge know how many inlets and how many outlets it should create for your class. This is done using the `declareInlets` and `declareOutlets` methods of `MaxObject`. `declareInlets` and `declareOutlets` take as arguments an integer array that describes the types of

inlets or outlets to be created in left to right order. The types of inlets and outlets that you are allowed to create are defined as static final variables in the class `com.cycling74.max.DataTypes`. The data types you will normally work with are:

`DataTypes.INT`	inlet/outlet accepts integers
`DataTypes.FLOAT`	inlet/outlet accepts floats
`DataTypes.ALL`	inlet/outlet accepts strings, integers and floats

Thus, if we want our external to have 2 inlets and 2 outlets we would have the following method calls in the body of our constructor:

```
import com.cycling74.max.*;
public class foo extends MaxObject
{
   /*member variables */
   private float _x;
   private float _y;
   private float _z;

   public foo()
   {
      bail("(mxj foo) must provide x,y, and z arguments");
   }

   public foo(float x, float y, float z)
   {
      _x = x;
      _y = y;
      _z = z;
      declareInlets(new int[]{ DataTypes.ALL, DataTypes.ALL});
      declareOutlets(new int[]{ DataTypes.ALL, DataTypes.ALL});

   }
}
```

The Info Outlet

By default all instances of Max Java externals have an extra outlet known as the **info outlet**. It is always created as the rightmost outlet of your external and is used to report values of attributes. If you have no need for the info outlet you can suppress its creation by calling the function createInfoInlet(false) in your constructor.

```
public foo(float x, float y, float z)
   {
      _x = x;
      _y = y;
      _z = z;
      declareInlets(new int[]{ DataTypes.ALL, DataTypes.ALL});
      declareOutlets(new int[]{ DataTypes.ALL, DataTypes.ALL});
      createInfoOutlet(false);
   }
```

Creating Max Java Externs With No Inlets or No Outlets

By default your Max Java external will always be created with one inlet and one outlet in addition to the info outlet mentioned above. If you wish to have an external with no inlets or no outlets you can use the `NO_INLETS` and `NO_OUTLETS` member variables of the `MaxObject` class.

```
declareInlets(NO_INLETS); //create no inlets
declareOutlets(NO_OUTLETS); //create no outlets
```

Note: In the instance that you are creating a Java signal processing external (see Writing Signal processing Objects in Java using mxj~) it is impossible to create an external with no inlets. One inlet will always be present for handling messages to your external.

Adding Assist Messages for Inlets and Outlets

If you are planning on sharing your Java external with other members of the Max community, you can provide assistance messages that describe what the function of each inlet and outlet is for your external. These messages appear in the Max application status bar when you mouse over an inlet or outlet in an unlocked Max patcher. It is good practice to provide these messages even if just for yourself. Who knows if you will remember which inlet or outlet does what in 2 weeks from now much less 6 months from now!

The easiest way to accomplish this is by using the `setInletAssist` and `setOutletAssist` functions of `MaxObject`. They both take an array of Strings describing the inlet or outlet in left to right order.

```
public foo(float x, float y, float z)
   {
      _x = x;
      _y = y;
      _z = z;
      declareInlets(new int[]{ DataTypes.ALL, DataTypes.ALL});
      declareOutlets(new int[]{ DataTypes.ALL, DataTypes.ALL});
```

```
        setInletAssist(new String[] { "bang to output",
                                      "bang to reset"});
        setOutletAssist(new String[] { "calc result 1",
                                       "calc result 2"});

    }
```

Shortcuts for Declaring Inlets and Outlets

It is worth mentioning that utility functions are provided in `MaxObject` to make declaring the number of inlets and outlets less cumbersome syntactically. These are `declareIO` and `declareTypedIO`. Say you want to have an external with 3 inlets and 2 outlets. You could use `declareIO` as an alternative to the `declareInlets` and `declareOutlets` calls. `declareIO` create inlets/outlets of `DataTypes.ALL`.

```
    public foo(float x, float y, float z)
    {
        _x = x;
        _y = y;
        _z = z;

        declareIO(3,2);
    }
```

`declareTypedIO` is similar to `declareIO` but it allows you to specify the types of inlets or outlets you would like to be created. Details of usage rules for `declareTypedIO` can be found in the javadoc for `MaxObject`.

Handling Numeric Messages

Communication between objects in the Max application environment is accomplished by by the sending and receiving of messages. For instance, when you have one Max box **box1** that outputs an integer and that integer outlet is connected to the inlet of another Max box, **box2,** Max will send an "**int**" message with an integer argument value to **box2**. This diagram can be considered a representation of how an "int" message with a value of 22 flows through your patch.

[box1] ---->-----"int 22"--->----**[box2]**

In C externals specific functions are bound to specific messages via the addmess function call. In Java we have abstracted this paradigm a little further. For primitive messages (bang, int, float, and lists) `MaxObject` provides empty functions that you can override to implement the appropriate reaction to an incoming message. For example, if you would

like your Java external to respond to the “bang” message you would override the bang method of `MaxObject`.

```
public void bang()
{
   post("I was just banged pretty well.");
}
```

This would cause "I was banged pretty well." to be printed to the Max console when your external receives a bang in any of its inlets.

Using getInlet() to Determine which Inlet Received a Message

If you have multiple inlets, determining which inlet an incoming message was passed through is done by calling the `getInlet` function of `MaxObject`.

```
public void bang()
{
   int inlet_num = getInlet();
   post("I was just banged pretty well in inlet "+inlet_num);
}
```

`getInlet` will return the number of the inlet which last received a message. The inlets are indexed starting at the number 0 from the leftmost inlet. Thus if you received a bang in the first inlet this function would print "I was just banged pretty well on inlet 0", and if you received a bang on the second inlet it would print "I was just banged pretty well on inlet 1.”

Handling Int and Float Messages

Handling integer and float messages is very similar to handing the bang message except that you will also be receiving a value in addition to the message. When someone connects a number box to an inlet of your Java external and changes its value your object will recieve an “int” or “float” message from an integer or floating-point number box, repsectively. To handle these primitive numerical messages one needs to override the inlet methods of `MaxObject`. There is one function for the “int” message and one function for the “float” message.

```
public void inlet(int i)
{
   int intlet_no;
   inlet_no = getInlet();
   post("I got an integer in inlet "+inlet_no);
}

public void inlet(float f)
{
```

```
    int intlet_no;
    inlet_no = getInlet();
    post("I got a floating point number in inlet "+inlet_no);
}
```

Notice that both functions are named the same but have different argument types. This is known as **method overloading**. In Java a method's **signature** is made up of not only its name but also the number and types of its parameters.

NOTE: If you only override the float inlet function, `inlet(float f),` **mxj** will coerce int messages to float messages automatically for you with the accompanying int arg being coerced to a float typed arg of an equivalent value. The same holds true the other direction as well. If you only override the int inlet function, `inlet(int i),` **mxj** will coerce float messages to int messages automatically for you with the accompanying float arg being coerced to an int typed arg with the floating point part truncated. Thus, if you only want to ever deal with all primitive number messages, int or float, as one type you only have to override one of the inlet methods.

Handling Lists

In Max a combination of primitive numeric types and symbols that begins with a number is known as a list. If the first element is a symbol we call it a message rather than a list – more about this in the next section.

Lets say you make a message box in Max that looks like **[1.2 4 killa 8.6 bass 3 88 mahler]** and connect it to the inlet of another Max box, **box2**. Whenever you click on this message box the message that it actually sends to **box2** is actually a "list" message:

mouse click'-->[1.2 4 killa 8.6 bass 3 88 mahler]-->"list 1.2 4 killa 8.6 bass 3 88 mahler"-->[box2]

Since a list can contain integers, floating point numbers and symbols we need to introduce a new data type to handle these types of messages. This data type is known as an `Atom`. An `Atom` can represent either a floating point value, an integer value or a `String` value. This could be likened to a union in C. At their lowest messaging levels both **mxj** and Max actually deal with all message arguments as `Atom` datums. As you will see later when communicating data back to the Max application environment via outlet calls you will often times be working with the `Atom` class as opposed to primitive Java data types. This is because the Max application itself knows nothing about Java data types. Internal to Max there are actually only 3 data types. Integers, floating point numbers and symbols. The **mxj** bridge automatically handles the conversion for you to/from symbols and Java String objects as a convenience. (This might be a good time to take a quick look at the javadoc for the `com.cycling74.max.Atom` class. Cycling '74

provides a rich API for manipulating `Atom` data to make the programming less cumbersome.)

Handling the “list” message is much the same as handling the primitive numerical messages above with the exception that you are passed an array of `Atom` as opposed to a singleton argument of a known type.

```
public void list(Atom[] args)
{
   post("Received list message at inlet "+getInlet());
   //loop through all the atoms in the list
   Atom a;
   for(int i = 0; i < args.length; i++)
   {
      a = args[i];
      if(a.isFloat())
         post("List element "+i+" is a floating point atom with
              a value of "+a.getFloat());
      else if(a.isInt())
         post("List element "+i+" is an integer atom with a
               value of "+a.getInt());
      else if(a.isString())
         post("List element "+i+" is a String atom with a value
               of "+a.getString());
   }
}
```

The list method above shows an example of very basic list processing. It loops through all the `Atom` elements in the list and prints a message in the Max console window identifying the element's position in the list, its data type and its value. This if we sent our example message above, “1.2 4 killa 8.6 bass 3 88 mahler” to any inlet of our Java external the following would be printed in the Max console window:

List element 0 is a floating point atom with a value of 1.2

List element 1 is an integer atom with a value of 4

List element 2 is a String atom with a value of killa

List element 3 is a floating point atom with a value of 8.6

List element 4 is a String atom with a value of bass

List element 5 is an integer atom with a value of 3

List element 6 is an integer atom with a value of 88

List element 7 is a String atom with a value of mahler

As with any other message we call the function getInlet() to determine which inlet the list message was input.

Handling Arbitrary Messages

When writing Max externals in C you need to explicitly tell the Max environment which messages your external is capable of responding to by explicitly registering it with Max via the `addmess` function. When registering your message handler via `addmess` you also provide information about the quantity and data type of the arguments your function expects.

For Java externals there is no explicit mechanism to define and register the messages to which you wish your external to respond. Rather we use Java's notion of method visibility to identify methods that should be called in response to certain messages. The introspective mechanism that makes this possible is absent in C but you may be familiar with it if you have programmed in C++, Smalltalk or some other object oriented programming language.

The visibility of a Java method can be restricted by the Java language keywords `public`, `private` and `protected`. This visibility determines which other classes are allowed to call the method. When a method is declared `private` it is only ever allowed to be accessed by instances of the class in which the method is defined. When a method is declared `protected` it is accessible by instances of the defining class as well as subclasses of that class and any other classes defined in the same package. This is similar to the notion of friendliness in C++ where you say class A is a 'friend' of class B and is thus given access to data and functions in class B. Finally a method declared `public` is accessible by any class currently loaded in the JavaVM. In essence `public` means that the function or member variable is accessible to all and is in no way restricted. One more visibility specifier exists and it occurs when you define a function without using any of the aforementioned keywords (`public`, `private`, `protected`). A method or member variable declared without a visibility specifier is accessible only to classes within the same package as the defining class.

To expose a message that your Java external will respond to in the Max environment one should declare a function of the same name as the message you wish to handle with the `public` visibility modifier and a return type of `void`. For example, if you want to respond to a "stop" message you could declare the following function in your Max Java external.

```
public void stop()
{
   post("I received the stop message.");
}
```

Thus if someone typed stop in a message box and connected it to an inlet of the Java external the above function would be called whenever they clicked on the message box.

You may at this point wonder why the methods to respond to “int” and “float” message are named `inlet` as opposed to functions named `int` and `float`. The reason is that `int` and `float` happen to be reserved keywords in the Java language specification and cannot be used as function names.

Arbitrary Messages and Parameter Types

Many messages need to be accompanied by a data argument – for instance, “resize 1000”. The most general way to define a message that accepts an argument is to declare the method with an array of `Atom` as a parameter. This is much like the list message signature. Thus we could write an implementation of resize as follows:

```
public void resize(Atom[] newsize)
{
   Atom a;
   if(newsize.length >= 1)
   {
      a = newsize[0];
      if(a.isInt())
         _do_resize(a.getInt());
      else if(a.isFloat())
         _do_resize((int)a.getFloat());
      else
         post("resize message does not understand the string
               argument "+a.getString());
   }
}
```

If you think this seems cumbersome don't despair. There is mechanism in place to make the declaration of message handlers much more syntactically streamlined. You could also declare the resize method as:

```
public void resize(int newsize)
{
   _do_resize(newsize);
}
```

In the previous section, *Writing The Constructor*, we talked about how **mxj** coerces arguments to your constructor from any type to the declared type automagically. The same logic applies in the case of message handling as well. In this instance if someone sent your Java external the resize message with a floating point argument i.e. resize 200.4 the **mxj** bridge would convert the floating point value 200.4 to the integer value 200 before calling your resize message handler.

NOTE: If someone sent your Java external the resize message with a symbol argument, such as resize goforit, **mxj** would print a message to the Max console warning them of a potentially strange coercion and call the resize function with an integer argument of 0. In all cases where someone is passing a symbol to a function which is expecting numerical arguments this warning message is printed and the function will be passed 0 in lieu of the symbol.

In the case where you would like your message handler to accept more than one parameter just add more parameters to the message handler itself.

```
public void doit(int i1, int i2, float f1, float f2)
{
   post("doit was called with "+i1+":"
              +i2+":"
              +f1+":"
              +f2);
}
```

The following types of parameters are all supported in the coercion scheme used by the **mxj** bridge and are all valid as parameter types when declaring your message handling functions.

`boolean` **mxj** will coerce all non zero arguments to true and all zero valued arguments to false. All symbol arguments except the symbol “false” will be coerced to true.

`byte` **mxj** will coerce all numerical arguments to a Java byte datatype (8 bits signed) which has a range of -128 to 127. Symbol arguments are always coerced to 1 unless the symbol itself has the value ‘false’.

`char` **mxj** will coerce all numerical arguments to their corresponding Java unicode character (16 bits). If a symbol of length 1 is given it will be passed through as such otherwise the first letter of the symbol is used.

`short` **mxj** will coerce all numerical arguments to a Java short data type (16 bits) which has a range of -32768 to -32767. Symbol arguments are always coerced to 1 unless the symbol itself has the value ‘false’.

`int` **mxj** will coerce all numerical arguments to a Java int data type (32 bits) which has a range of -2147483648 to 2147483647. Symbol arguments are always coerced to 1 unless the symbol itself has the value ‘false’.

`long` **mxj** will coerce all numerical arguments to a Java long datatype (64 bits). Since internally the Max application uses 32 bit integers you will not be able to get any added precision by using a Java long datatype. However you may wish to use 64 bit integers in calculations internal to your external. Symbol arguments are always coerced to 1 unless the symbol itself has the value 'false'.

`float` **mxj** will coerce all numerical arguments to a 32 bit IEEE 754 floating point number with a value in the range of +/-1.4E-45 to +/-3.4028235E+38. Symbol arguments are always coerced to 1 unless the symbol itself has the value 'false'.

`double` **mxj** will coerce all numerical arguments to a 64 bit IEEE 754 floating point number. Since internally the Max application uses 32 bit floating point numbers you will not be able to input any added precision by using a Java double datatype. However you may wish to use 64 bit floating point numbers in calculations internal to your external. Symbol arguments are always coerced to 1 unless the symbol itself has the value 'false'.

`String` **mxj** will coerce all numerical arguments to their equivalent String representation. In this case a warning message will be printed to the Max console. Symbol arguments are converted to an equivalent Java String data type.

Primitive arrays are also supported as valid parameter types for your message handlers. In this case however, the array parameter must be the only parameter to your function. Thus:

```
public foo(int[] ia1, int[] ia2)
```

and

```
public foo(float f, int[] ia)
```

are currently not legal message handlers for a Max Java external while

```
public foo(boolean[] ba)
```

is a valid message handler. The following Java array types are supported in this single parameter context with the same coercion rules defined above being applied to each element in the array. When arbitrarily ordered mixed type message parameters are required the `Atom[]` data type is also supported as illustrated in the first example above.

```
boolean[]
byte[]
char[]
short[]
int[]
long[]
```

```
float[]
double[]
String[]
Atom[]
```

The 'anything' Message

One further part of the Java message framework which should be mentioned briefly is the 'anything' message. If you define a message with the following signature:

```
public void anything(String msg, Atom[] args)
```

whenever your external receives a message which cannot be resolved to any other message, the `anything` method will be called with the `msg` parameter set to the name of the unresolved message and the arguments passed in as the `args Atom` array. The `anything` method allows you to implement a catchall message handler, or create your own message dispatching framework. Consider the following example, which posts the contents of the input message to the Max window.

```
public void anything(String msg, Atom[] args)
{
   post("I received a " + msg + " message.");

   if (args.length > 0)
   {
      post("It has the following arguments:");
      for (int i=0;i<args.length;i++)
         post(args[i].toString());
   }
}
```

The viewsource Message

The `MaxObject` class that every Java external must subclass provides by default an implementation of a message called “viewsource”. Upon receipt of this message your Max Java external will attempt to find its corresponding Java source file and display it in an instance of the `MXJEditor`. You may be familiar with the `MXJEditor` from the quickie development environment. If the source file is unable to be found the default viewsource message handler will try to decompile the class file and display a representation of the source code in an `MXJEditor` instance. If you do not desire this particular behavior to be active by default for your Java external, you simply need to override the `viewsource` method of `MaxObject` with an empty method. Since the majority of Max Java externs do respond meaningfully to this message it can be a valuable tool to see how someone has implemented functionality in their Java external that you wish to mimic.

Message Overloading

While the method overloading features of the Java language provide a nice framework for specifying the the construction of an instance of your Java external it is not suggested that method overloading be used when writing the functions that will respond to Max messages. This primarily because we are 'superloading' the Max data types when we provide the coercive behavior of **mxj**. Max has only 3 primitive data types while Java has 9 primitive data types.

The decision to support all of the primitive Java data types internally in the **mxj** bridge and the Max Java API was made so that coding Max Java externs felt more or less like writing a standard Java class, and so that porting existing Java code to **mxj** is easy. However, in this instance method overloading presents a small quandry. Consider the following method signatures:

```
public void doit(int i);
public void doit(short s);
public void doit(long l);
```

Internally Max only has one integer data type. With this in mind, when you send the message [doit 24] from within Max patcher land **mxj** has no sound basis from which to decide which of these 3 methods should be called. In this case it is undefined which method would actually be called.

Although it is valid and may seem useful to overload in a scheme such as

```
public void doit(int i);
public void doit(float f);
```

It is probably just easier to declare

```
public void doit(float f);
```

and let the built-in coercion of **mxj** handle coercing any integer arguments that may be passed to the doit function into floating point numbers.

Despite the fact that we do not recommend the practice, we have implemented some ordered rules that apply to method overloading:

1. Methods are first matched by most inclusive parameter type.

```
public void doit(Atom[] args);
public void doit(int[] args);
public void doit(int x);
```

Given the previous three methods the method `doit(Atom[] args)` will always be called no matter the number or types of arguments.

```
public void doit(int[] args);
public void doit(int x);
```

Given the previous two methods the method `doit(int[] args)` will always be called no matter the number of arguments.

2. Methods are matched by number of parameters.

```
public void doit(int x, int y);
public void doit(int x);
```

Given the previous 2 methods. If you send the doit message with a number of 2 or greater arguments doit(int x, int y) will be called. If you send the doit message with 0 or 1 arguments doit(int x) will be called.

Given the 'superloading' of the 3 Max primitive types, **mxj**'s built-in coercion scheme and the above rules you can see how it could potentially get confusing to keep a grip on which version of your method would be called under which circumstances. This is why we suggest that you simply make another method as opposed to overloading an existing method.

Interacting With Max

Communication is a two-way street. We have spent a lot of time up until this point describing how to set up your class to receive communication from the Max environment. This section explains how to output data and do other useful things.

Outlet Functions

If you have a look at the javadoc for `MaxObject` you will see that there are numerous methods you can use to output messages to the Max environment. Let's take another very simple example for starters. Say that for every bang message your external receives you wish to outlet 2 bang messages out of its first outlet:

```
public void bang()
{
   outletBang(0);
   outletBang(0);
}
```

The first and only argument to the `outletBang` call is the index of the outlet you wish the bang message to be sent out. Outlets are indexed left to right starting at 0. Thus if you had created 3 outlets in your constructor like so:

```
declareOutlets(new int[]{ DataTypes.INT, DataTypes.FLOAT,
DataTypes.ALL});
```

The leftmost `INT` outlet would be indexed as 0, the `FLOAT` outlet would be indexed as 1 and the `ALL` outlet would have an index of 2. With this in mind the rest of the outlet functions defined in `MaxObject` (detailed in the javadocs) should make sense. You can use the `outlet(int index, Atom[] args)` function to outlet any arbitrary message you wish to the Max environment. The Atom API (detailed in the javadocs) provides functions for constructing `Atom` objects from any primative Java data type. Thus if you wished to output a list of 2 integers and a floating point number out of the 2nd outlet in response to a bang message the method could look like this.

```
public void bang()
{
   int a = 42;
   int b = 38;
   float c = 0.005F;
   outlet(1,new Atom[]{Atom.newAtom(a),Atom.newAtom(b),
            Atom.newAtom(c)});
}
```

To make the process of outletting messages from your external less syntactically cumbersome there are a variety of convenience outlet functions defined in `MaxObject`. These cover the simplest cases such as when you wish to output a single `int` or `String`. There are convenience functions defined for every primitive Java data type in this singleton value situation. For instance, to outlet a single integer you can simply define your function as follows.

```
public void bang()
{
   outlet(0,42);
}
```

as an alternative to:

```
public void bang()
{
   outlet(0,new Atom[]{Atom.newAtom(42)});
}
```

or

```
public void bang()
{
```

```
    outlet(0,Atom.newAtom(42));
}
```

which are also both valid calls but are syntactically less concise. The same holds for all of the primitive Java types supported by **mxj**.

```
public void outlet(int index, boolean b);
public void outlet(int index, char c);
public void outlet(int index, short s);
public void outlet(int index, String msg);
public void outlet(int index, int[] i_array);
public void outlet(int index, String[] str_array);
```

etc.

Currently, there are no convenience functions for the situation when you wish to output a message from your Java external that has more than one argument. If you wish to output something along the lines of [setdim 64 32] from your Java external, you must use the the `outlet(int idx,String msg,Atom[] args)` function.

```
public void bang()
{
    outlet(0,"setdim", new Atom[]{ Atom.newAtom(64),
          Atom.newAtom(32)});
}
```

It is also worthwhile to note that the following call is equivalent as well.

```
public void bang()
{
    outlet(0, new Atom[]{Atom.newAtom("setdim"),
          Atom.newAtom(64), Atom.newAtom(32)});
}
```

If the first argument in your message is a primitive numerical type it will automatically be considered a 'list' message by Max. Thus the following two calls are also equivalent.

```
public void firstway()
{
    outlet(0,new Atom[]{Atom.newAtom(11.5), Atom.newAtom(23))
}

public void secondway()
{
    outlet(0,"list", new Atom[]{Atom.newAtom(11.5),
                                Atom.newAtom(23)});
}
```

Sending Messages Out Of The InfoOutlet

There are currently no functions explicitly defined in the `MaxObject` API to output messages directly from the info outlet. If you recall, the info outlet is normally created by default for your Java extern. If you wish to access this outlet you can use the `getInfoIdx` function to determine its index and use any of the outlet functions as you normally would. For instance you may wish to respond to a message 'dump' which dumps the current state of your Java external for debugging purposes. It would make sense that this sort of message may respond by dumping the state information to the info outlet.

```
public void dump()
{
   //_frequency, _load, and _stepsize are private member
   // variables of this class
   int info_idx = getInfoIdx();
   outlet(info_idx, new Atom[]{Atom.newAtom("frequency"),
                               Atom.newAtom(_frequency));
   outlet(info_idx, new Atom[]{Atom.newAtom("load"),
                               Atom.newAtom(_load)});
   outlet(info_idx, new Atom[]{Atom.newAtom("stepsize"),
                               Atom.newAtom(_stepsize)});
}
```

Using Attributes

Attributes are another feature of the Max Java API that provide extended functionality with little additional work for the Java external developer. Those who are familiar with Cycling 74's jitter extensions for video and matrix processing may already be familiar with the attribute paradigm.

An attribute can be considered a property of your Java extern. Attributes may be read only or mutable depending on how you define them in your class. When something is declared as an attribute in your Java external, you do not need to define messages to get or set its value. These are provided by default by the **mxj** bridge. The functions allowing you to declare something as an attribute are inherited from `com.cycling74.max.MaxObject`.

Consider as an example a Java external that outputs a certain number of bangs in response to an incoming bang:

```
import com.cycling74.max.*;
public class banger extends MaxObject
{
   private int repeat; //how many bangs to output
                       //in response to each bang received

   public banger()
```

```
    {
       repeat = 2; //default to 2 bangs for every bang received
       declareIO(1,1); //one inlet, one outlet
       declareAttribute("repeat"); //make repeat an attribute
    }
    public void bang()
    {
       for(int i = 0; i < repeat; i++)
          outletBang(0);
    }
}
```

Declaring the `repeat` member variable as an attribute causes this Java external to respond by default to the messages "repeat *int*" and "getrepeat". If you wish to change the number of bangs the external will output to 10 you can send the external the message "repeat 10". Correspondingly you can query the current `repeat` value by sending the message "getrepeat". If the default info outlet exists **mxj** will outlet the current value of `repeat` out of the info outlet. The value will be prefixed by the attribute name. If the default outlet does not exist because you called `createInfoOutlet(false)` in your constructor the attribute name and value will be printed to the Max console. This is equivalent to the developer defining the following methods in their Max Java class if the `repeat` member variable was not declared as an attribute:

```
public void repeat(int val)
{
   repeat = val;
}
public void getrepeat()
{
   int info_idx = getInfoIdx();
   if(info_idx == -1)
      post("repeat "+repeat);
   else
      outlet(info_idx,repeat);
}
```

Another nice benefit of attributes is that the developer does not have to handle the optional setting of initial values in the constructor. The value of an attribute can be set via the '@' syntax in a new object box. To set the initial value of repeat in a banger instance the user could create **mxj banger @repeat 33** and the value of repeat in the new instance would be initialized to 33. At any time in the future the user can still send the "repeat *int"* message to the **mxj banger** box to change the value of repeat.

All the primitive Java data types supported by **mxj** are able to be exposed as attributes. These are

```
boolean, byte, char, short, int, long, float, double,
String,boolean[], byte[], char[], short[], int[], long[],
float[], double[], String[]
```

When an attribute is of an array type the behavior is exactly the same except that you can send multiple values corresponding to the elements in the array in the attribute set message. Conversely you will receive a list of values as opposed to a single value out of the info outlet when you send the get attribute message. If you declared a member variable named `frequencies` of type `int[]` as an attribute you could set the current value of the `frequencies` array by sending the message "frequencies 1 33 4 55 66 77 2 1 66 78 99" to your Java extern.

Special Handling of Attributes

Sometimes it is desirable to override how an attribute is get or set. This may happen if you wish to limit the range of an attribute or present a different representation of an attribute to the Max user that differs from how the attribute is defined internally. To accomplish this there are additional attribute declaration functions that allow you to specify a different getter and/or setter method than the default getter/setter provided by the **mxj** bridge.

Imagine that you have a class that has a member variable which represents a value in radians, but you would rather the Max user be able to deal with the value in degrees. You might accomplish this with the following code:

```
import com.cycling74.max.*;
public class foo extends MaxObject
{
   private static final double PI_OVER_180 = Math.PI / 180.0;
   private static final double 180_OVER_PI = 180.0 / Math.PI;

   private double angle;//value of angle in radians
   public foo()
   {
      angle = 0.0;
      declareIO(1,1);
      declareAttribute("angle","getangle","setangle");
   }
   private void setangle(double angle)
   {
      angle = angle * PI_OVER_180; //convert degrees to radians
   }
   private double getangle()
   {
      return angle * 180_OVER_PI; //convert radians to degrees
   }
}
```

You could also imagine a situation where you have a member variable that is not of a primitive type supported by the **mxj** bridge but you would like to expose it as an attribute anyway. For example, `java.awt.Point` is a class that stores the x and y-coordinate of a two-dimensional point in the Cartesian plane.

```
import com.cycling74.max.*;
import java.awt.Point;
public class foo extends MaxObject
{
   private Point coord;
   public foo()
   {
      coord = new Point(0,0);
      declareIO(1,1);
      declareAttribute("coord","getcoord","setcoord");
   }
   private void setcoord(int x, int y)
   {
      coord.x = x;
      coord.y = y;
   }
   private int[] coord_ret = new int[2];
   private int[] getcoord()
   {
      coord_ret[0] = coord.x;
      coord_ret[1] = coord.y;
      return coord_ret;
   }
}
```

Notice that in both cases above the getter and setter methods are declared as private. This is because if they were declared as public they would be registered as message handlers by **mxj** which in this case is not what we want. You may be wondering why you'd want to declare a member variable as an attribute when you lose the benefits of built-in getters and setters. One reason is that attributes enjoy the ability to use the @ object-instantiation syntax. Also, in the future **mxj** will support integration with the **pattr** state-management objects introduced in Max 4.5, so by declaring appropriate member variables as attributes you will automatically gain the additional benefits of that integration.

It is also worthwhile to note that you can specify solely a custom getter OR a custom setter - you do not have to specify both. To do this just pass `null` as the argument for the getter or setter you wish to remain the default in your call to `declareAttribute`.

```
declareAttribute("foo",null,"setfoo");//default getter,
                                      //custom setter,
declareAttribute("foo","getfoo",null);//custom getter, default
                                      //setter
```

All the methods which are used to define various kinds of attributes are documented in more detail in the javadoc for `MaxObject` including methods for declaring attributes as read only.

There you have it. At this point you should have a basic understanding of how to shuttle data to and from Java classes you create and the Max application environment, which is enough to make a fully functioning Java external. In the next sections we'll concentrate on aspects of the Max Java API with more advanced capabilities.

Scheduler, Clocks, and Executables, Oh My!

It is not within the scope of this document to provide an in depth explanation of the event scheduling mechanisms present within the Max application or a general discussion of threading in modern operating systems. For a nice overview of these mechanisms you may wish to have a look at the article, Event Priority in Max (Scheduler vs. Queue) which is provided as an Appendix to this document.

By default all event processing within Max occurs in the main thread of the Max application. Events are generated by such things as MIDI input/output, mouse clicks, and time-dependent externals, and every event takes some amount of time to process. The amount of time an event takes to process depends on what needs to happen computationally to successfully complete the processing of that event. For instance if someone sends your Java external a bang message and you simply output an integer or do some simple calculation in response, the time it takes to complete the processing of this `bang` method is extremely short. On the other hand, if someone sends your Java external the bang message and your `bang` method does some intensive weather simulations it could take quite some time. During the time that it takes to calculate your weather simulation the Max application is unable to process any other events since there is only one thread, the main application thread, processing all events. This could potentially result in a non-responsive user interface in your Max patch or even worse the dreaded spinning wheel or wristwatch until the computations necessary to complete the processing of the event are complete. This is because in addition to processing patch events the main application thread handles all the tasks that make the application interactive: redrawing the user interface and polling the underlying event manager for mouse clicks. If you have a message handling function which takes a significant amount of time to complete, all these important services will be blocked until the execution of your message handler completes.

When the Max user has the Overdrive option enabled there is another thread created within the Max application which is dedicated solely to processing events. This enables events scheduled for processing by this thread to be serviced while to Max application continues to be responsive. This thread is known as the scheduler or high-priority thread since events scheduled for processing by this thread are given a higher priority than

events scheduled for processing in the main thread. Correspondingly the main Max application thread is known as the low-priority thread since events processed by this thread are not given as high a priority as those scheduled for processing by the high-priority thread.

The fact that scheduler events are given higher priority allows a faster response to be achieved since the scheduler thread more frequently processes events and is not laden with any additional user interface tasks. It is important to note that like the main thread, the scheduler thread can only process one event at a time, so if a particular event takes a relatively long time to process, all the other events in the scheduler thread will be delayed. Therefore a properly written Java external will not use the scheduler for functions which may take a long time to execute. Expensive functions should either be calculated in a separate Java thread or deferred to the low priority main thread using a method from `MaxSystem`. In addition, one may use a `MaxQelem` object to ensure that no more than one of these events will be in the event queue at any given moment. This is explained in further detail below.

The Max Java API provides many functions to give the Java external developer control over event processing. These are documented in the `MaxObject`, `MaxSystem`, `MaxQelem`, and `MaxClock` APIs.

Outputting Data at High Priority

If your Java class has a method which responds to an incoming message by outputting a message, the message will be output in the same thread as the incoming message. For instance, because the bang messages from a **metro** object are output in the scheduler thread, a `bang` method in your Java class responding to the **metro**'s input will be called in the scheduler thread. User interface events, such as a click on a message box, are handled in the low priority thread.

It is also possible to schedule events to be processed by the high-priority scheduler thread from within a different thread. If for reasons of timing you feel that you need a message to be handled by the Max application as soon as possible, the `outletHigh` function call can result in your message being processed by Max sooner due to the finer timing granularity of the high-priority scheduler thread. Note that the scheduler thread is only active if the end user has overdrive enabled. Otherwise this call is more or less equivalent to the normal `outlet` method.

Directly Scheduling High Priority Events with the Executable Interface

Another way to have messages processed by the high-priority thread is to use the `schedule` or `scheduleDelay` calls in the `MaxSystem` API. These calls take an

instance of an object implementing the Max Java `com.cycling74.max.Executable` interface. An `Executable` object must implement an `execute` method. The following code shows how you might use the high-priority scheduler thread to delay the output of a bang message by some arbitrary time upon receipt of a bang message.

```
public void bang()
{
   //create a new instance of executable
   Executable e = new Executable()
   {
      public void execute()
      {
         outletBang(0); //send a bang out the first outlet
      }
   }
   scheduleDelay(e, 2000);     //call the execute function of e
                               // 2 seconds (2000 ms) from now.
}
```

Another strategy would be to reuse the same instance of `Executable` instead of creating a new one every time, thus reducing the overhead of Java object creation. In this instance `e` would be declared as a member variable of your class.

```
private Executable e = new Executable()
{
   public void execute()
   {
      outletBang(0); //send a bang out the first outlet
   }
};
public void bang()
{
   scheduleDelay(e, 2000);     //call the execute function of e
                               //2 seconds (2000 ms) from now.
}
```

Using Clocks and the Callback Object

If you have a time sensitive task that needs to be repeated at a regular interval the `MaxClock` API provides a mechanism to accomplish this. Once again you need to wrap your function call in an instance of the `Executable` interface. Here is a simple implementation of a metronome which outputs a bang message every 500 ms upon receipt of the 'start' message and stops outputting bang messages upon receipt of the 'stop' message.

```
importcom.cycling74.max.*;
public class simplemetro extends MaxObject
{
   MaxClock c;
```

```
    public simplemetro()
    {
       declareIO(1,1); //one inlet, one outlet

       //create instance of MaxClock that will call the
       //function dobang everytime it "ticks"

       c = new MaxClock(new Callback(this, "dobang"));
    }
    private void dobang()
    {
       outletBang(0);//outlet a bang out of the first outlet
       c.delay(500.0);//set the clock to execute again in 500 ms
    }

    public void start()
    {
       c.delay(0.0); //set the clock to execute immediately
    }
    public void stop()
    {
       c.unset(); //stop the clock from executing
    }

    //notifyDeleted is called by the Max application
    //when the user deletes your external from a Max patch
    //or closes a Max patch of which your Java extern
    //is a member.

    public void notifyDeleted()
    {
       c.release();  //release native resources which
                     //Max associates with the MaxClock object
                     //this is extremely important. Otherwise
                     //these resources would be unable to be
                     //reclaimed
    }
}
```

You may notice that in this example we introduce the `Callback` object as an alternative to using a raw instance of `Executable` to specify which function the clock should execute when it "ticks". The `Callback` class takes care of implementing the `Executable` interface for you and is often syntactically cleaner than creating an instance of `Executable` yourself.

The first argument to the `Callback` constructor is the instance of the object that contains the function you wish to execute and the second argument is the name of the function itself. If you wished to not use the `Callback` object the following code is functionally equivalent when creating your instance of `MaxClock`.

```
public simplemetro()
```

```
{
   declareIO(1,1); //one inlet, one outlet
   c = new MaxClock(new Executable(){
      public void execute()
      {
         outletBang(0);
         c.delay(500.0);
      }
   });
}
```

Using the `Callback` object potentially improves the readability of your code. You can use an instance of the `Callback` object in any situation where a Max Java API function requires an instance of `Executable` since the `Callback` object itself implements the `Executable` interface. There are other situations where the `Callback` object is also useful, such as the ability to provide arguments to the function you wish to execute when it is called. For more details, see the javadocs for the `Callback` object.

It is also worthwhile to note that `MaxClock` provides a convenience constructor in which you can directly specify the object and the function you wish to execute. Thus the following `MaxClock` construction code is also equivalent to the previous 2 examples.

```
MaxClock c;
public simplemetro()
{
   declareIO(1,1); //one inlet, one outlet

   //create instance of MaxClock that will call
   //the function dobang everytime it "ticks"

   c = new MaxClock(this, "dobang");
}
```

Qelems and Throttling Time-Consuming Events

The `MaxQelem` object provides a mechanism by which you can defer execution of an event to the main thread. This can be useful in the context of an expensive process which will be triggered from the scheduler thread. Rather than tie up the scheduler and delay time-sensitive tasks, it is often preferable to transfer the processing to the main thread, delaying instead the user interface and other events with more relaxed expectations. One can also use `MaxQelem` to ensure that only one instance of an event will ever be in the main queue at any given moment.

For instance, imagine that in response to an input parameter the Java external will populate an array with several million elements. Since this could take some time it makes sense to defer this method's execution to the main thread as follows:

```
import com.cycling74.max.*;
public class qelemtest extends MaxObject
{
   private MaxQelem q;
   private double[] darray;
   private double someParameter;

   public qelemtest()
   {
      declareIO(1,1);
      q = new MaxQelem(new Callback(this,"fillarray"));
   }

   private void fillarray()
   {
      double thisParameter = someParameter;
      darray = new double[10000000];//10 million element array
      for(int i = 0; i < darray.length;i++)
      {
         darray[i] = Math.sin(System.currentTimeMillis()
                        + thisParameter);
      }
   }

   public void inlet(double d)
   {
      someParameter = d;
      q.set();
               //schedule the function fillarray to
               // be executed by the low priority thread.
               //if it is already set do not schedule it
               //again. The fillarry function was specified
               // as the qelem  function when we created the
               // MaxQelem q in the construtctor
   }

   public void notifyDeleted()
   {
      q.release();  //release the native resources associated
                    //with the MaxQuelem object by the Max
                    //application. This is very important!!
   }
}
```

So when the user sends a number in the inlet, execution of the `fillarray` method is deferred to the main thread. Furthermore, if more numbers are input while the `fillarray` method is active, the `MaxQelem q` can only be set once, and the most recent value for `someParameter` will be used when the `fillarray` method is finally called.

Using Java Threads for Computationally Intensive Tasks

It is also worth noting that you can leverage the built-in threading capabilities of the Java language to similarly offload some of the overhead of computationally expensive event processing from the Max application to the Java VM. So rather than use the scheduler or the main thread, this option allows you to create an entirely new thread to handle your task.

Here we present another simple example in which we fill a large array with values and send a bang message out of the info outlet when the event processing is complete. Note that this does not provide the same throttling behavior as the `MaxQelem` mechanism but it does allow us to continue processing events in the Max application after receipt of a bang message while filling the array in a separate thread.

```
import com.cycling74.max.*;
public class threadtest extends MaxObject
{
   private double[] darray;
   public threadtest()
   {
      declareIO(1,1);
      darray = new double[1000000];
   }
   public void bang()
   {
      //create a new thread to fill out darray
      Thread t = new Thread(){
         public void run()
         {
            for(int i = 0; i < darray.length;i++)
            {
               darray[i] = Math.sin(System.currentTimeMillis());
            }
            outletBang(getInfoIdx());   //outlet a bang out
                                        //of the info outlet
                                        //when we are through
                                        //filling the array.
         }
      };

      t.start(); //start the thread executing
   }
}
```

It is worth noting that when using Java threads inside of your Max external, any outlet calls you make back into the Max application will automatically be deferred for handling by the low-priority main thread for normal outlet calls and the high-priority scheduler thread for `outletHigh` calls. This is because within the Max application the main thread and the scheduler thread are the only valid threads that are allowed to make these calls.

An arbitrary Java thread is unknown to the Max application. Thus, if you are using Java threads to output data which is time-critical you may wish to use the `outletHigh` calls or schedule your events directly via the `schedule` or `scheduleDelay` functions of `MaxSystem`.

Scripting Max with Java

Many of the scripting capabilities introduced by the **js** JavaScript external in Max 4.5 are also exposed to the Java external developer. These functions are described in detail in the javadoc for the `MaxWindow`, `MaxPatcher` and `MaxBox` classes. The documentation for the **js** object distributed with Max 4.5 can also be a useful resource as well since the JavaScript and Java APIs are very similar in this regard.

Writing Signal Processing Objects In Java Using mxj~

With the advent of Just In Time compilation technology present in most modern Java virtual machines, extremely computationally intensive tasks can now be performed in Java with acceptable performance. In the Max application environment one such task in which we leverage this aspect of modern JVMs is to allow developers to write real time digital signal processing code entirely in Java using the **mxj~** Max/Java bridge.

This is not to say that Java is the best choice in terms of performance for your signal processing object. If obtaining the highest possible performance is a critical requirement for your project you probably will want to write the external in C. A properly written Java signal processing external is typically about 1.5-2.5X slower than the equivalent external written in C. A lot of this depends on what sort of processing your external is doing. In most cases the more computationally expensive your external, the closer that the Java and C versions will be in performance. This is because the biggest overhead introduced by doing your signal processing in Java is in the context switching that occurs when the **mxj~** bridge calls from C into your Java signal processing routine. The cost of this context switch is amortized across the amount of time you spend actually accomplishing your task, so the more expensive your task, the less relatively expensive is the cost of the context switch.

All of the features exposed by the Max Java API described previously in this document such as message handling, event scheduling etc, are also available for use in your signal processing external and are accessed and/or defined in exactly the same way as they would be in the development of a normal Max Java external. So writing signal processing externals in Java using **mxj~** is very similar to writing a normal max external in Java with the exception of some extra methods you must define and a slightly different setup. This is all described below.

Using mxj Quickie to Develop Java Signal Processing Externs

This section assumes you have read the section of this document entitled ‘mxj quickie—Integrated Java external Development from Within Max’ as we will merely extend upon the information presented there.

By default the **mxj quickie** integrated development environment presents you with a template for developing a normal Max external. If you take a look at the quickie help patch you may notice that there is an optional 3rd argument which allows you to specify which template you would like to use for the current development session. By default these templates are located in the *quickie-templates* directory in your *c74:java* folder but can be located anywhere in the Max search path. If you send quickie the message “listtemplates” it will print the names of the currently available code templates in the *quickie-templates* directory to the console.

When writing a Java signal processing external the simplest template to use is called MSPOBJ_PROTO. Thus to develop a new signal processing external called **mysigextern** you would type **mxj quickie mysigextern MSPOBJ_PROTO** into a new Max box. Double clicking this box will open up an `MXJEditor` instance filled out with the MSPOBJ_PROTO template.

Unlike the default template for normal max Java externals which is just a skeleton, the MSPOBJ_PROTO template contains the code for a fully functional simple gain object. This is primarily for mnemonic purposes. If you wish the template to look differently you can always open up the template itself in the quickie-templates directory and edit it to your liking or add an additional template of your own.

The compilation process is exactly the same as when using quickie to develop a normal Max external and the resulting class is placed in the same default location, your *c74:java/classes* directory.

When creating an instance of your Java signal processing external in a Max patcher you must use the **mxj~** external to load your class instead of **mxj**.

Writing the Constructor

There is very little difference when writing the constructor for your Java signal processing external when compared with writing the constructor for a normal Java Max external. Instead of inheriting from `MaxObject`, located in the `com.cycing74.max` package of the Max Java API, you will need to inherit from `MSPPerformer` which lives in the `com.cycling74.msp` package. Thus you will want to add an additional import to the top of your Java signal processing external’s source file so that you can refer to the `MSPPerformer` class in your class declaration:

```
import com.cycling74.max.*;
import com.cycling74.msp.*;

public class mysigobj extends MSPPerformer
{
```

When extending `MSPPerformer` you are given access to an additional type of inlet/outlet called '`SIGNAL`' which you can use to declare an inlet/outlet for which it is valid for an MSP signal to be connected in a Max patcher.

Inlet or outlets with type '`SIGNAL`' will always be created as the leftmost outlets no matter where you declare them. Thus it is good practice to make sure that you declare your '`SIGNAL`' inlets or outlets as the first ones in your signal declaration array. So if you had an external in which you wanted two signal inlets and one message inlet and one signal outlet and one message outlet you would declare them as follows in your constructor.

```
declareInlets(new int[]{SIGNAL,SIGNAL,DataTypes.ALL});
declareOutlets(new int[]{SIGNAL,DataTypes.ALL});
```

Since `MSPPerformer` actually is a subclass of `MaxObject` everything else with regards to the constructor, declaring attributes etc., is exactly the same.

NOTE: Due to the architecture of **mxj~** and MSP it is impossible to have a Java signal processing external with no inlets defined at all. Even if you try and declare your inlet specification as `NO_INLETS` a default message inlet will always be created.

dspsetup and the perform method

There are two other methods your **mxj~** external must implement. These are the `perform` method and the `dspsetup` method.

dspsetup()

The `dspsetup` method is called by the MSP signal compiler when your signal processing extern is being added to the dsp chain of a Max/MSP patcher. This occurs when the **dac~** is turned on or sent the "startwindow" message, when a user adds a new MSP object to the Max patcher, or when someone modifies the dsp chain of the patcher in some other way such as deleting a signal connection or making a new signal connection from/between MSP objects in a patcher. Since this method is called only once before the current dsp chain is run it provides an ideal opportunity to make calculations that will be static over the life of your Java external in the current signal processing context.

When the dspsetup method is called your external will be informed of the current sampling rate and vector size of the current dsp context as well as how many signal connections your external has at each of its signal inlets and outlets. For instance, many dsp algorithms require the inverse of the sampling rate as part of the calculation. Since the sampling rate will not change between calls to dspsetup it makes sense to calculate it once and cache the result for use in the workhorse perform method. The dspsetup method is defined as follows:

```
public void dspsetup(MSPSignal[] in, MSPSignal[] out)
{

}
```

The dspsetup method is passed two arrays of MSPSignal objects which correspond to the SIGNAL inlets and outlets you declared in your constructor. Thus if you declared two signal inlets in your constructor and three signal outlets the MSPSignal[] in parameter would have two elements and the MSPSignal[] out parameter would have three elements corresponding to the signal inlets/outlets you declared in left to right order.

When dspsetup is called the MSPSignal instances provide information about the current dsp context as well as some meta information regarding whether or not there is a signal connection at each signal inlet and outlet in the patcher. The member variables of MSPSignal are defined as follows:

```
public class MSPSignal
{
   public final float[] vec;   //sample vector.
   public double sr;           //the current sampling rate
   public int n;               //the current signal vector size
   public short cc;            //connection count
   public boolean connected;   //are any signals connected
                               //to this inlet/outlet
}
```

So if you wanted to cache the inverse sampling rate for later use, you could use the following dspsetup method (where inv_sr is a member variable of the class):

```
public void dspsetup(MSPSignal[] in, MSPSignal[] out)
{
   double sr = in[0].sr;
   inv_sr = 1.0/sr;
}
```

perform()

The perform method is the workhorse of your signal processing external. After the MSP signal compiler builds the signal processing chain for the current Max patcher it

repeatedly calls the `perform` method of all the dsp externals to process samples. Samples are passed into your external's `perform` method as vectors. In other words your external is given arrays of samples to process as opposed to one sample at a time. It is generally more efficient to process chunks of data at a time as opposed to passing one element at a time through the whole signal processing chain. Samples in MSP are 32-bit floating point values. Thus the signal vector is represented as an array of type Java `float[].`

The size of the signal vector is determined by the user via the DSP Status inspector accessed through the Options menu of the Max application. The size of the vector is always a power of two. Generally a smaller signal vector size results in increased computational overhead for a Max patcher because of the chunking mentioned above. A smaller signal vector means that smaller chunks of samples are passed through the dsp chain per chain computation resulting in increased function overhead.

The perform method is defined as follows:

```
public void perform(MSPSignal[] in, MSPSignal[] out)
{

}
```

As in the `dspsetup` method your external is passed an array of `MSPSignal` objects corresponding to the `SIGNAL` inlets and outlets you declared in your constructor in right to left order. Thus if you had the following in your constructor:

```
declareInlets(new int[]{SIGNAL,SIGNAL});
declareOutlets(new int[]{SIGNAL});
```

The `MSPSignal` object corresponding to the first signal inlet of your external would be the 0th element in the `MSPSignal[] in` array and the second signal would be the 1st element of the `MSPSignal[] in` array. The `MSPSignal` object corresponding to the only outlet would be likewise be accessed as the 0th element of the `MSPSignal[] out` array.

For example, say you wanted your perform routine to multiply these two signal inputs together and send the resulting signal out of the sole outlet. Your perform method in this case would look like:

```
public void perform(MSPSignal[] in, MSPSignal[] out)
{
   float[] in1  = in[0].vec;
   float[] in2  = in[1].vec;
   float[] out1 = out[0].vec;
   int vec_size = in[0].n;

   for(int i = 0; i < n;i++)
      out[i] = in1[i] * in2[i];
```

```
}
```

This is in essence the perform routine for a simple ring modulator. As you may notice the vector of sample data which you are currently responsible for processing is contained in a public member variable of the `MSPSignal` object known as `vec`. It is simply a Java floating point array populated with the current sample vector.

Note that you do not have to explicitly output the sample data out of an outlet as you might when outputting a message from a normal Max object with an `outlet` call. This behavior is implicit to the MSP paradigm. When you fill the sample vector of an `MSPSignal` object corresponding to one of your outlets these values are passed on to the next object connected to that outlet when your `perform` routine exits.

Complete Example

It seems warranted at this point to provide a complete example of a simple Java signal processing external. This simple external will add two signals together and output the result. If only one signal is connected it will use the last received floating point value as the other addend.

```
import com.cycling74.max.*;
import com.cycling74.msp.*;

public class sigadd extends MSPPerformer
{
   private float addend;

   public sigadd()
   {
      this(0);
   }
   public sigadd(float f)
   {
      addend = f;
      declareInlets(new int[]{SIGNAL,SIGNAL});
      declareOutlets(new int[]{SIGNAL});

   }

   public void inlet(float f)
   {
      addend = f;
   }

   public void dspsetup(MSPSignal[] in, MSPSignal[] out)
   {
      post("dspsetup was called.");
   }
```

```
    public void perform(MSPSignal[] in, MSPSignal[] out)
    {
       float[] in1 = in[0].vec;
       float[] in2 = in[1].vec;
       float[] o   = out[0].vec;
       int vec_size = in[0].n;
       int i;

       if(in[0].connected && !in[1].connected)
       {
          //2nd signal inlet is not connected
          for(i = 0;i < vec_size;i++)
                o[i] = in1[i] + addend;

       }
       else if( !in[0].connected && in[1].connected)
       {
          //1st signal inlet is not connected
          for(i = 0;i < vec_size;i++)
                o[i] = in2[i] + addend;
       }
       else if( in[0].connected && in[1].connected)
       {
          //both signal inlets are connected
          for(i = 0;i < vec_size;i++)
                o[i] = in1[i] + in2[i];
       }
       else
       {
             //neither signal inlet is connected
             for(i = 0;i < vec_size;i++)
                o[i] = addend;
       }
    }
}
```

Composition of Multiple MSPPerformers

Extending the `MSPPerformer` class is not the only option when developing signal processing externals in Java but it does have advantages. The most obvious advantage is that it is extremely straightforward to develop your extern. All you need to do is provide implementations of the `perform` and `dspsetup` methods. In many simple cases you may not even have to implement a `dspsetup` method if you have no expensive calculations that you wish to calculate once per dsp context.

The other advantage may not be so obvious. `MSPPerformer` actually implements the `MSPPerformable` interface. This means that every class that extends `MSPPerformer` is guaranteed to have an implementation of the perform method which will presumably

handle its signal processing duties and the `dspsetup` method which will take care of initializing the object for signal processing.

The fact that all `MSPPerformer` subclasses conform to this `MSPPerformable` interface makes it very easy to leverage existing `MSPPerformer` derived classes in your Java signal processing extern. This is probably most easily demonstrated with another simple example. Say you have the classes `myFilter` and `myNoise` which are both subclasses of `MSPPerformer`. You may hook them together within another `MSPPerformer` like so:

```
import com.cycling74.max.*;
import com.cycling74.msp.*;

public class comptest extends MSPPerformer
{

    myNoise noise;
    myFilter filter;
    MSPSignal[] tmp;

    public comptest(Atom[] args)
    {
        declareInlets(new int[]{DataTypes.ALL});
        declareOutlets(new int[]{SIGNAL});
        noise = new myNoise();
        filter = new myFilter();
        tmp = new MSPSignal[1];
    }

    public void cutoff(float cf)
    {
        filter.cutoff(cf);
    }

    public void dspsetup(MSPSignal[] in, MSPSignal[] out)
    {
         //dupclean..
         //create an identical clone of the MSPSiganl out[0]
         //with its own zeroed out sample vector. This is the
         //safest way of composing multiple MSPPerformer
         //instances since you are passing unique MSPSignal
         //references per connection in your MSPPerformer
         //graph. In many instances though you can reuse
         //the vectors being passed into your perform routine.
         //See perform routine…

        tmp[0] = out[0].dupclean();

         noise.dspsetup(in,out);
         filter.dspsetup(in,out);
    }
```

```
    public void perform(MSPSignal[] in, MSPSignal[] out)
    {

        //use out tmp MSPSignal array we created as an
        //intermediate buffer so we can chain perform calls
        //This code is equivalent in the instance where you
        //are able to resue the signal objects passed to your
        //routine for performance duties:
        //
        // noise.perform(in,out);
        // filter.perform(out,in);

        noise.perform(in,tmp);
        filter.perform(tmp,out);
    }

}
```

Going Deeper with MSPObject

If you look at the javadoc for `MSPPerformer` you may notice that it is actually a subclass of another class in the `com.cycling74.msp` package known as `MSPObject`. `MSPObject` provides a slightly lower level interface to the Max application's MSP signal processing layer than `MSPPerformer`. Those developers who are familiar with the C API for writing signal processing externals for Max/MSP may notice that the interface exposed by the `MSPObject` class is very similar to the C API.

It consists of a single method named `dsp` which the developer must implement to describe how the external will handle it signal processing duties. As with the `dspsetup` method of `MSPPerformer`, the dsp method of `MSPObject` is called by the MSP signal compiler when it is adding your external to the signal processing chain but before your external is called upon to process any samples. Thus, as with `MSPPerformer`'s `dspsetup` method, it is an ideal place to cache expensive calculations which will remain valid over the entire course of the current signal processing context.

The main difference between the `dsp` method of `MSPObject` and the `dspsetup` method of `MSPPerformer` is that the `dsp` method needs to return a reference to the method which should be called upon by the signal compiler to process samples. The `dsp` method is defined as:

```
public Method dsp(MSPSignal[] ins, MSPSignal[] outs);
```

Notice that the `dsp` method returns an instance of `java.lang.reflect.Method` when specifying which method should be called upon to process sample vectors. Thus when implementing a signal processing object that extends `MSPObject` directly you will also need the following import at the top of your source file:

```
import java.lang.reflect.Method;
```

There is a utility method defined in `MSPObject` called `getPerformMethod` which makes the process of constructing the `Method` object returned by dsp a little easier. `getPerformMethod` takes a single argument, the name of the method you wish returned as a `java.lang.reflect.Method` instance.

The signature of your signal processing method needs to have a return type of void and two `MSPSignal` array parameters which will correspond to the inputs and outputs of your signal processing routine.

Here is the `MSPPerformer` example above implemented as an `MSPObject`:

```
import com.cycling74.max.*;
import com.cycling74.msp.*;
import java.lang.reflect.Method;

public class sigadd extends MSPObject
{
   private float addend;
   private Method _p1;
   private Method _p2;
   private Method _p3;
   public sigadd()
   {
      this(0);
   }
   public sigadd(float f)
   {
      addend = f;
      declareInlets(new int[]{SIGNAL,SIGNAL});
      declareOutlets(new int[]{SIGNAL});

      _p1 = getPerformMethod("p1");
      _p2 = getPerformMethod("p2");
      _p3 = getPerformMethod("p3");
      _p4 = getPerformMethod("p4");
   }

   public void inlet(float f)
   {
      addend = f;
   }

   public Method dsp(MSPSignal[] in, MSPSignal[] out)
   {
      if(in[0].connected && !in[1].connected)
         return _p1;
      else if( !in[0].connected && in[1].connected)
         return _p2;
```

```
      else if( in[0].connected && in[1].connected)
         return _p3;
      else
         return _p4;
   }

   public void p1(MSPSignal[] in, MSPSignal[] out)
   {
      float[] in1 = in[0].vec;
      float[] o   = out[0].vec;
      int vec_size = in[0].n;
      int i;

         //2nd signal inlet is not connected
         for(i = 0;i < vec_size;i++)
               o[i] = in1[i] + addend;
   }

   public void p2(MSPSignal[] in, MSPSignal[] out)
   {
      float[] in2 = in[1].vec;
      float[] o   = out[0].vec;
      int vec_size = in[0].n;
      int i;

         //1st signal inlet is not connected
         for(i = 0;i < vec_size;i++)
               o[i] = in2[i] + addend;
   }

   public void p3(MSPSignal[] in, MSPSignal[] out)
   {
      float[] in1 = in[0].vec;
      float[] in2 = in[1].vec;
      float[] o   = out[0].vec;
      int vec_size = in[0].n;
      int i;

         //1st signal inlet is not connected
         for(i = 0;i < vec_size;i++)
               o[i] = in1[i] + in2[i];
   }


  public void p4(MSPSignal[] in, MSPSignal[] out)
   {
           //neither signal inlet is connected
           for(i = 0;i < vec_size;i++)
              o[i] = addend;
   }
}
```

If you do choose to implement your Java signal processing externs using the lower level interface exposed by `MSPObject` it is suggested that you also implement the `MSPPerformable` interface as part of your class. This will allow your object to be used in the same fashion as the `myNoise` or `myFilter` object in the composition example above. Here is an example `MSPObject` derived external which also implements the `MSPPerformable` interface . It is a white noise generator.

```
import com.cycling74.max.*;
import com.cycling74.msp.*;
import java.lang.reflect.Method;

public class noise extends MSPObject implements MSPPerformable
{
    Method _p;
    long _last_val;
    private static final long jflone = 0x3f800000;
    private static final long jflmsk = 0x007fffff;
    public noise(Atom[] args)
    {
      declareInlets(new int[]{DataTypes.ALL});
      declareOutlets(new int[]{SIGNAL});
      _last_val = System.currentTimeMillis();
      _p = getPerformMethod("p");
    }

    public Method dsp(MSPSignal[] in, MSPSignal[] out)
    {
      post("dsp was called!");
      return _p;
    }
    private void p(MSPSignal[] in, MSPSignal[] out)
    {
      long tmp;
      long idum,itemp;
      int i;
      idum = _last_val;

      float[] o = out[0].vec;

      for(i = 0; i < o.length;i++)
       {
         idum = 1664525L * idum + 1013904223L;
         itemp = jflone  | (jflmsk & idum );
         o[i] = ((Float.intBitsToFloat((int)itemp)) *
                (float)2.0 ) - (float)3.0;
       }
      _last_val = idum;
    }

   //////MSPPerformable interface
```

```
    public void dspsetup(MSPSignal[] in, MSPSignal[] out)
    {
       dsp(in,out);
    }

    public void perform(MSPSignal[] in, MSPSignal[] out)
    {
       p(in,out);
    }
}
```

Appendix A: Event Priority in Max (Scheduler vs. Queue)

Introduction

The following article is designed to shed some light on the different priority levels of Max events. We will cover low priority events, high priority events, threading issues related to these two priority levels, and when it is important and/or useful to move events from one priority level to the other. We will also cover the MSP audio thread and how it can interact with low and high priority events. And finally, we will touch on some additional threading issues that will be of interest to Javascript, Java and C developers.

An event in Max is fundamental element of execution that typically causes a sequence of messages to be sent from object to object through a Max patcher network. The time it takes an event to be executed includes all of the time it takes for messages to be sent and operated upon in the Max patcher network. The way in which these messages traverse the patcher network is depth first, meaning that a path in the network is always followed to its terminal node before messages will be sent down adjacent paths.

Events can be generated from different sources—for example, a MIDI keyboard, the metronome object, a mouse click, a computer keyboard, etc. The first two examples typically have timing information associated with the event—i.e. the events have a scheduled time when they are to be executed. The latter two do not—i.e. they are simply passed by the operating system to the application to be processed as fast as they can, but at no specifically scheduled time. These first two events (i.e. MIDI and metro) fall into the category of high priority or scheduler events, while the second two events (i.e. mouse click and key press) fall into the category of low priority or queue events.

Overdrive and Parallel Execution

Only when overdrive is enabled, are high priority events actually given priority over low priority events. With overdrive on, Max will use two threads for the execution of events so that high priority events can interrupt and be executed before a low priority event has finished. Otherwise, Max processes both high priority and low priority events in the same thread, neither one interrupting the other—i.e. a high priority event would have to wait for a low priority event to complete its execution before the high priority event itself may be executed. This waiting results in less accurate timing for high priority events, and in some instances a long stall when waiting for very long low priority events, like loading an audio file into a buffer~ object. So the first rule for accurate timing of high priority events is to turn overdrive on.

With overdrive on, however it is important to note that messages can be thought of as passing through the patcher network simultaneously, so it is important to take into consideration that the state of a patch could change mid event process if both high and low priority events were passing through the same portion of the patcher network simultaneously. Using multiple threads also has the advantage that on multi-processor machines, one processor could be executing low priority events, while a second processor could be executing high priority events.

Changing Priority

Sometimes during the course of a high priority event's execution, there is a point at which the event attempts to execute a message that is not safe to be performed at high priority, or is a long operation that would affect the scheduler's timing accuracy. Messages that cause drawing, file i/o, or launch a dialog box are typical examples of things which are not desirable at high priority. All (well-behaved) Max objects that receive messages for these operations at high priority will generate a new low priority event to execute at some time in the future. This is often referred to as deferring execution from the high priority thread (or scheduler), to the low priority thread (or queue).

Occasionally, you will want to perform the same kind of deferral in your own patch which can be accomplished using either the defer or the deferlow objects. An important thing to note is that the defer object will place the new event to be executed at the front of the low priority event queue, while the deferlow object places the event at the back of the low priority event queue. This is a critical difference as the defer object can cause a message sequence to reverse order, while deferlow will preserve a message sequence's order. Another difference in behavior worth noting is that the defer object will not defer event execution if executed from the low priority queue, while the deferlow object will always defer event execution to a future low priority queue servicing, even if executed from the low priority queue.

There may also be instances when you want to move a low priority event to a high priority event, or make use of the scheduler for setting a specific time at which an event should execute. This can be accomplished by using the delay or pipe objects. Note that only the high priority scheduler maintains timing information, so if you wish to schedule a low priority event to execute at a specific time in the future, you will need to use the delay or pipe objects connected to the defer or deferlow objects.

Feedback Loops

You may encounter situations where the output of one sub-network of your patch needs to be fed back into that subnetwork's input. A naïve implementation of such a feedback loop without some means of separating each iteration of the subnetwork into separate

events will quickly lead to a stack overflow. The stack overflow results since a network containing feedback has infinite depth, and the machine runs out of memory attempting to execute such an event. In order to reduce the depth of the network executed per event, we can break up the total execution into separate events once for each iteration of the sub-network. This can be done either using either the delay or pipe objects if it is important that the events are executed at high priority, or the deferlow object if it is important that the events are executed at low priority. These objects can be placed at any point in the patcher network along the cycle, but typically it would be done between the ”output” node and the “input” node.

Note that the defer object alone will not solve the above problem, as it will not defer execution to a future event if called at low priority. However, the combination of delay and defer could be used to accomplish this task.

Event Backlog and Data Rate Reduction

With very rapid data streams, such as a high frequency metronome, or the output of the snapshot~ object with a rapid interval like 1ms, it is easy to generate more events than can be processed in real time. This can lead to event backlog—i.e. the high priority scheduler or low priority queue has more events being added than those it can execute in real time. This backlog will slow down the system as a whole and can eventually crash the application. The speedlim, qlim, and onebang objects are useful at performing data rate reduction on these rapid streams to keep up with real time. One common case of such backlog that has been reported by users is the connection the output of “snapshot~ 1” to lcd, js, or jsui; each of which defers incoming messages to low priority. Here the solution would typically be to use the **qlim** object to limit the data stream.

High Priority Scheduler and Low Priority Queue Settings

As of MaxMSP 4.5, there is an Extras menu item patch titled “PerformanceOptions”. This patch demonstrates how to set a variety of settings related to how the high priority scheduler and low priority queue behave--both the interval at which the scheduler and the queue are serviced as well as the number of events executed per servicing (aka throttle). There is also a mechanism called scheduler slop that can be used to balance whether long term or short term temporal accuracy is more important, as well as settings for the rate at which the display is refreshed. Each of these settings are sent as messages to Max, and while these values are not stored in the preferences folder, you can make a text file that is formatted to send these messages to Max and place in your C74:/init/ folder if you want to set these values to something other than the default each time you launch Max. An example which would set the default values would contain the following:

max setslop 25;

max setsleep 2;

max setpollthrottle 20;

max setqueuethrottle 10;

max seteventinterval 2;

max refreshrate 30;

max enablerefresh 1;

For more information on the various settings exposed by this patch please read the descriptions contained in the Performance Options patcher.

Scheduler in Audio Interrupt

When “Scheduler in Audio Interrupt” (SIAI) is turned on, the high priority scheduler runs inside the audio thread. The advancement of scheduler time is tightly coupled with the advancement of DSP time, and the scheduler is serviced once per signal vector. This can be desirable in a variety of contexts, however it is important to note a few things.

First, if using SIAI, you will want to watch out for expensive calculations in the scheduler, or else it is possible that the audio thread will not keep up with its real time demands and hence drop vectors. This will cause large clicks and glitches in the output. To minimize this problem, you may want to turn down poll throttle to limit the number of events that are serviced per scheduler servicing, increase the I/O vector size to build in more buffer time for varying performance per signal vector, and/or revise your patches so that you are guaranteed no such expensive calculations in the scheduler.

Second, with SIAI, the scheduler will be extremely accurate with respect to the MSP audio signal, however, due to the way audio signal vectors are calculated, the scheduler might be less accurate with respect to actual time. For example, if the audio calculation is not very expensive, there may be clumping towards the beginning of one I/O vector worth of samples. If timing with respect to both DSP time and actual time is a primary concern, a decreased I/O vector size can help improve things, but as mentioned above, might lead to more glitches if your scheduler calculations are expensive. Another trick to synchronize with actual time is to use an object like the date object to match events with the system time as reported by the OS.

Third, if using SIAI, the scheduler and audio processing share the same thread, and therefore may not be as good at exploiting multi-processor resources.

Javascript, Java, and C Threading Concerns

The first thing to note is that at the time of this writing the Javascript implementation is single threaded and will only execute at low priority. For timing sensitive events the js and jsui objects should not be used for this reason. However, this may change in a future release.

External objects written in both Java and C support execution at either low or high priority (except where those objects explicitly defer high priority execution to low priority). When writing any Java or C object this multithreaded behavior should not be overlooked. If using thread sensitive data in your object, it is important to limit access to that data using the mechanisms provided in each language—i.e. the synchronized keyword in Java, and critical regions, mutexes, semaphores, or another locking mechanism in C. It is important not to lock access around an outlet call as this can easily lead to thread deadlock. Deadlock is where one thread is holding one lock waiting on second lock held by a second thread, while the second thread is waiting on the lock that is held by the first thread. Thus neither thread can advance execution, and your application will appear to be frozen, although not crashing.

Finally, if you are writing an object in Java or C which creates and uses threads other than the low priority and high priority threads in Max, you may not make outlet calls in the additional threads your object has created. The only safe threads to output data through a Max patcher network are the low priority queue and high priority scheduler threads. The Max Java API will automatically detect when attempting to output into a patcher network from an unsafe thread and generate an event to be executed by the low priority queue when using the outlet() method, and generate an event to be executed by the high priority scheduler when using the outletHigh() method. In C there is no such safety mechanism when calling from an unsafe thread, so it is up to the C developer to generate such events using defer_low(), or and qelem functions for low priority events, and schedule() or the clock functions for high priority events. Otherwise the C developer risks crashing the application.

More information on related issues can be found in the Java and C developer SDKs. A good introduction to the concepts behind multi-threaded programming and other scheduling concepts, may be found in “Modern Operating Systems”, by Andrew S. Tanenbaum.