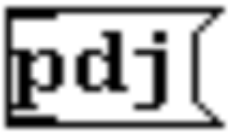

pdj : java interface for pure-data electric-one software

Legal Information

PDJ is not endorsed or known of by Cycling74. It was written by a Max/MSP user [me] that wanted to use his objects in both environment.

PDJ is not 100% compatible with mxj on Max/MSP, I am doing my best to keep PDJ API compatible. The most important is to keep the object IO (inlet/outlet interaction) as close has possible.

This document is nowhere from professional, please note that English is my second language.

Feel free if you want to contirubte :

Pascal Gauthier

asb2m10 at users.sourceforge.net

Licence

Table Of Content

Getting Started

Introduction

PDJ is a pure-data external that help to bridge the Java environment to pure-data. Java has multiple advantages over C and one of the them is the fact that you can write your external on one platform and the run it on any other platform that runs Java. This also apply on MAX/MSP since the PDJ API is based on the mxj implementation.

While Java is certainly not the only language you should use with pure-data, it gives you these advantages :

1. It is multi-platform, you can run your external on Windows/OSX/Linux with the same package.
2. It is multi-program, you can run your external on pure-data/MaxMSP with the same package.
3. You don't have to restart pure-data each time you recompile your external
4. Access to a wide range of open source Java libraries that works right out of box.

So in theory, you can write a Java external object on Linux with pure-data and then give it to your friend that runs MAX/MSP on OS X PowerPC. You don't have to recompile your external, the package remains the same for all platforms and program.

Building PDJ

Unless you are running Linux, we strongly advise you to use the binary distribution. Before building, make sure you have these pre-requisite :

1. Java development Kit version 1.4 or better
2. Pure-Data 0.40 or better
3. GNU GCC (for Linux/OS X) or Microsoft MSVC (Windows)
4. Apache ANT 1.6 or better

You must edit the file [your platform]-build.properties that contains the definition of your build environment. Usually, you will have to change the *jdk.home* and the *pd.home* properties. On OS X, you can point your pd.home to the directory where pure-data is installed. If for example it is installed in */Application,* you can point the the directory */Application/pd.app/Contents/Resources*.

You can build PDJ by simply issuing “ant”. This will a create a PDJ distribution in the directory *dist*. You can use the ant target “ant test” to

test your PDJ build.

Configuring PDJ

Most of the PDJ configuration properties is available in the file *pdj.properties*. This file can be in two directories if you are using the source distribution, so please use the file in the "*dist*" directory since the "*res*" directory is used for source control.

Under OS X, you will have to put the PDJ distribution directory to the pure-data search path. If PDJ is not in the pure-data search PATH, it will not be possible for PDJ to find the configuration file *pdj.properties*.

Key Listing

Key pdj.classes-dir=[directory]

Load your classes directly from a directory. Unlike mxj, you can also put your sources in that directory. Each time you will load a .class, PDJ will compare the time of when the .class has been generated and when the .java has been modified. If the Java source file is younger, PDJ will try to compile the classes with Javac. If the compile was successful, PDJ will simply load the class as an external into pure-data. You don't have to restart pure-data each time you modify theses classes since PDJ will reload your the classe each time it is loaded. Only one directory must be specified.

Key pdj.classpath=[path]

Load your classes from a dynamic classpath. Unlike a real classpath, if you point to a directory, PDJ will try to find any .jar in this directory and add it to the real Java classpath. Each time you re-instantiate a pdj object, the Java classpath is rebuild so you don't have to restart pure-data when you have changed a .jar or a class file.

Key pdj.system-classpath=[path]

Unlike the dynamic classpath, this classpath is static and will only be loaded once the first PDJ object is used. In some circumstances, you will have to put your classes in this classpath. Some projects that use Java reflection will not work with the dynamic classpath since it is always rebuilded when the MaxObject is loaded.

Key pdj.verbose-classloader=[boolean]

This will tell pdj to print the current classpath each time a class is trying to be loaded. This is often useful if your class cannot be loaded since

Java cannot find the specified class.

Key pdj.JAVA_HOME=[path]

This will tell PDJ where to find the Java virtual machine. Usually, you can comment out this field since PDJ will try to find the JVM by looking into the environment of the operating system.

Key pdj.vm_args=[args]

By using this key you can specify additional parameters that will passed when the JVM will be loaded.

Using OS X

If you are using pdj on Mac OS and the Java code uses AWT/Swing class, you will need extra pd parameters to make it work. The problem is that OS X have a very strict GUI event mechanism that pd is not compatible with. Linux and Windows don't have this limitation.

To counter this problem, pdj also comes with a custom pd scheduler. This scheduler will simply process the GUI event in the main thread and it will create a secondary thread that will process the standard pd scheduler.

From pd 41.2, you will need to apply a special patch since the portion of code in pure-data that read the “custom scheduler” is broken. You can apply the patch file osx_extsched_fix.patch in the src/pd_patch directory. This patch will be submitted to pure-data development team.

Once the patch is applied, you need to initialize the pdj scheduler. This is done by following menu Pd->Preference->Startup.

Add the following argument from “Startup Flags”

```
-schedlib [full path of the pdj external without the extension]
```

Be sure to put the right path since this can crash pure-data at startup. If it is the case, you can erase your pure-data parameters by deleting this file :

~/Library/Preferences/org.puredata.pd.plist

If the pdj external scheduler is initialized correctly you should see to message at startup :

```
pdj: using pdj scheduler for Java AWT.
```

Working Environment

Before we get starting working with PDJ, we will look at different ways to work on a Java project.

Directly from the pdj project directory

Unlike mxj, it is possible to write .java files that will automatically be compiled by PDJ. This can be useful if you need to write a quick proof of concept class or a very simple object.

First you need to configure the PDJ properties named pdj.classes-dir. Each time you will ask pure-data to instanced a new PDJ object, this tool will look if there is a java file in the same directory. If this property is not configured, PDJ will look in the *classes* directory in the same path where it is installed.

You can try out new this feature by simply creating a new java file with this content. This file must be dropped in the *classes* directory.

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

public class void Test {

      public bang() {

                  post(“Hello World”);

      }

};
```

Then you can simply call the PDJ with the name “pdj Test”. Later, you can connect a bang message to this object and you should see a “pdj: Hello World” message in the pure-data console when the bang message is sended.

From the command line

Apache ANT is a great tool for working on the command line. You can create very simple ANT build file that will be compatible with most Java environment. To create a new project; into your main project directory, create 3 sub-directory :

- /lib where you can put the pdj.jar reference (copy it from the PDJ distribution)
- /src where you can put your java source files
- /work where you can put the compiled java source file.

```
<project name="pdj-project" default="package">
     <target name="package">
                <javac srcdir="src" destdir="work">
                     <classpath>
                          <fileset dir="lib" includes="*.jar"/>
                     </classpath>
                </javac>
     </target>
</project>
```

Later, you can configure the *pdj.classpath* property to point where the directory "work" is. Then, simply issue "ant" from the command line when you need to rebuild your project.

From Eclipse IDE

Eclipse is also a great tool to use to develop Java programs. Building a pdj external with Eclipse is simple :

- Select "File" -> "New" - > "New Java Project" from the IDE menu
- In the project layout section, you will have to select "Create separate folders for source and class files". Click on "Next"
- In the Java settings, select the "Libraries" thumb.
- Click on "Add External JARS...".
- Select the file : pdj.jar that included in the PDJ distribution

Then, you can configure the "*pdj.classpath*" property that is specified in the pdj.properties files. You should point to the "bin" directory of where your project has been created by Eclipse.

One of the great advantages of use Eclipse is that your Java classes will automatically be compiled each time you modify the Java file.

Working with Atoms

All information that will be sended and received by pure-data will be encapsulated into atoms.

Since atoms are type-less, they are encapsulated into a Atom (from com.cycling74.max.Atom) type of object.

Atom.getInt() or Atom.getFloat() to get the number value.

Atom.getString() to get the string content.

MaxObject Class

Creating the Object

You can use two constructor: MaxObject() or MaxObject(Atom[] args). If arguments are specified when the object is created, MaxObject(Atom[] args) will be called. Failing to override this constructor when arguments are added to the contructor will throw an exception. If no arguments are added to the object but the MaxObject(Atom[] args) is overriden then this constructor will be called with an empty atom array.

Object inlets/outlets declaration

Unlike mxj with Max/MSP, PDJ doesn't have any typed inlets/outlets (to the exception for signal (audio) inlets/outlets). To assure the compatibility with Max/MSP, the API has been match and any typed declaration will always be considered as “ANYTHING”.

If our object only receives/output data, you can use the MaxObject method :

declareIO(int inlets, int outlets)

Inlets and outlets must be defined in the object constructor. Falling to do so will throw an exception.

Handling inlet data

Depending on the data that has been received by the inlet, a specific Java method in your object will be called. The know from which inlet your object received the data, you can use the method getInlet(). This will return the inlet number from which it has received the data.

- If it is a bang message then void bang() method will be called. If the bang method has not been overridden then void

anything(“bang”, null) will be called.

- If it is a float, the method void inlet(float f) will be called. If the void intlet(float f) has not been overridden, PDJ will try to look for void inlet(int i), this is done to assure the Max/MSP compatibility. If void inlet(int i) is not overridden then PDJ will call the anything method with argument void anything(“float”, Atom[1] = { Atom.newAtom(f) }).
- If it is a list, the method void list(Atom content[]) will be called. If the method is not overridden, then anything(“list”, Atom content[]) will be called.
- If it is a symbol, PDJ will try to map this symbol to a Java method. For example if the message “foo” is received, the method “foo” will be called. If this method is not overridden, then the method void anything(“foo”, null) will be called. If the symbol is within a list, the method void foo(Atom []args) will be called. Same thing applies if the method has not been overridden with void anything(“foo”, Atom args[]);

Information Inlet

Unless you specify not to create an “info inlet”, PDJ will always create a last inlet that will support “info message”.

Sending Atom via outlet

It is possible to send Atom by using the outlet method. This method has been overridden to support multiple type. Once it is parsed, a real pure-data atom will be created supporting the original Java type.

Use this method to send data to an outlet

outlet(int outlet_number, <data>)

Processing DSP signal

To be able to process audio signal with your MaxObject you will need to extends the class MSPObject and not MaxObject. All the methods that you used with MaxObjects are still available since MSPObject extends MaxObject.

The signal pipe : MSPSignal

Signal audio is always passed into a MSPSignal. For every signal inlet or outlet the is an attached MSPSignal. In every MSPSignal, it is possible to know sampling sampling rate (member *sr* MSPSignal) and the number of sample (member *n* in MSPSignal) to be processed. This object will be instantiated by PDJ when pure-data will start DSP processing. See DSP initializations.

Inlets/Outlets creation

Because your object will process audio data, signal inlets or outlets will have to be defined. Unlike declareIO(int inlet, int oulet), data type (between message inlets/outlets and audio inlets/outlets) will have to be identified.

This quickie method

declareTypedIO(String ins, String outs)

To identify audio data, the type 's' must be used. If you need to define a data inlet/outlet, you can use “a” to define “ANYTHING”.

For example, if you need to create 2 signal inlets, 1 message inlet and 1 signal outlet, 2 message outlet, you can use :

```
declareTypeIO(“ssa”, “saa”);
```

Because pure-data does not type any data, the outlet type “ANYTHING” will be map to a standard message inlet/outlet. Please note that the order given in the format is important..

DSP initialization

Before pure-data starts processing audio signal, a special startup method will be called with name “dsp”. In this method, you will receive the initial MSPSignal objects. For the first time, you will be able to know the block size of each DSP samples (from the number of samples *n* member) and the sample rate used (from the *sr* member). The number of sample and sample rate will be the same across MSPSignal given by PDJ.

This method must return a DSP processing method that will be called at each DSP cycle. The method can be referred by using the special helper *getPerformMethod*.

This is an example of a “dsp” method with a performer :

```
public void dsp(MSPSignal[] ins, MSPSignal[] outs) {
      return getPerformMethod(“doit_dsp”);
}
public void doit_dsp(MSPSignal[] ins, MSPSignal[] outs) {
      for(int i=0;i<ins[0].n;i++) {
            outs[0] = ins[0] + 1;
      }
}
```

You decide on the name of the perform method, the only limitation is that your method must return “void” and have a “MSPSignal[], MSPSignal[]” parameters.

DSP perform

Once the DSP method given to PDJ, pure-data will call your method at each DSP cycle. It is in this method that your data will be processed. Note that this method must be optimized because it will be called multiple time within a second. It is strongly advised to not create any objects in this method. If you have any objects to create, create them in the “dsp” startup method and cache theses objects for maximum performance.

If an exception occurs in this method, the signal will be reset to “0” and the performer method will never be called until it is re-initialized.