ProcSee Java API tutorial

This tutorial is intended to give the user a hands-on experience on how to develop an external application for ProcSee using the Java API. It is not a tutorial in designing a ProcSee application. This information is found elsewhere in the ProcSee tutorial distributed with ProcSee. Instead, this tutorial focuses on the external application, i.e. how to interact with ProcSee from a Java application. You do not have to be an expert, but some knowledge about the Java language is recommended, or you have some experience from other languages, like C#, C++, etc. Still, it is required that you do have some knowledge of the fundamental concepts of OOP. Therefore, this tutorial will not describe in detail the Java code needed to implement the framework based on the Java Swing library, i.e. how to create the menus, message boxes, etc. However, this tutorial will describe in detail each line of code that involves the Java API.

It is assumed that you are acquainted with one of the integrated development environments (IDE) in Java, like Netbeans or Eclipse. Notice that the screenshots, the code examples and the description in this document are based on the Netbeans IDE.

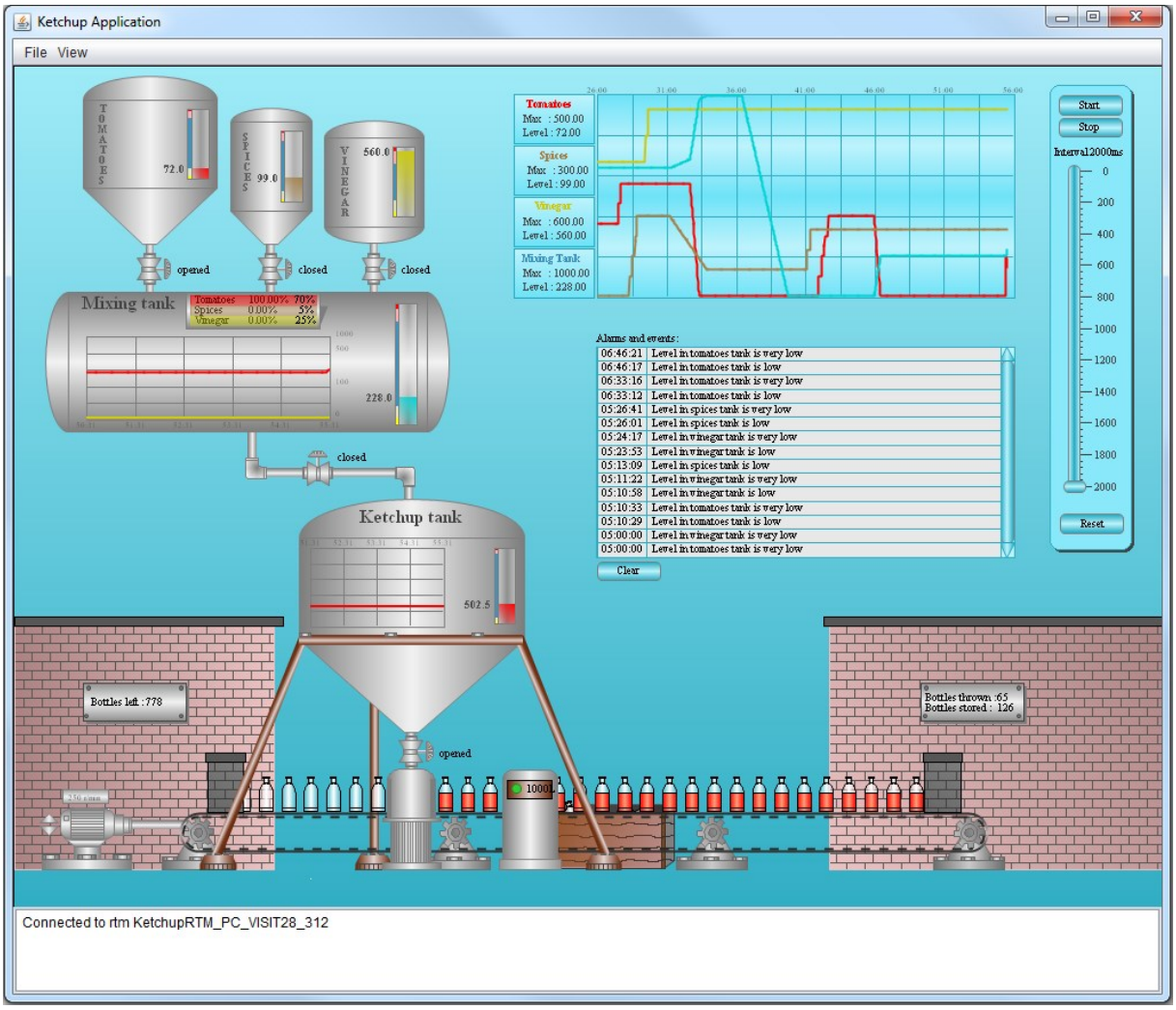
This tutorial is based on the Ketchup demo application distributed with ProcSee. It is located in the $(PROCSEE_DIR)/demo/ketchup/$(ARCH) directory.

Before we start developing the external java application you should familiarize yourself with the Ketchup demo application. This ProcSee application is the base for the external application we are going to develop in this tutorial. Try running it, either by double clicking on the KetchupApp.pctx file, or selecting the Ketchup Demo under the demo folder in the ProcSee menu. Note that this demo application uses a simulator developed in the pTALK language to get live updated values in the pictures. Next in this document we will be developing parts of this simulator in the Java language.

In the tutorial, you will develop an external application that step by step covers most parts of the Java API. The external application will have functionality to start an RTM, update variables, respond to method calls from an RTM, and export a window where an RTM can display its pictures. When completing this tutorial you should have learned how to:

1. Prepare the IDE for the Java API.
2. Create and initialize the external application using the Java API.
3. Create a message output provider.
4. Initialize and register the external application with the ProcSee control server.
5. Initiate a connection to a ProcSee RTM.
6. Export a window to the ProcSee RTM where pictures can be displayed.
7. Start the ProcSee RTM from the external application.
8. Create Java wrapper classes for the struct definitions.
9. Create and link to variables in the ProcSee RTM.
10. Update and transfer variables to the ProcSee RTM.
11. Register remote functions which can be called from the ProcSee RTM.

12. Use the execute function to run pTALK commands in the ProcSee RTM.



1. Prepare the IDE for the Java API

This description applies to the Netbeans IDE only. If you are not a Netbeans user, refer to the manual on how to create a new Java application. We begin by working through the New Project wizard in the Netbeans IDE. At the end we will have a basic Java project that every Netbeans IDE requires to build an application. Start the Netbeans IDE. Select New project… in the File… menu. This action opens the New Project dialogue window. In this window select new Java Application and click Next. Specify the project location and name the project KetchupApp. Leave the Create Main Class check box enabled, i.e. let the IDE create the main class. We have now made the necessary changes and settings in the wizard. Click the Finish button to complete the creation of the Java application. The Netbeans IDE has now generated the KetchupApp project containing sources and project metadata needed in order to build the Java application.

The next step is to setup the dependency to the Java API library. The Java compiler will issue errors in the output window complaining about unknown classes if you forget to add this dependency before building the application. Therefore, in the Netbeans IDE open the Project

Properties window. Right-click the KetchupApp project and select Properties in the pull down menu. In the categories view, select Libraries. Click on the Add JAR/Folder button which opens a file selection box. In the file selection box move to the $(PROCSEE_DIR) /lib/japi folder and select the JAR file ProcSeeApiJava.jar. Click on the OK button. The ProcSeeApiJava.jar should now appear in the Compile time libraries list. Click the OK button to close the Project Properties window.

Please note that the bin folder of the JRE or JDK needs to be set in PATH environment variable, for the JAVA API library to be successfully loaded.

You have now completed this step. Next in this tutorial we will be creating the first class utilizing the Java API.

2. Create and initialize the external application using the Java API

Now, we can start developing the external application. The main focus in this section and the remaining parts of this tutorial is on the class ProcSee. This class contains the functionality needed to get the ketchup application up and running.

Start by adding a new class deriving from the Swing component JFrame in the ketchupapp package (in the KetchupApp project). Give the class the name ProcSee and make sure that the package name is ketchupapp. Click on the Finish button to close the window. The IDE has now created the source files. The generated class is shown in the IDE in a design view and a source view. For the time being ignore the source view. We will be coming back to this view later on in this tutorial.

Before continuing with the ProcSee class, create an instance of this class in the body of the static main method in the KethcupApp.java source code. It is important that this class instance runs in the AWT event thread. The AWT event thread is the background event dispatching thread used by the AWT and the Swing libraries that causes the components to update and redraw themselves. The events generated in this thread are primarily update and input events such as mouse and keyboard. Due to the nature of the event dispatching thread, GUI applications utilizing the Java API must be created in this thread. Invoking them from other threads risks thread interference or memory consistency errors.

Switch back to the class KetchupApp and find the static *main* method. This is the first method invoked by the Java virtual machine when the external application starts. Put the following lines of code in this method.

```
public static void main(String args[]) {

        java.awt.EventQueue.invokeLater(new Runnable() {

            @override public void run() {
                new ProcSee().setVisible(true);
            }
        });
}
```

This code creates and runs the ProcSee instance in the AWT event thread, and it makes the instance visible on the screen. The KetchupApp class is now finished.

Switch back to the design view for the *ProcSee* class. Now, create a menu bar with a file menu and an exit button. Add an action listener for the exit button. Switch to the source view and implement the exit action listener. In the exit action listener open a confirm dialogue box (class JOptionPane) where the user can choose whether to exit the external application or not.

You can now build the project which should compile without any errors and warnings. For the time being this application is not very interesting. It displays a frame window with a menu bar. Try running it to check that you have added the source code described so far in this tutorial.

3. Create a message output provider

Before continuing with the PcProcess class we need to explain the message output functionality in the Java API, which is provided by the class PcMessageOutput. The messages issued in the Java API are by default sent to standard output, which simply doesn't fit very well into a GUI application. By providing a message output class the messages can be redirected to an output window, like a JList component or a log file, for example.

Each message issued in the Java API is associated with a message category, where the message category is one of the predefined categories defined in the Java API, like error, warning, debug, flow, etc. The PcMessageOutput class offers functionality to suppress these message categories. By default fatal errors, errors, warnings and information messages are enabled. Note that fatal errors and errors should never be suppressed by the external application. Such information should always be reported to the user so she/he can take appropriate actions. Normally the debug categories (five different debug levels exist in the API) are suppressed by default and should only be enabled when running the external application in a development phase.

The PcMessageOutput class also offers functionality to customize the message format in the Java API. The message format is a concatenation of pure text and directives. A directive is a sequence of characters surrounded by curly braces, {}, recognized by the PcMessageOutput class, for example "Message: {MSG}", "{TIME:T} {MSG}", etc. The default message format in the Java API is "{TIME:T} {CAT} {MSG}", which is a concatenation of the date and time ({TIME:T}, plus the category ({CAT}) and the actual message to print ({MSG}).

We recommend that you always create an instance of the class PcMessageOutput before calling any method or instantiating any other class in the Java API. The information provided in the message outputs can be very useful when the application fails for various reasons. For GUI applications the messages issued in the Java API is silently ignored without an output provider.

Now, switch back to the design view. Create a list component in the lower part of the window of the Java Swing class JList. Click on the JList component in the Swing palette and drag it onto the design surface. Name this JList component `mMessageOutputList`. This component should be horizontal resizable and anchored to the bottom. It is used by the external application to print messages from the external application and messages issued in the Java API. Associated with the list is a model, which manages the items in the list. In this tutorial we will use the class *DefaultListModel* as model (described next).

Switch back to the source view again. Before we instantiate the default message output provider, add the following import statements in your ProcSee source file.

```
import javax.swing.DefaultListModel;
import no.hrp.procsee.api.messageoutput.PcMessageOutput;
import no.hrp.procsee.api.messageoutput.PiMessageListener;
```

Next, in the ProcSee class, create two private attributes. The first attribute is the message output provider used by the Java API. Name the attribute mMessageOutput. The data type is of the class PcMessageOutput. The second attribute is the model used by the Swing component JList. This attribute is of the class DefaultListModel. Give this attribute the name `mMessageOutputModel`.

```
private PcMessageOutput   mMessageOutput;
private DeafaultListModel mMessageOutputModel;
```

In the class constructor after the call to the initialization routine `initComponents()` make a call to the method `initMessageOutput()`. Add the following lines of code to your ProcSee class.

```
private void printMessage( String message ) {
   mMessageOutputModel.add( 0, message );
   System.out.println( message );
}

private void initMessageOutput() {
    mMessageOutputModel = new DefaultListModel();
    mMessageOutputList.setModel( mMessageOutputModel );


    mMessageOutput = new PcMessageOutput( "{TIME:T} {MSG}", new PiMessageListener()
    {
      @Override public void onMessage(PeMessageCategory category, String message) {
         printMessage( category + message );
      }
    });

    mMessageOutput.disable(PeMessageCategory.PeInformation);
}
```

In the source code above the output message is printed to standard output in addition to being added to the list component. In a real application, the message would probably be dumped to a file as well (using another message output instance providing more information). Notice that the Java API supports multiple output providers with different or equal message formats. But this is out of the scope for this tutorial. In the source code above is the information category disabled (the call to the class PcMessageOutput method `disable`). This category produces lots of messages that we don't need to pay any attention to, but can be useful in a development phase for debugging purposes.

4. Initialize and register the external application with the ProcSee control server

So far in this tutorial you have learned how to setup the dependency to the ProcSeeApiJava.jar library, create a ProcSee class and initialize a message output provider. Now, it's time to initialize the Java API and register the external application, i.e. register the name of the external application and supply other process related information to the ProcSee control server. The class *PcProcess* provides this functionality in the Java API.

Create a new attribute in the *ProcSee* class of type *PcProcess* and give it the name *mProcess*. In the source code declare this attribute after the *PcMessageOutput* attribute, like:

```
private PcProcess mProcess;
```

In the *ProcSee* constructor, put the call to the method *initProcess()* after the call to the method *initMessageOutput()*, like:

```
public ProcSee() {
   initComponents();
   initMessageOutput();
   initProcess();
}
```

The *initProcess()* method creates the *PcProcess* instance which registers the process in the ProcSee control server. The first argument to the constructor is the name of the process. This name must be unique. The *PcProcess* constructor will otherwise fail to initialize the instance and throw an exception. Notice that the Java API provides a helper method which creates names that are guaranteed to be unique. The name returned is based on a user defined prefix (and postfix), plus the name of the host computer and the PID (process identifier). In the tutorial the prefix is "KethcupApp". An example of a unique name is: "KethcupApp _GANDALF_2469", where GANDALF is the name of the computer, and 2469 is the PID. In this tutorial we will be using this method to create unique process names. See the source code below.

The process class argument to the *PcProcess* constructor is optional. In this tutorial the process class is called KETCHUP_APP, which provides extra information about the process. This data can be useful when requesting data from the *control* server or when running the utility application *controlutil*. Therefore, we recommend that you always provide the process class when initializing the external application.

```
private void initProcess() {
  try {
     String name = PcControlUtils.createUniqueName("KetchupApp");
     mProcess = new PcProcess( name, "KETCHUP_APP" );

  } catch (PcException ex) {
     JOptionPane.showMessageDialog( this, ex.getMessage(), "PcProcess failed",
                                    JOptionPane.ERROR_MESSAGE );
     System.exit( 1 );
  }
}
```

The following new import statements must be included in the import section:

```
import no.hrp.procsee.api.PcProcess;
import no.hrp.procsee.api.control.PcControlUtils;
import no.hrp.procsee.api.exceptions.PcException;
import no.hrp.procsee.api.messageoutput.PeMessageCategory;
```

Now, compile and run the external application. This is done to test that you have the environment variables set correctly. If the external application fails to start, check that the environment variables PROCSEE_DIR and ARCH are set properly.

5. Initiate a connection to a ProcSee RTM

The Java API provides two classes which can be used to represent an RTM in an external application. The names of these classes are PcRTMAsClient and PcRTMAsServer (derived from the base class PcRTM). The class PcRTM is an abstract base class which cannot be instantiated directly in the external application. It is used as arguments in methods calls invoked by the Java API. These classes represent the external application's view or link to a specific ProcSee RTM.

In this tutorial the external application will be a client to the RTM, i.e. the external application initiates the connection to the RTM. Therefore, we will be using the class PcRTMAsServer.

Notice that the class PcRTMAsClient is used when the RTM is responsible for establishing the connection to the external application, i.e. the RTM is the client. Instances of the class PcRTMAsClient are created on requests from RTMs in the onCreate call-back method of the interface PiConnectFactory .

The following functionality is provided by the PcRTM derived classes:

- executing pTALK commands.
- read and link to variables in the RTM.
- register remote functions.

Before we can use the class PcRTMAsServer in the tutorial we have to add the following import statements in the source code's import section. These classes are used later on in this tutorial.

```
import no.hrp.procsee.api.rtm.PcRTM;
import no.hrp.procsee.api.rtm.PcRTMAsServer;
import no.hrp.procsee.api.rtm.PaConnectionStateListener;
import no.hrp.procsee.api.rtm.PeConnect;
import no.hrp.procsee.api.rtm.PeDisconnect;
import no.hrp.procsee.api.rtm.PeDisconnectInfo;
```

In the ProcSee class add a private attribute of the class PcRTMAsServer. Give this attribute the name mRTM.

```
private PcRTMAsServer mRTM;
```

Now, we have to initialize the mRTM attribute. The first argument to the PcRTMAsServer constructor is the name of the RTM to connect to. If the RTM with the specified name isn't running, you have to manually start an RTM with that name (start it with the –n option and the –r KetchupApp.pctx arguments). Later in this tutorial you will learn how to start the RTM from the external application. The second argument specifies the name of the ProcSee application in the RTM. This is the current ProcSee application the external application will be connected to in the RTM. If the application doesn't exist in the RTM, an empty application will be created with the specified name. The last argument is a connection state listener which provides information when the connection state changes, for instance when a connection is established or broken. Extra information is provided in the arguments to the callback methods in this interface. For simplicity, we use in this tutorial the adapter class PaConnectionStateListener instead of the interface PiConnectionStateListener. Notice that you should always implement the onError callback of the PiConnectionStateListener interface when running more complex systems.

The callback methods in this tutorial just calls the *printMessage*() method. Note that we haven't provided any component in the external application's GUI that visualizes the connection state (and we are not going to either). The connection state is displayed in the message output list component. Add the following lines of code in the class ProcSee to create and initiate the connection to the RTM.

```
private void initRTM() {
   try {
```

```
      mRTM = new PcRTMAsServer( "rtm", "KetcupApp", new PaConnectionStateListener()
      {
         @Override public void onConnect(PcRTM rtm, PeConnect type) {
            printMessage( "Connected to rtm " + rtm.name() );
         }

         @Override public void onDisconnect(PcRTM rtm, PeDisconnect type,
                                            PeDisconnectInfo info) {
            printMessage( "Lost connection to the rtm " + rtm.name() );
         }
       } );

     mRTM.connect();

    } catch (PcException ex) {
        printMessage( "PcRTMAsServer error " + ex.getMessage() );
    }
}
```

Notice the last method call *connect()* in the source code above. The external application will never try to connect to the RTM if you forget to call the connect method.

In the ProcSee constructor after the call to *initProcess*() add a call to the *initRTM*() method just implemented.

```
public ProcSee() {
   initComponents();
   initMessageOutput();
   initProcess();
   initRTM();
}
```

Now, compile and run the external application again. Be sure that the RTM is still running with the ProcSee application, Ketchup. This time the external application should be able to connect to the RTM. A message is issued in the message output window when the connection is established.

6. Export a window to the ProcSee RTM where pictures can be displayed

This section focuses on how to make a drawing surface in the external application available to the RTM, i.e. a window the RTM can use to display pictures.

This description applies to the Netbeans IDE only. If you are not a Netbeans user, refer to the manual on how to add GUI components to a palette in your favorite IDE. The classes PcWindow and PcjWindow must be added to the graphical palette in the IDE before we can start creating an exported window. Follow the next steps to add these classes. First, right click with the mouse in the *Palette* window in the Netbeans IDE. From the pull down menu that appears, select Create New Category. This action opens a dialogue window where you can specify a category name. Enter a name, for instance ProcSee and press OK. The category you entered will appear in the graphical palette. Now, right click with the mouse in the palette again, but this time select Palette Manager. In the dialogue window that appears on the screen, select the new category you just created. Next, press the button Add from JAR… which opens

the file dialogue window. Move to the $(PROCSEE_DIR)/lib/japi directory and select the Java API jar file, i.e. ProcSeeApiJava.jar. Press OK in the file dialogue window and the Palette Manager window. The classes PcWindow and PcjWindow should now appear in the graphical palette under the category that you created.

What is the difference between the classes PcWindow and PcjWindow, and which of these classes are we going to use in the tutorial, and why? The answer to the first question is that they are quite similar. The only difference is their look and feel. The class PcWindow is based on the Abstract Window Toolkit (AWT), while PcjWindow uses the look and feel of the Swing toolkit. We will be using PcjWindow in this tutorial because the Swing toolkit seems to be most popular.

Note that the class PcjWindow is not a lightweight component as most of the other components in the Swing library, because the RTM needs a native window surface where it can display its pictures. Therefore, both the PcWindow and PcjWindow classes are heavyweight components, i.e. they are associated with their own native screen resource (a window). There is one problem with mixing lightweight and heavyweight components in an application, and that is when lightweight components are placed in front of a heavyweight component (in the windows' Z order). The problem is that the lightweight components don't appear on the screen, they are invisible because they are put behind the heavyweight components. In the external application this is the case for the menu bar, which has lightweight pull down menus placed in front of the PcjWindow class. Fortunately, the Swing toolkit provides a setting which forces the JPopupMenu to disable the lightweight popup. It supports both a static and a non-static method. In this tutorial we will use the static method which sets the default for all JPopupMenu components. In the ProcSee class insert the following code.

```
private void initialize() {
   JPopupMenu.setDefaultLightWeightPopupEnabled( false );
}
```

Call this method from the constructor before the call to the method initComponents(). It is important that it is called before and not after the call to initComponents() because this setting is used when the pulldown menus are created and not afterwards.

Now, we are ready to create an instance of the class PcjWindow. Switch to the design view. In the palette locate PcjWindow. Click on the PcjWindow object and drag it to the design surface. You don't need to struggle with the layout manager to control its size and position; it will automatically fit into the position and the size where you drop it. Attach it to the upper left corner and the lower right corner of the drawing surface. Let it be resizable both in the vertical and horizontal direction. Name this component mExportedWindow. In the properties window, locate the windowName property, and give this property the name ExportedWindow. This is the name of the exported window within the ProcSee RTM.

Next, find the pictureName property and enter the name of the initial picture to display in the window. The name of this picture in the KetchupApp demo application is KetchupProcess. Switch back to the source view again.

There is one issue that has to be resolved before we can compile and run the external application. That is, which RTM is associated with the PcjWindow instance. The Java API does not provide this functionality in the design view. It cannot be set via a property. The class PcjWindow (and the class PcWindow for that matter) has a method called setRTM which creates this association. It accepts a single argument of type PcRTM. Now, scroll down to the initRTM() method. Insert the following code line after the call to mRTM.connect().

```
mExportedWindow.setRTM(mRTM);
```

That's it! If you have followed this tutorial step by step the external application should now be in a state where the RTM has a drawing surface inside the external application where it can display its pictures. Compile the project and run it. Be sure that the RTM is still running with the ProcSee application, Ketchup. As you should see, the ketchup process picture is automatically displayed inside the external application when the connection to the RTM is established. You can play a little bit around with it. The simulator code is still created entirely in the pTALK language. So far there are no variable updates or any interaction between the RTM and the external application. We will be coming back to this later in this tutorial.

7. Start the ProcSee RTM from the external application

This section focuses on how to start the RTM from the external application instead of connecting to an already running RTM, which we have been doing so far in this tutorial. The question is how do we do that? The solution to the question is the class PcRTMProcessBuilder. This class contains functionality to start an RTM on the host computer. In addition it manages the collection of process attributes needed by the RTM, i.e. options to pass on to the RTM, environment settings, and the RTM's current working directory. These settings must to be set programmatically before invoking the method which starts the RTM.

The class PcRTMProcessBuilder provides functionality to add new, change or remove existing environment settings. It is also possible to start an RTM with an explicit set of environment variables, i.e. the RTM does not inherit any environment settings from the external application. If you have cleared the environment settings, always remember to set the PROCSEE_DIR and the ARCH environment variables before invoking the start method.

The class PcRTMProcessBuilder also stores the options which are passed on to the RTM. It provides two overloaded add argument methods. The first add method accepts an option and an option argument, like "-r KetchupApp.pctx", where "-r" is the option, and "KetchupApp.pctx" is the option argument. The second method accept one option, like "-g". Note that this class does not check whether the options are recognized by the RTM or not. It only maintains a list of the options to pass on to the RTM. For more information about the options the RTM accepts, see the RTM executable manual page in the ProcSee Reference Manual.

There are two issues that have to be described before you can add the source code below. First, when the RTM is started it must be given a unique name. The reason is that we do not

want to get a name conflict from the control server when starting the RTM. A call to the static PcControlUtils method createUniqueName generates a unique name. This name is added to the PcRTMProcessBuilder instance with the addArgument method. Second, always set the RTM's current working directory (CWD) before invoking the start method. Normally the CWD should be set to the directory where the ProcSee application (*.pctx) file is located. In this tutorial the KetchupApp.pctx file is located in the $(PROCSEE_DIR)/demo/ketchup/$(ARCH) directory.

Add the following method to the ProcSee class.

```
private String startRTM() {
   try {
      String rtmName          = PcControlUtils.createUniqueName( "KetchupRTM" );
      String workingDirectory = System.getenv( "PROCSEE_DIR" ) +
                                "/demos/ketchup/"              +
                                System.getenv( "ARCH" );

      PcRTMProcessBuilder processBuilder = new PcRTMProcessBuilder();
      processBuilder.addArgument( "-n", rtmName );
      processBuilder.addArgument( "-r", "KetchupApp.pctx" );
      processBuilder.addEnvironment("KETCHUP_APP_VERSION", "1" );
      processBuilder.addEnvironment("KETCHUP_APP_NAME", mProcess.processName() );
      processBuilder.setWorkingDirectory( new File( workingDirectory ) );

      processBuilder.start();
      return rtmName;
   } catch (PcException ex) {
       JOptionPane.showMessageDialog( this, ex.getMessage(),
                                      "Failed to start the RTM",
                                      JOptionPane.ERROR_MESSAGE );
       System.exit( 1 );
   }

   return null;
}
```

Two environment variables were added to the process builder instance in the startRTM method. These variables are needed by the constructor function in the Ketchup application. The KETCHUP_APP_NAME is used in the ProcSee application to create and load the pdat files into the correct process. The environment variable KETCHUP_APP_VERSION is required when the ketchup application updates the simulator. Notice that the value of the environment variable KETCHUP_APP_VERSION will be modified as we proceeds in this tutorial. It is used to disable functionality in the ketchup simulator (pTALK code) that we instead implement in the external application. These environment variables will not be described any further in this tutorial.

Add the following import statements to your source code:

```
import no.hrp.procsee.api.rtm.processbuilder.PcRTMProcessBuilder;
import java.io.File;
```

Call the startRTM method from initRTM, and modify the PcRTMAsServer constructor's first argument to use the name returned from the startRTM method, like:

```
  private void initRTM() {
    try {
        String rtmName = startRTM();
        mRTM = new PcRTMAsServer( rtmName, "KetchupApp", … ) {
```

The last changes we have to do before we can test the external application is to stop the RTM when the application terminates. In the exit action listener (Exit menu item), add the following lines of code.

```
try {
        mRTM.stopRTM();
        mProcess.exit();

    } catch (PcNoConnectionException ex) {
    }
```

Now, compile and run the external application. Try to start several instances of the application. They should start without errors. That is because we have created unique names for both the external application and the RTM.

8. Create Java wrapper classes for the struct definitions

Now, it's soon time to start updating the ProcSee variables from the external application. But, first we have to create Java wrapper classes for the ProcSee struct definitions. The ProcSee struct definition resembles the "C" programming language syntax which does not fit very well into a Java world. The Java API provides wrapper classes for the primitive data types and arrays of the primitive data types, but not for the complex struct definitions used by the Ketchup demo application.

Therefore, wrapper classes must be created for the struct definitions used by the Ketchup application. The utility program pdattoapi provides this functionality. It is located in the $(PROCSEE)/bin/$(ARCH) folder. It accepts one or several ProcSee variable definition (*.pdat) files, and generates two Java files for each struct definition. The first file contains the Java wrapper class for the ProcSee struct, and the second file contains a Java wrapper class for an array of the struct (the filename has the postfix Array).

We need to explain some of the pdattoapi program arguments before running the command which generates the wrapper classes.

The pdattoapi utility program generates output for multiple target languages like Java, C# and Managed C++. To generate wrapper classes for the Java language, specify "-language Java". Note that the language argument is a required option. If this argument isn't specified the program terminates without generating any wrapper classes.

The output directory where the generated files are put is specified with the namespace option. This is a required argument as well. In this tutorial the files are put under the namespace ketchupapp.structs.

All the Java wrapper classes generated by pdatatoapi can be prefixed with a string using the prefix option, i.e. -prefix. In this tutorial we will use the three letters Pwc, which is an abbreviation for ProcSee Wrapper Class.

Now, it's time to generate the struct definitions. Open a command shell. Change the folder to the src directory under your Java project. Run the following command from the terminal window (or add it as a pre-build step in the IDE).

```
pdattoapi -language Java -prefix Pwc -namespace ketchupapp.structs
$(PROCSEE_DIR)\demos\ketchup\$(ARCH)\RecordDefinitions.pdat
```

If the command is entered exactly as in the example above it should produce the wrapper classes needed by the external application. Let's take a look at these classes in the IDE. Switch back to the IDE and expand the ketchupapp.structs package. As you can see, six different classes have been generated and added to the project. Note that you should never modify any of these classes (because they are generated). However, you can create new wrapper classes by deriving from the generated classes, and overload the PiReadScriptListener method onCreateVariable or onCreateArrayVariable. But, that is out of the scope of this tutorial. If you have made modifications to the struct definitions, always remember to rerun the pdattoapi command to generate new wrapper classes. Otherwise, it may lead to unpredictable results and heap corruption. Therefore, it is recommended that these wrapper classes are generated in a pre-build step.

The struct definitions must be registered in the ProcSee API in order to use variables of these data types in the KetchupApp application. The namespace that was used when running the tool *pdattoapi* is required to register these classes. In this tutorial is the name of the namespace: *ketchupapp.structs*. Find the initProcess() method and make a call to the PcProcess method registerStructDefinitions. The argument to this method is the namespace just described. This call will register all ProcSee struct classes generated with the tool *pdattoapi*.

```
private void initProcess() {
  try {
     …
     mProcess = new PcProcess( name, "KETCHUP_APP" );
     mProcess.registerStructDefintions( "ketchupapp.structs" );
  }
  …
}
```

The next section focuses on how to utilize these wrapper classes in the external application.

9. Create and link to variables in the ProcSee RTM

In the previous section the wrapper classes for the ProcSee struct-definitions were created. This section's main focus is on how to create and get access to the instances of these classes in the external application.

The pdat variables must be created before they can be accessed in the external application. These variables also needs to be linked to the variables in the RTM. A link means that it's automatically updated in the RTM when it's modified in the external application, and vice versa. This is the default behavior in the Java API which can be changed by pragma directives in the pdat file (#pragma). We will not describe the pragma directives any further in this document because it's out of the scope for this tutorial. For more information about the pragma directives, see the API documentation in the ProcSee Reference Manual.

The readScript method in the class PcProcess provides functionality to read the contents of the pdat file(s) and create the struct definitions and the variables in the external application. At the same time the links are created from the variables in the external application to the variables in the RTM.

So, we start modifying the onConnect method in the PiConnectionStateListener interface. In the source view, move to the initRTM method and locate the onConnect method. As described in a previous section, this method is automatically invoked by the Java API when a connection to an RTM is established. In this method, make a call to the method connectedToRTM (which we haven't implemented yet).

```
@Override public void onConnect(PcRTM rtm, PeConnect type) {
   printMessage( "Connected to rtm " + rtm.name() );
   connectedToRTM();
}
```

Some import statements have to be added before continuing with the connectedToRTM method. Put the following lines of code in your import section in the ProcSee source file.

```
import no.hrp.procsee.api.variables.PcInt32;
import no.hrp.procsee.api.variables.PcFloat;
import procseeapp.structs.PwcSTank;
import procseeapp.structs.PwcSValve;
```

In the attribute section of the class ProcSee add the following private attributes. These attributes will be needed in the connectedToRTM method and when updating the Ketchup process.

```
private PwcSTank  mTankTomatoes         = null;
private PwcSTank  mTankSpices           = null;
private PwcSTank  mTankVinegar          = null;
private PwcSTank  mTankMixing           = null;
private PwcSValve mValveTomatoes        = null;
private PwcSValve mValveSpices          = null;
private PwcSValve mValveVinegar         = null;
private PcInt32   mSimTime              = null;
private PcFloat   mTomatoesInMixingTank = null;
private PcFloat   mSpicesInMixingTank   = null;
private PcFloat   mVinegarInMixingTank  = null;
```

Now, it's time to start implementing the connectedToRTM method. First, create the filename where the pdat files are located. The ProcSee Ketchup application is put in the $(PROCSEE_DIR)/demos/ketchup/$(ARCH) folder. In the tutorial we use the System.getenv method to get the values of the environment variables. Note that no null pointer check is made

on the return value from the System.getenv method. It is assumed that the environment variables PROCSEE_DIR and ARCH are set before running this external application. The name of the struct and the variable definition files are RecordDefinitions.pdat and Variables.pdat, respectively.

The class PcProcess provides several overloaded readScript methods. In this tutorial we will be using the readScript method accepting one argument of type string array, i.e. an array of filenames. This method reads the struct and the variable definitions in the pdat files and creates instances of the default wrapper classes (both built-in and generated wrapper classes). The try-catch block surrounding the readScript methods is needed because this method throws an exception if an unrecoverable error is detected when reading the files.

The PcProcess class provides a method called variable that returns an instance of a pdat variable. It is used in the connectedToRTM method to initialize the attributes we declared in the ProcSee class. This is a generic method, returning a compile-time error if the type doesn't match the target variable. We advise you to always use the generic method because it provides compile-time checks instead of run-time checks as the non-generic method do. The method variable returns a pointer of the correct type or a null pointer if the variable doesn't match the variable it is assigned to. The first argument to this method is the return type, and the second argument is the name of the variable, which is the name of the variable in the pdat file.

Implement the following private method in the ProcSee source file.

```
private void connectedToRTM() {
   try {

      String path = System.getenv( "PROCSEE_DIR" ) + "/demos/ketchup" +
                    System.getenv( "ARCH" );

      mProcess.readScript( new String[] { path + "/RecordDefinitions.pdat",
                                          path + "/Variables.pdat" } );


      // Get the variables...
      mTankTomatoes  = mProcess.variable( PwcSTank.class, "TankTomatoes" );
      mTankSpices    = mProcess.variable( PwcSTank.class, "TankSpices"   );
      mTankVinegar   = mProcess.variable( PwcSTank.class, "TankVinegar"  );
      mTankMixing    = mProcess.variable( PwcSTank.class, "TankMixing"   );


      mValveTomatoes = mProcess.variable( PwcSValve.class, "ValveTomatoes" );
      mValveSpices   = mProcess.variable( PwcSValve.class, "ValveSpices"   );
      mValveVinegar  = mProcess.variable( PwcSValve.class, "ValveVinegar"  );

      mTomatoesInMixingTank = mProcess.variable( PcFloat.class,
                                                 "TomatoesInMixingTank" );
      mSpicesInMixingTank   = mProcess.variable( PcFloat.class,
                                                 "SpicesInMixingTank" );
      mVinegarInMixingTank  = mProcess.variable( PcFloat.class,
                                                 "VinegarInMixingTank" );

      mSimTime = mProcess.variable( PcInt32.class, "SimTime" );

   } catch (PcException ex) {
      printMessage( "connectedToRTM error " + ex.getMessage());
   }
}
```

We are now finished creating and initializing the variables needed to update the Ketchup process.

10.Update and transfer variables to the ProcSee RTM

In this section you will learn how to update and transfer variable values to the RTM. The Java API is implemented so it can run in a multi-threaded environment, but that puts some constraints when it comes to variable updates. Before a thread can start updating variables, it needs to gain ownership of a variable lock, i.e. a write lock. The variable lock is implemented so that only one thread can have ownership of a write lock, while several threads can have access to a read lock. Therefore, it is very important that a thread releases the ownership of the lock when it is no longer needed, e.g. after the variables have been updated and transferred to the RTM. If not, the external application will sooner or later end up in a deadlock situation.

Now, it's time to change the small ketchup simulator. This is a simple simulator developed in the pTALK language. We will not develop a complete simulator since that is outside the scope of this tutorial. Instead, only parts of this simulator will be modified to utilize variables updated from the external application. In the external application the following variable groups are updated, i.e. the variables controlling the tomatoes, spices, vinegar valves and

tanks, as well as different control variables, like simulator time. The remaining variables are not used by the simulator in the external application, and are therefore left as is.

The simulator runs periodically using a timer in the external application. Before implementing the simulator code you have to import the following class in the import section of your ProcSee file.

```
import javax.swing.Timer;
```

Next, define the following class attributes needed by the simulator code.

```
private Timer mTimer         = null;

private final int   mTimerInterval = 1000;
private final float mFlowFactor    = 10.0f;
```

The attribute mTimerInterval is the simulator speed, i.e. the number of milliseconds between each invocation from the timer. The interval is initially set to once per second. Later on in this tutorial you will learn how to change the simulator speed from a remote function. The mFlowFactor is just a factor used in the calculation of the fluid through the valves (pipes).

Scroll down to the initRTM method in the class ProcSee. In the PiConnectionStateListener method onConnect, add a call to the method createSimulator after the method call connectedToRTM.

```
 private void initRTM() {

    <code>
    mRTM = new PcRTMAsServer( rtmName, "Ketchup", new PaConnectionStateListener() {
       @Override public void onConnect(PcRTM rtm, PeConnect type) {
           printMessage( "Connected to rtm " + rtm.name() );
           connectedToRTM();
           createSimulator();
       }
    <code>
}
```

The createSimulator method creates the timer object which is the engine in the simulator process. Initially, the timer runs periodically with an interval of 1000 milliseconds. The call to the updateSimulator is surrounded by the static PcVariableAccess class methods beginWriteValues and endWriteAndSendValues. The update of the process variables must be surrounded by these calls. Forgetting to do so will throw exceptions in the Java API. The method beginWriteValues gains the ownership of the write lock, while endWriteAndSendValues copies the updated variable values into a transfer buffer, sends the values to the RTM and releases the write lock.

```
private void createSimulator() {

  mTimer = new Timer( mTimerInterval, new ActionListener() {

      @Override public void actionPerformed(ActionEvent e) {

          PcVariableAccess.beginWriteValues();
          try {
```

```
            updateSimulator();
        } catch ( PcException ex ) {
            printMessage( " updateSimulator: " + ex.getMessage() );
        }
        PcVariableAccess.endWriteAndSendValues();
    }
  } );

  mTimer.start();
}
```

The method updateSimulator updates the ketchup process. We will not describe in details the simulator code, as it is out of the scope for this tutorial. Just copy the contents of this method to your ProcSee class. To summarize, the source code:

- Increments the simulator time.
- Checks if any of the valves is open.
- Calculates the flow through the valves (pipes).
- Checks the contents in the tanks.
- Calculates the contents of the mixing tank.
- Closes the valves if the content of the mixing tank is full.

```
 private void updateSimulator() throws PcException {

   float   tomatoes     = 0.0f;
   float   spices       = 0.0f;
   float   vinegar      = 0.0f;
   boolean valvesClosed = true;

   mSimTime.value( mSimTime.value() + 1 );

   if (mValveTomatoes.Open() != 0) {
      valvesClosed = false;
      tomatoes     = mValveTomatoes.Flow() / mFlowFactor;
   }

   if (mValveSpices.Open() != 0) {
      valvesClosed = false;
      spices       = mValveSpices.Flow() / mFlowFactor;
   }

   if (mValveVinegar.Open() != 0) {
      valvesClosed = false;
      vinegar      = mValveVinegar.Flow() / mFlowFactor;
   }

   if ( valvesClosed ) return;

   float inT = checkTankLevel(mTankTomatoes, mValveTomatoes, tomatoes);
   float inS = checkTankLevel(mTankSpices,   mValveSpices,   spices);
   float inV = checkTankLevel(mTankVinegar,  mValveVingegar, vingegar);

   increaseLevelInMixingTank( inT, mTomatoesInMixingTank );
   increaseLevelInMixingTank( inS, mSpicesInMixingTank );
   increaseLevelInMixingTank( inV, mVinegarInMixingTank );

}
```

The checkTankLevel method performs a validation check on the tank level, i.e. whether the tank level is inside the legal limits or not. If not, the inlet valve is closed. This method is called from updateSimulator. It returns the change of the fluid through the valve at a given timestamp.

```
private float checkTankLevel( PwcSTank tank, PwcSValve valve, float in )
                  throws PcException {
  float level = tank.Level() - in;
  if ( level < 0.0f ) {
     in = tank.Level();
     tank.Level( 0.0f );
     valve.Open( 0 );
  }
  else if ( level > tank.MaxLevel() ) {
     in = tank.MaxLevel() - tank.Level();
     tank.Level( tank.MaxLevel() );
  }
  else {
     tank.Level( level );
  }
  return in;
}
```

The last function required by the ketchup process is increaseLevelInMixingTank. This function performs a check which closes the inlet valves if the level exceeds the maximum limit in the mixing tank.

```
private void increaseLevelInMixingTank( float in, PcFloat inTank )
                 throws PcException {
   if ( ( mTankMixing.Level() + in ) > mTankMixing.MaxLevel() ) {
      in = mTankMixing.MaxLevel() - mTankMixing.Level();
      inTank.value( inTank.value() + in );
      mTankMixing.Level( mTankMixing.MaxLevel() );
      mValveTomatoes.Open( 0 );
      mValveSpices.Open(   0 );
      mValveVinegar.Open(  0);
   }
   else {
      inTank.value( inTank.value() + in );
      mTankMixing.Level( mTankMixing.Level() + in );
   }
}
```

One last change has to be done to the source code before we can compile and run the external application. Move to the startRTM method and modify the environment variable KETCHUP_APP_VERSION. This time modify it to the value 2 (it was initially set to value 1). This is one of the secrets in the Ketchup application that we are not going to describe any further. Compile and run the external application again. This time, the simulator engine is run from external application.

11.Register remote functions which can be called from the ProcSee RTM

This section's focus is to learn how to register a remote function in the external application. A remote function is a function that can be called from the pTALK language in the RTM and executed in the external application. Those of you, who are familiar with the native ProcSee "C" API, know how difficult and time consuming it can be to create remote functions. It is very easy to make mistakes when unpacking the arguments. This is much easier in the Java API, with the use of annotations, which is a special form of syntactic metadata used in the

Java API that can be applied to methods and method parameters. This information is retrieved run-time when registering the remote functions. How this is done is described in detail later in this section.

The ketchup demo uses three remote functions which must be implemented in the ProcSee class. The tutorial needs to implement one method to start the simulator, one method to stop the simulator and one method to modify the simulator interval. These methods are more or less self-explanatory. This tutorial will therefore not go into the methods' implementation details. This section's focus is instead on how to register the functions and what's needed in the source code so that the Java API can recognize these methods.

So, how do we register remote functions? The registerFunctions method of the class PcRTM accepts one instance argument derived from the class Object. It uses a feature in Java called reflection which run-time examines the Object instance and detects which method to export as remote functions and which methods to ignore. The methods to export to the RTM must be annotated with the metadata attribute @PaRegisterFunction. When the Java API detects this annotation, it gathers information about the method, i.e. the type of the arguments and the name of the method. These data are then used by the Java API to create the remote function in the RTM.

The @PaRegisterFunction annotation accepts an optional argument, name. This argument provides functionality to change the remote function's default name. If it isn't specified, the default is taken from the Java method name. All the remote functions implemented in this tutorial assign a different name than the default. The names of the remote functions in the pTALK language will be StartSimulator, StopSimulator and SetSimulatorInterval, respectively.

There are some constraints when it comes to the data types allowed in remote functions. Only simple data types (int, float, double, short,…), and arrays of these data types can be used, plus the class String which represents char pointers in ProcSee. However, one class is handled specially when used as a parameter in remote functions, and that's PcRTM. This class is legal only when used as the first argument in a remote function. It isn't exported as an argument to the pTALK function, but provides information about the RTM that called the remote function. This information can be useful when an external application is connected to several RTMs.

The Java API also provides an annotation which can be applied to method parameters. The name of this annotation is @PaParam. This annotation is not used in this tutorial, but is used to specify the size of an array parameter, like:

```
@PaRegisterFunction private void test( @PaParam( size = 10 ) int[] arg ) { … }
```

Add the following import statement to your ProcSee class.

```
import no.hrp.procsee.api.rtm.PaRegisterFunction;
```

Now, it's time to add the remote functions which must be added to the PcProcSee class.

```
@PaRegisterFunction( name = "StartSimulator" )
private void startSimulator() {
   if ( !mTimer.isRunning() )
      mTimer.start();
}

@PaRegisterFunction( name = "StopSimulator" )
private void stopSimulator() {
   if ( mTimer.isRunning() )
      mTimer.stop();
}

@PaRegisterFunction( name = "SetSimulatorInterval" )
private void setSimulatorInterval( int interval ) {
   boolean isRunning = mTimer.isRunning();
   if ( isRunning )
      mTimer.stop();

   mTimer.setDelay( interval );

   if ( isRunning )
      mTimer.start();
}
```

Also implement the registerRemoteFunctions in the ProcSee class. This method makes a call to the PcRTM class registerFunctions to register all the remote functions in the ProcSee class instance (note that the this pointer is passed as argument to this method).

```
private void registerRemoteFunctions() {
   mRTM.registerFunctions( this );
}
```

Now, we have to make a call to the method registerRemoteFunctions. Move to the initRTM method in the class ProcSee. In the PiConnectionStateListener method onConnect make a call to the method registerRemoteFunctions after the call to createSimulator.

```
 private void initRTM() {

    <code>
    mRTM = new PcRTMAsServer( rtmName, "Ketchup", new PaConnectionStateListener() {
       @Override public void onConnect(PcRTM rtm, PeConnect type) {
           <code>
           createSimulator();
           registerRemoteFunctions();
       }
    <code>
}
```

As in the previous section we have to modify the environment variable KETCHUP_APP_VERSION before we can compile and run the external application. This time change the environment variable from 2 to 3 (in the startRTM method). Compile and run the external application again. This time, the start, stop and set interval buttons call the remote functions that we just added in the external application.

12.Use the execute function to run pTALK commands in the ProcSee RTM

This sections focus is on how to send commands to the RTM using the pTALK language from an external application. The Java API provides two methods that executes a pTALK command in the RTM (available in the classes PcProcess and PcRTM), these are:

- executeAsync
- execute

The method executeAsync executes a pTALK command and returns to the caller immediately. The API offers several overloaded executeAsync methods which can be used to send a pTALK command to one, several or all connected RTMs. The method execute executes a pTALK command, waits for the result, and returns the result back to the caller as a string. Notice that the execute method can be sent only to a single RTM due to the nature of synchronous method calls which returns a result.

In this section you will learn how to change the picture's resize mode from the external application. We will use the methods execute and executeAsync for this purpose. Before we start on the implementation details, we need a small introduction to the resize functionality in ProcSee. A picture in ProcSee has an attribute called resizeMode that controls how it is resized inside the window it's displayed. This attribute provides three resize modes, which are:

- normal size (this is the default resize mode and the one that we have used so far).
- resize the picture to fit inside its window.
- resize the picture to fit inside its window, but maintain the picture's width to height aspect ratio.

In this tutorial we will be developing a menu where these options are available. How this is done will be described next, but before we continue with that, we need to add a radio button group. A radio button group is needed because these menu options are mutually exclusive, i.e. only one of these items can be selected at a given point. Each of these menu items will be associated with the button group.

Now, switch to the design view in the IDE. Click on the Swing control Button Group in the palette and drag it onto the design surface. This class doesn't have a visual appearance so just drop it wherever you like. Rename this button group to mResizeButtonGroup.

We will not go into the implementation details on how to add these menu items in the external application. That is out of the scope for this tutorial. But, do the following. Add a View menu in the menu bar. In the View menu add a Resize menu. Under the Resize menu add the following three menu items (Swing class JRadioButtonMenuItem) with the following text:

- Normal Size
- Fit To Window
- Maintain Aspect Ratio

In the property window locate the buttonGroup attribute. For each of these menu items, set this attribute to the radio button group mResizeButtonGroup. Next, locate the actionCommand attribute. Set the menu items “Normal Size”, “Fit To Window” and “Maintain Aspect Ratio” equal to the values “0”, “1”, and “2”, respectively. We well later describe why.

Select the Events tab page in the properties window and press the actionPerformed event handler. Enter the name of the call-back method which will be invoked by the Swing library when the menu item is selected. Give it the name onResizeModeChanged. Use the same call-back method for the three menu items.

Now, switch to the source view and locate the call-back method onResizeModeChanged. Enter the following code in this method.

```
private void onResizeModeChanged(java.awt.event.ActionEvent evt) {
   try {
      String value = evt.getActionCommand();
      mRTM.executeAsync( "::KetchupApp.KetchupProcess",
                         "{ resizeMode = " + value + "; update(); } ");
   } catch (PcNoConnectionException ex) {
      printMessage( "Failed to change resize mode " + ex.getMessage() );
   }
}
```

The method getActionCommand in the class ActionEvent returns the action command applied to the menu items (set with the actionCommand attribute in the property window). These are the values of the different resize modes in the ProcSee RTM which can be applied to the resizeMode attribute in the picture. The first argument in the executeAsync method is the scope, i.e. KetchupApp.KetchupProcess. This is the full qualified name of the picture displayed in the exported window. The next argument to the executeAsync method is the pTALK command. This command contains the resizeMode attribute followed by a call to the update function. The update function is needed to refresh the picture after the resize mode modification.

Finally, we need to initialize the menu item with current resize mode. To get current resize mode we need to use the execute method. The scope argument is the same that we used in the onResizeModeChanged method described earlier, and the pTALK command is resizeMode. This pTALK command returns the current value of the picture’s resize mode attribute. We will not go into the implementation details here, but the source code uses the ButtonGroup to enumerate through the menu items to find the correct button in the group to select (while the other are turned off automatically). Note that we use the getActionCommand to return the menu items’ resize mode values.

```
private void setResizeMode() {
   try {
      String result = mRTM.execute( "::KetchupApp.KetchupProcess", "resizeMode");
      if ( result != null ) {
         Enumeration<AbstractButton> e = mResizeButtonGroup.getElements();
         while ( e.hasMoreElements() ) {
            JRadioButtonMenuItem rbI = (JRadioButtonMenuItem)e.nextElement();
```

```
            if ( result.compareTo( rbI.getActionCommand() ) == 0 ) {
               rbI.setSelected( true );
               break;
            }
         }
      }
  } catch (PcNoConnectionException ex) {
      printMessage( "Failed to set resize mode. " + ex.getMessage() );
  }
}
```

Locate the initRTM method and call the setResizeMode method from the PiConnectionStateListener's onConnect method. Make a call to this method after the call to registerRemoteFunctions, like:

```
private void initRTM() {
   …

      mRTM = new PcRTMAsServer( rtmName, "KetchupApp",
                                new PaConnectionStateListener() {
         @Override public void onConnect(PcRTM rtm, PeConnect type) {
             …
             registerRemoteFunctions();
             setResizeMode();
         }
   …
}
```

Now, we are finished with this tutorial. You can now compile and run the external application again. Try to change the resize modes. As you can see the picture is resized differently depending on the selected resize mode.

If you have walked through this tutorial you should now be in a position to create your own external application utilizing the Java API. This tutorial has addressed the most fundamental aspects of the Java API. We hope that you now have learned the basic concepts and the fundamental building blocks to develop your own external application using the Java API.