# Getting Started with Little-JIL
# Case Study: Measuring Stream Discharge


## Barbara Lerner
## Mount Holyoke College

## May 2010

## 1. What is Little-JIL?

Little-JIL is a coordination language. It is used to define processes that coordinate the activities of multiple people and software tools, operating in a distributed environment and sharing data and resources. It differs from a programming language in that it is not used to express actual computations. For that, we use Java. Instead, we describe how people and tools should interact, who/what is responsible for which tasks, the order in which the tasks should be performed, the data that should be passed between tasks, the resources to be shared and the actions to take when things go wrong.

This certainly all sounds vague at this point, but will become clear as we move through this tutorial. A Little-JIL process is defined in several pieces:

- A coordination diagram. This is a graphical depiction of the steps making up the process, their decomposition into substeps, and the control flow among these steps. The coordination diagram also serves as the skeleton to which we attach various annotations about other aspects of the process.
- An artifact model. This is the collection of datatypes that are used by the objects passed around during the process's execution. We will be using Java to define the datatypes.
- A resource model. This describes the resources shared by the agents during execution of the process. At least initially, we will only be using the resource model to identify the agents.
- The agents. These will be written as Java programs.

The environment in which we execute Little-JIL processes is called Juliette. It consists of a number of tools:

- Visual-JIL. This is the editor that allows us to create Little-JIL processes. It is an Eclipse plug-in.
- JSim. This is a simulator for Little-JIL. It allows us to simulate the processes before they are complete and is useful to debug as we create the process.
- Little-JIL Checker. This does some syntactic and semantic checking of the Little-JIL processes and is another plug-in to Eclipse.
- jul. This is the interpreter that is used to execute a complete process.

## 2. Stream Discharge Process

The following description of the Stream Discharge process is excerpted from A Software Engineering Approach to Scientific Data Management, written by Cori Teshera-Sterne as the final report of her independent study project in May 2010.

> The current research was most closely focused on using data processing from the stream-flow sensors as an initial example process. The data is derived from weirs installed on Big and Little Nelson Brooks, the parallel outlet streams of the Black Gum Swamp wetland (a second set of gages in the form of pipes (culverts) are installed on another stream, Bigelow Brook, above and below a beaver swamp. The

pipes and weirs operate on the same principles, but installation decisions depend on the hydrological and physical characteristics of the stream banks.) The weirs consist of a reinforced dam with a rectangular spillway and a V-shaped notch of a specified size and shape. A sensor located in the catchment area behind the weir measures water pressure, which can then be converted to flow with a linear equation relating pressure to area using the known area of the notch. At the time of this research, the data was stored in an onsite data logger and downloaded to a portable device at predetermined intervals, although a local wireless network was under construction to allow near-real-time data streaming from the data logger for processing and uploading to the internet.

Processing consists of several sequential steps for quality control and transformation of initial readings into stream discharge measurements. One of the issues associated with this system in particular, and such networks in general, is detection of and adjustment for sensor malfunction. The data is range-checked to provide a broad determination of sensor function at the level of individual values, then adjusted by a specified equation for known sensor drift. The final stream-flow data product is then calculated from this adjusted value.

Some (hopefully large) percentage of data values will undergo this process as described, but there are a number of ways in which errors can occur and be corrected. Errors such as missing data may be the result of intermittent equipment or transmission errors, or ongoing situations such as winter ice buildup. These errors may be corrected by various gap-filling or modeling approaches, such as substituting an average of several previous data values. The process for such corrections could easily be more complicated than the original. However errors are handled, it is important that it does not interfere with the subsequent processing of the remaining data.

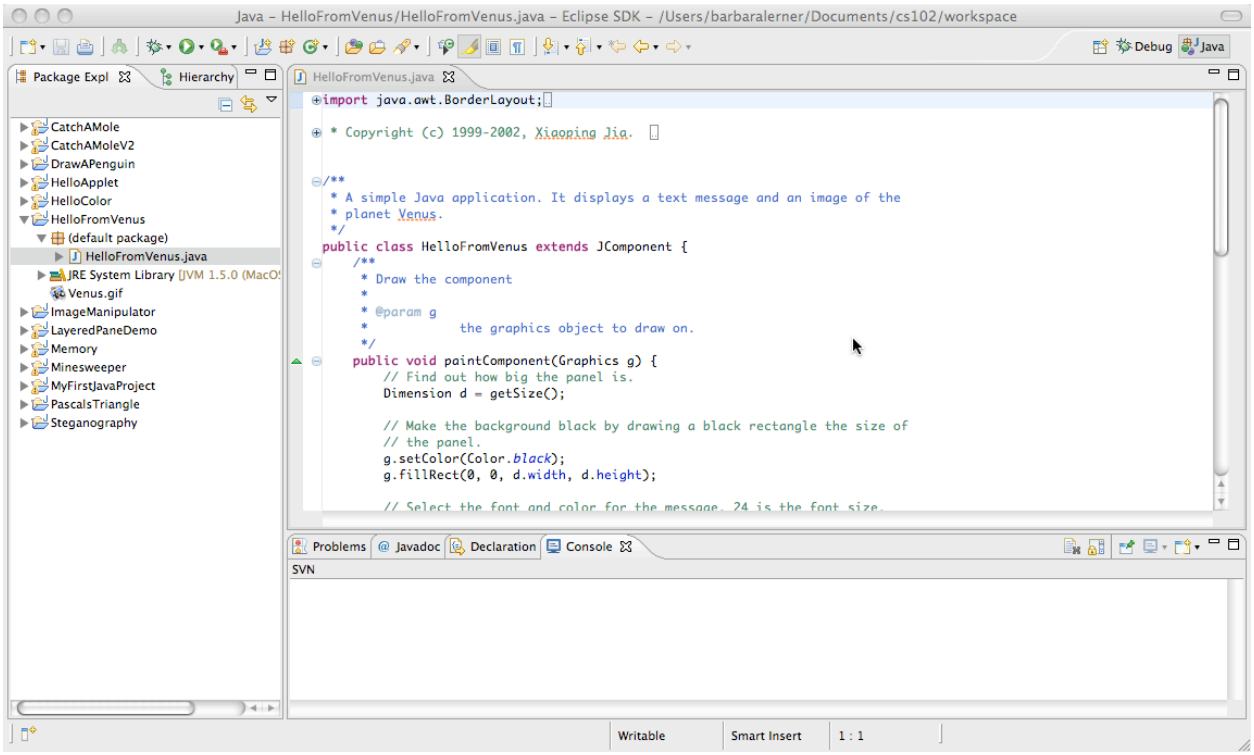
## 3. Getting Started with Eclipse

### The Basics

We will use Eclipse as the editor for this project. Eclipse is available on all major platforms: Linux, Windows and Mac. You can download Eclipse for your own personal computer for free from http://www.eclipse.org. Eclipse has become one of the most widely used software development environments in recent years for Java programming. It is built with a plug-in architecture that allows it to be extended in many ways. We will be using Eclipse not just for our Java programming but also for our Little-JIL programming.

To start Eclipse on the Macs, go to the Applications folder by clicking on the Applications icon on the left side of a Finder window. Find the folder named Eclipse and double-click on it. Then double-click on the Eclipse icon. To make it easier to start Eclipse the next time that you login, drag the Eclipse icon from the Finder window into the Dock at the bottom of the screen. You will then be able to start Eclipse in the future by clicking on the icon in the Dock.


Eclipse maintains all your code in a "workspace". The workspace contains a collection of projects, where each project typically corresponds to a single program. When you first start Eclipse, you need to select the arrow icon on the right middle side of the window that says "Go to Workbench". It will then ask you where it should create the workspace. Create a folder for the workspace, called "workspace" or "HFworkspace" or something similar.




The Eclipse window above shows the Java perspective. It consists of a package explorer panel on the left, a panel where error messages (and other informational messages) will be displayed at the bottom. The main area on the top right is where your code will appear. There is a lot of functionality to Eclipse, so it may take some time to master it, but the basics are mostly intuitive.

One feature of Eclipse is that it checks your code as you enter it. When it finds an error, an icon will appear in the left margin of the code area. The icon will either be a red X  or a red X with a light bulb  . In either case, an error message will appear in the bottom panel. If a light bulb is present and you single-click on the light bulb, Eclipse will offer some suggestions on how to correct the error. Often it has the solution in its list; you can then double-click on the solution to have Eclipse perform the necessary edits for you.

Take your time when Eclipse reports errors. First, errors often occur simply because you are in the middle of doing something and these errors go away when you complete the task. So, it is generally a good idea to ignore the warnings that come up while you are editing. Second, read the messages carefully. They often have good advice. (But not always! Computers are not actually very smart!) Third, before selecting a change to be applied automatically, make sure you understand Eclipse's suggestion. If you don't understand it, it's probably not the right thing to do! Ask for help if you are confused.

Don't try to run your program if it has any red X's.

**Some helpful commands**

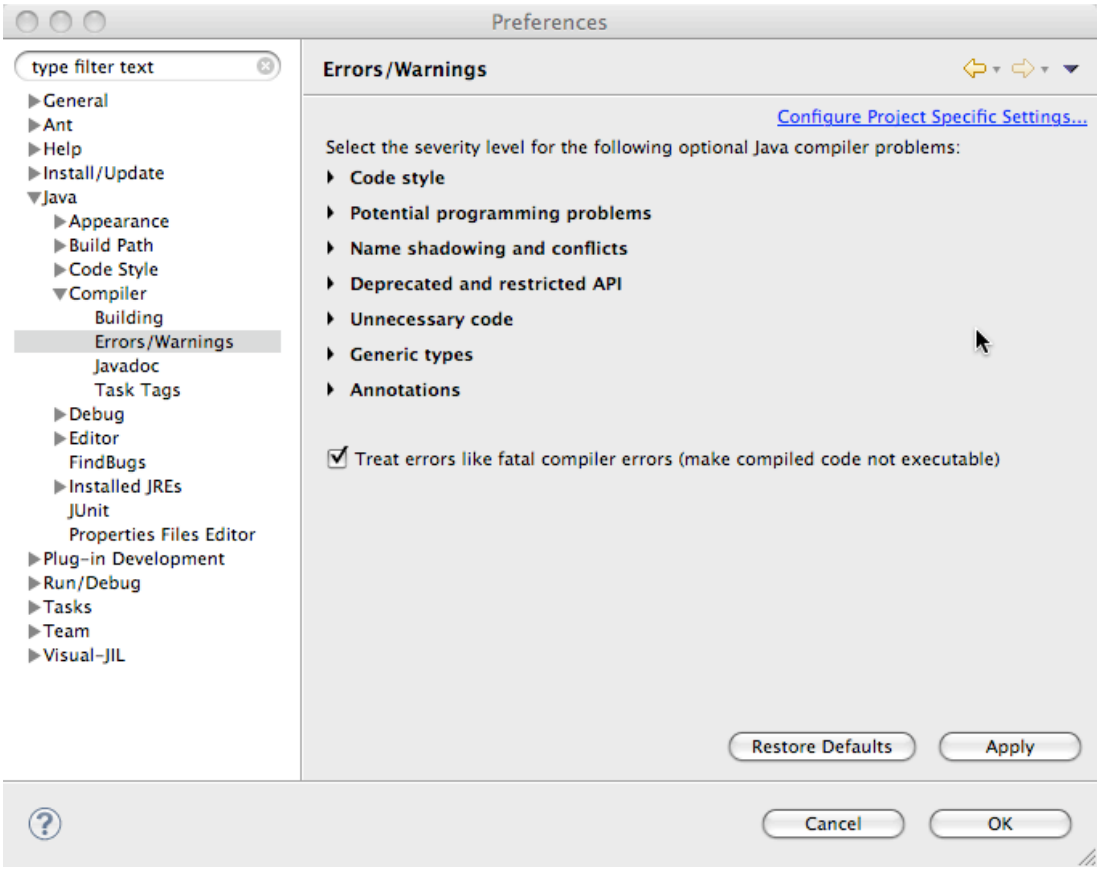
Over the course of the summer, we will investigate some of the commands in Eclipse that are very helpful in getting your programming tasks done efficiently. For now, I would like you to get used to using a few of these commands:

**Format.** The source menu contains a command called "Format". Please be sure to use this command on your Java code. It will ensure that your program is indented reasonably.

**Organize imports.** This command is also in the Source menu. It arranges your import statements alphabetically and also helps create the import statements for you so that you import individual classes (like java.awt.Color) rather than entire packages (like java.awt.*).

**Preferences**

There are also many preference settings available in Eclipse. You can leave most of these at their default values until you become comfortable and want to start exploring Eclipse's power. I do, however, suggest that you change some of the settings concerning the warning messages that Eclipse provides. By doing so, Eclipse will be able to help you out by identifying some potential errors in your code that are technically legal, but rarely good ideas. To do that select the Preferences command from the Eclipse menu. Open the Java preferences, then the Compiler preferences and select Errors/Warnings. You should have a window that looks something like this:




Open the "Potential programming problems" in the right panel. Make sure that the following are set to "Ignore":

- Serializable class without serailVersionUID
- Boxing and unboxing conversions

Make sure that the remaining choices are all set to Warning.

Next, open the "Name shadowing and conflicts" in the right panel. Make sure that "Local variable declaration hides another field or variable" is set to Warning. The remainder can be left as they are.

Now, open the "Unnecessary code" in the right panel. Make sure that the following are set Warning:

- Local variable is never read
- Unused import
- Unused local or private member
- Redundant null check
- Unnecessary cast or 'instanceof' operation

The remainder can be set to either Warning or Ignore.

## 4. Using Eclipse to create a Coordination Diagram

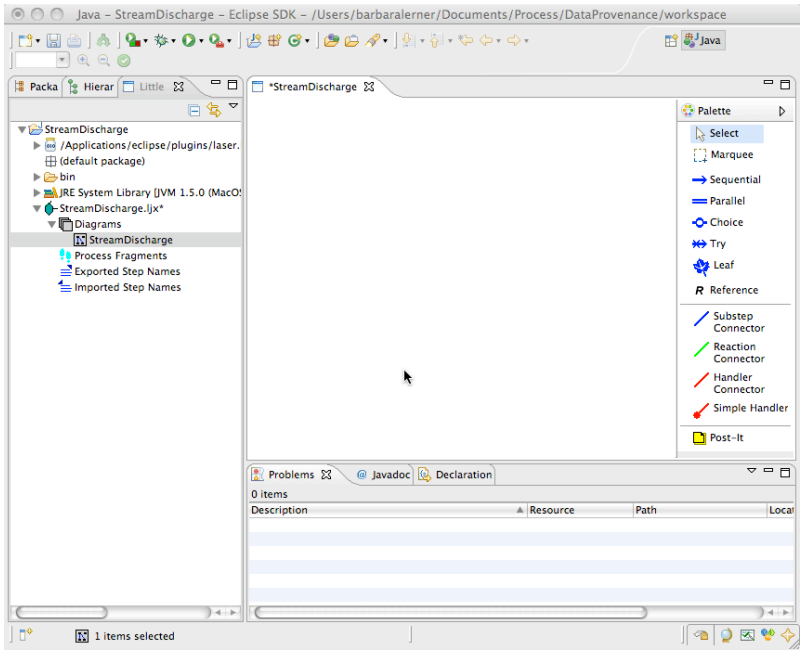
Coordination diagrams are the most visible part of Little-JIL. A Little-JIL process is defined as a collection of steps that are connected to form a step-substep hierarchy. A step is decomposed into substeps, which provide more detail about the step. In this way, a step represents an abstraction, similar to a method, while its substeps represent details, like the statements within the method body. The coordination diagram is represented graphically with each step represented as a collection of icons. Steps and substeps are connected with lines.

There is a Little-JIL tutorial on the UMass LASER wiki that you should refer to to create a Little-JIL Coordination Diagram as shown above within Eclipse. The Wiki is available at:

https://collab.cs.umass.edu/groups/laser_library/wiki/debb5/Little-JIL_Tutorial.html

Using the instructions on the Wiki, first create a Little-JIL Project named StreamDischarge. Within it, create a Module named StreamDischarge.ljx and finally a Diagram also named StreamDischarge.

At this point, Eclipse will be in the Little-JIL perspective and should look like this:



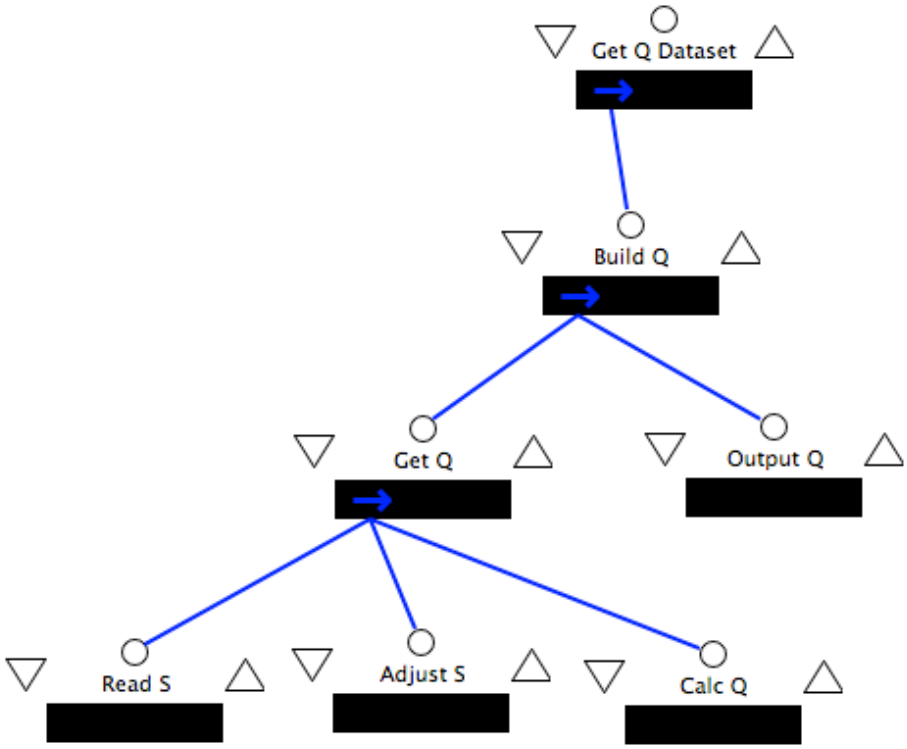
The right side of the window shows a palette that you will use to create your coordination diagram. Select and Marquee allow you to select pieces of an existing diagram so that you can move them around or delete them. Below that are 6 icons used to create different kinds of steps. In this process, we will be using just the Sequential and Leaf step kinds. A Sequential step is a non-leaf step in which the substeps are executed from left to right. A Leaf step is a step with no substeps.

In the next section, there are several types of connectors that can be placed between steps. Initially, we will be using just the Substep Connectors to create steps to their substeps.

To create a step, click on the appropriate icon in the palette (Sequential or Leaf, for our case study) and then click in the large blank drawing area to place the step. Initially, its name will be "New Step #0". Double-click on the step name to rename it.

You can connect 2 steps by selecting the Substep Connector from the palette. Click on the parent step. Then click on the substep. Eclipse will draw a line connecting the 2 steps that will stay connected as you move either or both steps around.

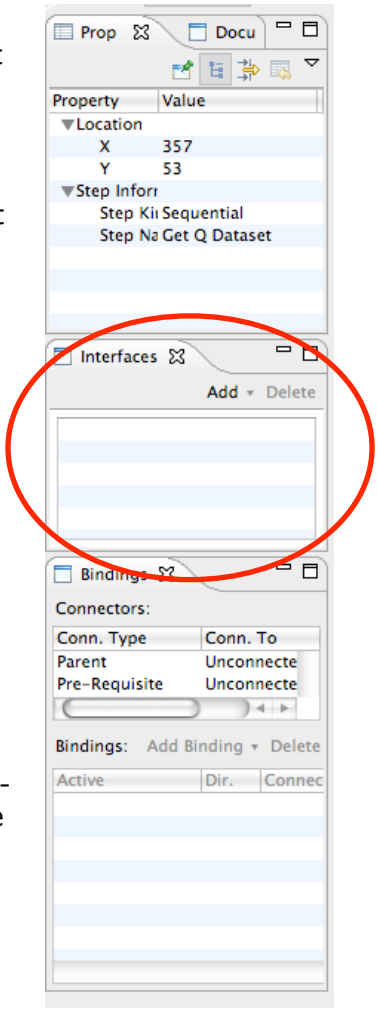
Using this knowledge, complete construction of the Little-JIL coordination diagram as shown below.




## 5. Defining Resources

Processes require resources to operate. These can be pieces of equipment, data artifacts that cannot be shared, or people who participate in the process. One special resource is the "agent". The agent is the person or software that is responsible for executing a process. Many simple processes have the agent as the only resource, so that is where we will start.

We will attach an agent to the root step. This agent will then be responsible for executing all steps within the process. To do that select the root step. On the right hand side of the Eclipse window, you should see a view named Interfaces as shown on the right. Pull down the Add menu and select "Add Resource acquisition". For parameter name say "agent" (without the quotes). For Query name, say "StreamDischargeAgent" (without the quotes).




We will use VSRM as our resource model. To define a resource model, create a new text file with the following contents:

```
StreamDischargeAgent:
streamDischargeAgent
```

This indicates that the type StreamDischargeAgent as a single instance named streamDischargeAgent. Name this file StreamDischarge.vsrm.

## 6. Simulating the Little-JIL Process

**Warning: This section is slightly out of date. In the version of jsim released in Summer 2010, the jsim file should be exported in the jul file. The command line to run jsim only needs the jul file as an argument.**

The simulator gives us a mechanism to experiment with partially-developed processes. To use the simulator we create an XML file written in the JSim Oracle Language. Such a JSim file allows us to simulate the behavior of agents as a process runs. To do this, we specify a collection of rules that define what actions to take when a step is posted and what actions to take when a step is started. For now, we will work with a very simple JSim file that simply starts steps as soon as they are posted and completes steps as soon as they are started. Details on the JSim Oracle language are available at:

http://laser.cs.umass.edu/documentation/jsim/language.html

Here we create this very basic jsim file. Create a new Text file in your Little-JIL project named StreamDischarge.jsim. Place the following in that file:

```
<simulation>
  <rules>
    <posted>
      <start>
        <fixed value="1" />
      </start>
    </posted>
    <started>
      <complete>
        <fixed value="1" />
      </complete>
    </started>
  </rules>
```

```
</simulation>
```

To run a process in the simulator, we must first create a jul file. Please follow the instructions at the URL below to create a jul file of your StreamDischarge process. Do not select Romeo as your resource model. Instead, select VSRM. You do not need to use JGrid.

https://collab.cs.umass.edu/groups/laser_library/wiki/8de48/Creating_a_JUL_file.html

Now, you are ready to run the simulator. Open a Terminal window, cd to your StreamDischarge project within your Eclipse workspace and enter the command:

```
jsim StreamDischarge.jul StreamDischarge.jsim
```

You should see the following output:

```
Get Q Dataset (0) was assigned to streamDischargeAgent at 0
streamDischargeAgent requested that Get Q Dataset (0) start at 1
streamDischargeAgent started performing Get Q Dataset (0) at 1
Build Q (1) was assigned to streamDischargeAgent at 1
streamDischargeAgent requested that Build Q (1) start at 2
streamDischargeAgent started performing Build Q (1) at 2
Get Q (2) was assigned to streamDischargeAgent at 2
streamDischargeAgent requested that Get Q (2) start at 3
streamDischargeAgent started performing Get Q (2) at 3
Read S (3) was assigned to streamDischargeAgent at 3
streamDischargeAgent requested that Read S (3) start at 4
streamDischargeAgent started performing Read S (3) at 4
streamDischargeAgent requested that Read S (3) complete at 5
Read S (3) was completed sucessfully at 5
Adjust S (4) was assigned to streamDischargeAgent at 5
streamDischargeAgent requested that Adjust S (4) start at 6
streamDischargeAgent started performing Adjust S (4) at 6
streamDischargeAgent requested that Adjust S (4) complete at 7
Adjust S (4) was completed sucessfully at 7
Calc Q (5) was assigned to streamDischargeAgent at 7
streamDischargeAgent requested that Calc Q (5) start at 8
streamDischargeAgent started performing Calc Q (5) at 8
streamDischargeAgent requested that Calc Q (5) complete at 9
Calc Q (5) was completed sucessfully at 9
streamDischargeAgent requested that Get Q (2) complete at 9
Get Q (2) was completed sucessfully at 9
Output Q (6) was assigned to streamDischargeAgent at 9
streamDischargeAgent requested that Output Q (6) start at 10
streamDischargeAgent started performing Output Q (6) at 10
streamDischargeAgent requested that Output Q (6) complete at 11
Output Q (6) was completed sucessfully at 11
streamDischargeAgent requested that Build Q (1) complete at 11
Build Q (1) was completed sucessfully at 11
streamDischargeAgent requested that Get Q Dataset (0) complete at 11
Get Q Dataset (0) was completed sucessfully at 11
```

This output shows each step being assigned to an agent and the agent starting that step. If the step is a leaf, we see the agent completing the step. If the step is a sequential step, it automatically completes when its last substep completes. The "at" numbers identify the time on the simulation timeline at which these activities took place.

## 7. Data flow

Steps and substeps can pass data between each other. Specifically, a step passes data to its substeps when the substep is posted. The substep passes output back to its parent when the substep completes. To get this to happen in Little-JIL, you need to do several things:

- Define the Java class that is the data's type.
- Declare a parameter on the steps that use that data.
- Bind parameters in the parent to parameters in the substep. (In text-based languages, this binding is inferred from the order of parameters in the method call and method declaration. In Little-JIL, the calls are not explicit and so the bindings must be explicit.)

**Declaring the Java types.** Switch to the Java perspective. Open the File menu, select New and then Source Folder. Name the Source Folder src. Next, create a new package by clicking on the package icon in the toolbar:

Set the Source folder to StreamDischarge/src and the name to streamDischarge.

Next, we will create a class. Click on the icon in the top toolbar that is a C in a green circle. Make sure the source folder is StreamDischarge/src and the package is streamDischarge. Enter SensorData for the name and click Finish. This will create the stub for a new class named SensorData.

Define the SensorData class like this:

```
package streamDischarge;

import java.io.Serializable;

public class SensorData implements Serializable{
	private static final long serialVersionUID = 1L;

}
```

Create a second class similarly defined named DischargeData.

**Declaraing the parameters.** Select the Read S step. In the Interfaces view, pull down the Add menu and select Parameter. Set the Declaration Type to Out. Use s for the parameter name and streamDischarge.SensorData for the parameter type.

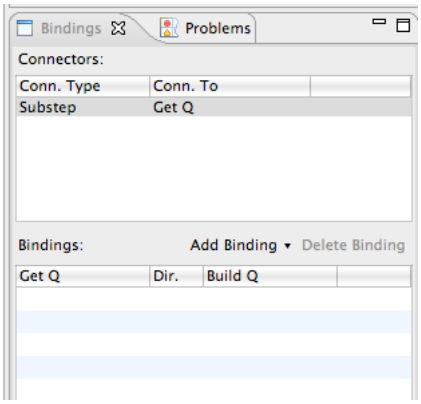
On the Adjust S step, add an in/out parameter named s whose type is streamDischarge.SensorData.

On the Calc Q step, add an in parameter named s whose type is streamDischarge.SensorData. Add an out parameter named q whose type is streamDischarge.DischargeData.

On the Get Q step, add a local parameter named s whose type is streamDischarge.SensorData. Add an out parameter named q whose type is streamDischarge.DischargeData.

On the Output Q step, add an in parameter named q whose type is streamDischarge.DischargeData.

On the Build Q step, add a local parameter named q whose type is streamDischarge.DischargeData.




**Binding the parameters.** Now, we need to bind together all of these parameters. Let's start at the top. Select the connector that goes from Build Q to Get Q. On the right hand side of the Eclipse window, below Interfaces, you should see a view labeled Bindings. When the connection between Build Q and Get Q is selected, it should look like shown at the right. Pull down the Add binding menu and select Create scope binding. Type q under Get Q, out under Dir and q under Build Q. In a similar way, create bindings for all parameters passed between steps.

Run the simulator again. It should behave exactly the same as previously, but you will see errors if the parameter bindings are not set correctly.

## 8. Little-JIL Checker

The Eclipse environment for Little-JIL includes a number of "critics". These are rules that check if your Little-JIL diagrams are valid. Since Little-JIL is an interpreted language, the critics serve the roles of syntactic and semantic checking that you would usually get from a compiler. To turn the critics on, Open the Eclipse menu and select Preferences… Then open the Visual-JIL triangle and select Critics. Select all of the critics and then OK. If any errors are found by the critics, the step name will have a squiggly line drawn under it. Hover the mouse over the step name to see the error.

At this point, all steps but the root should report "Step inherits agent from parent." This is how processes are typically defined, in fact. When an agent is attached to a step, all substeps use the same agent unless they explicitly define a new one. I would recommend that you go back into the Preferences menu and disable the critic.

If any other error messages appear, try to resolve them, or ask for help if you do not understand the message.

## 9. Exception Handling

One thing that distinguishes Little-JIL from other workflow languages is its support for exception handling. Just as with more traditional languages, like Java, exceptions are used to indicate that an operation could not be completed successfully. By throwing an exception, we can divert control to an exception handler that is defined separately from the normal control flow, making the process easier to understand. There are several steps necessary to throw and handle exceptions in Little-JIL:

- Define a Java type for the exception
- Declare that a step throws the exception
- Modify the agent code or simulator script to throw the exception

- Define a handler where control should go when the exception occurs and attach the handler to the correct place in the process
- Determine where control should flow after handling the exception
- Propagate the exception from the step that throws it to the step to which the handler is attached.

These steps are now described in more detail.

**Define the Java type for the exception.** Any Java type that implements Serializable can be thrown. It is not necessary for the type to extend Exception. Add the following Java class in the streamDischarge package:

```
package streamDischarge;

import java.io.Serializable;

/**
 * Exception class for out of range values passed in from original data
 */
public class OutOfRangeException implements Serializable {

	private static final long serialVersionUID = 1L;
	private SensorData sData;

	/**
	 * Constructor for OutOfRange exception object
	 *
	 * @param s the SensorData object whose value is out of range
	 */
	public OutOfRangeException (SensorData s){
		sData = s;
	}

	public OutOfRangeException (){
	}

	public SensorData getData(){
		return sData;
	}

}
```

**Declare that a step throws the exception.** Exceptions can be thrown by leaf steps or by prerequisites or postrequisites attached to any step. The purpose of a prerequisite is to check that the input parameters being passed to a step are valid. The purpose of a postrequisite is to check that the output being produced by a step is valid. If a prerequisite or postrequisite detects a problem, it throws an exception. Similarly, the agent responsible for a leaf step can throw an exception directly if it encounters errors during its execution. For this process, we will define a postrequisite on the Read S step. We will call this postrequisite Check S. It will throw an exception if it determines that the value is out of

range, indicating that either the sensor failed or the data got corrupted in some way between the sensor and when it was read by the process.

To define a postrequisite, we first define a leaf step and name it Check S. With Check S selected, go to the Interfaces view and pull down the Add menu. Select Add Message/ Exception. In the window that comes up, select Exception and enter streamDischarge.OutOfRangeException as the parameter type.

Next select the Read S step and go to the Bindings view in the right column. Select Post-Requisite. Then click on the … to the right of Unconnected. Select Step and enter Check S as the step.name. Click the Select button next to the Failure exception type field and enter streamDischarge.OutOfRangeException as the exception name. Then click OK. The triangle to the right of Read S in the main process window should turn red.

**Modify the simulator script to throw the exception.** Add the following to your jsim file. Place this after the Posted rule and before the Started rule. JSim rules are selected based on the order in which they appear in the file. This rule will be selected when the Check S starts, while the existing started rule serves as a default that is used for all other steps. This rule uses a JSim script to define the steps behavior. Each time that Check S is started, the next entry in the script is used to control its behavior. The rule shown here indicates that Check S was complete successfully the first time it is started, but will terminate, throwing the OutOfRangeException the second time. After that, all the actions in the script will be used up, so the Check S step will then use the default rule.

```
<step name="Check S">
   <started>
      <script>
         <group>
            <complete>
               <fixed value="1" />
            </complete>
         </group>
         <group>
            <terminate>
               <exception type="streamDischarge.OutOfRangeException" />
               <fixed value="1" />
            </terminate>
         </group>
      </script>
   </started>
</step>
```

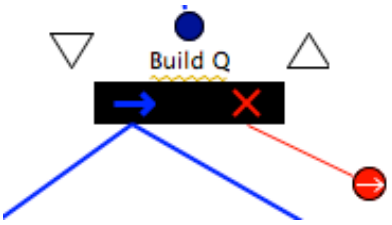
For this script to work, we need to modify the process so that it reads in more than one S value. To do this, we will change the "cardinality" on the edge from Get Q Dataset to Build Q to 2. This means that the Build Q step will loop. For our purposes, it is sufficient for it to loop twice, so we will set the cardinality to 2.

To do this, select the connector between Get Q Dataset and Build Q. In the Properties view at the top right, select Cardinality. Click on the … that appears. Click other for both the lower bound and upper bound and add 2 for each value. This will cause Build Q (and all its substeps) to be repeated exactly twice.

**Define a handler where control should go when the exception occurs and attach the handler to the correct place in the process.** When we encounter a bad S value, it

will not be possible for us to calculate a Q value. As a result, we will want to skip the steps Adjust S, Calc Q and Output Q. Therefore, we will attach our handler to the Build Q step. This handler will simply complete this execution of Build Q to allow the process to continue.

To do this, select Simple Handler from the Little-JIL palette. Click in the Build Q step and then click a second time down and to the right of the step. This should give you something like the figure at the right.




Now select the red circle with arrow. In the Properties view, select Exception Template and click on the … Select streamDischarge.OutOfRangeException for the type and select OK.

**Determine where control should flow after handling the exception.** In this case, we want to skip the rest of the substeps in Build Q, so we select "complete" for the control flow. To do that, select Continuation Action in the Properties view. Pull down the menu and select Complete.

**Propagate the exception from the step that throws it to the step to which the handler is attached.** You may have noticed that the critics in Little-JIL have placed red squiggly lines under 2 of your steps. If you hover over the Read S step, you should see "Exception streamDischarge.OutOfRangeException should be added to the interface". If you hover over Build Q, you should see "Simple handler for streamDischarge.OutOfRangeException is redundant. Both of these messages are related to the exception handling that we are working on.

Little-JIL requires that every ancestor of a step that throws an exception up to the step where it is handled declare that it throws the exception. That explains the error on the Read S step. Using this information, Little-JIL can then know if handlers are in reasonable places. The error on Build Q indicates that we have a handler for an exception that Little-JIL does not expect to be thrown. We will fix both of these problems by propagating the error from Read S to Get Q and from Get Q to Build Q.

Click on the Read S step. In the Interfaces view, pull down the Add menu and select Add Message/Exception. Select Exception. When you click the Choose button, OutOfRangeException should be listed. Select it. Repeat this at the Get Q step. Now, you should see no more red squiggly lines.

Export this into a Jul file and run it through the simulator again. The output should look similar to earlier, except that the Check S step is also executed, the Build Q step and its substeps get executed twice and on the second execution of Check S, it throws an exception. The most significant differences from last time begin at time 13. Notice how the exception propagates from Check S to its handler and where control flows after it is handled.

```
Get Q Dataset (0) was assigned to streamDischargeAgent at 0
streamDischargeAgent requested that Get Q Dataset (0) start at 1
streamDischargeAgent started performing Get Q Dataset (0) at 1
Build Q (1) was assigned to streamDischargeAgent at 1
streamDischargeAgent requested that Build Q (1) start at 2
streamDischargeAgent started performing Build Q (1) at 2
Get Q (2) was assigned to streamDischargeAgent at 2
streamDischargeAgent requested that Get Q (2) start at 3
streamDischargeAgent started performing Get Q (2) at 3
Read S (3) was assigned to streamDischargeAgent at 3
```

```
streamDischargeAgent requested that Read S (3) start at 4
streamDischargeAgent started performing Read S (3) at 4
streamDischargeAgent requested that Read S (3) complete at 5
Check S (4) was assigned to streamDischargeAgent at 5
streamDischargeAgent requested that Check S (4) start at 6
streamDischargeAgent started performing Check S (4) at 6
streamDischargeAgent requested that Check S (4) complete at 7
Check S (4) was completed sucessfully at 7
Read S (3) was completed sucessfully at 7
Adjust S (5) was assigned to streamDischargeAgent at 7
streamDischargeAgent requested that Adjust S (5) start at 8
streamDischargeAgent started performing Adjust S (5) at 8
streamDischargeAgent requested that Adjust S (5) complete at 9
Adjust S (5) was completed sucessfully at 9
Calc Q (6) was assigned to streamDischargeAgent at 9
streamDischargeAgent requested that Calc Q (6) start at 10
streamDischargeAgent started performing Calc Q (6) at 10
streamDischargeAgent requested that Calc Q (6) complete at 11
Calc Q (6) was completed sucessfully at 11
streamDischargeAgent requested that Get Q (2) complete at 11
Get Q (2) was completed sucessfully at 11
Output Q (7) was assigned to streamDischargeAgent at 11
streamDischargeAgent requested that Output Q (7) start at 12
streamDischargeAgent started performing Output Q (7) at 12
streamDischargeAgent requested that Output Q (7) complete at 13
Output Q (7) was completed sucessfully at 13
streamDischargeAgent requested that Build Q (1) complete at 13
Build Q (1) was completed sucessfully at 13
Build Q (8) was assigned to streamDischargeAgent at 13
streamDischargeAgent requested that Build Q (8) start at 14
streamDischargeAgent started performing Build Q (8) at 14
Get Q (9) was assigned to streamDischargeAgent at 14
streamDischargeAgent requested that Get Q (9) start at 15
streamDischargeAgent started performing Get Q (9) at 15
Read S (10) was assigned to streamDischargeAgent at 15
streamDischargeAgent requested that Read S (10) start at 16
streamDischargeAgent started performing Read S (10) at 16
streamDischargeAgent requested that Read S (10) complete at 17
Check S (11) was assigned to streamDischargeAgent at 17
streamDischargeAgent requested that Check S (11) start at 18
streamDischargeAgent started performing Check S (11) at 18
streamDischargeAgent requested that Check S (11) terminate at 19 because
OutOfRangeException
Check S (11) was terminated at 19 because OutOfRangeException
streamDischargeAgent requested that Read S (10) terminate at 19 because
OutOfRangeException
```

```
Read S (10) was terminated at 19 because OutOfRangeException
streamDischargeAgent requested that Get Q (9) terminate at 19 because
OutOfRangeException
Get Q (9) was terminated at 19 because OutOfRangeException
streamDischargeAgent requested that Build Q (8) complete at 19
Build Q (8) was completed sucessfully at 19
streamDischargeAgent requested that Get Q Dataset (0) complete at 19
Get Q Dataset (0) was completed sucessfully at 19
```

## 10.Defining Dummy Agents

Up until this point, we have not executed processes, but simulated them. In particular, the simulation scripts hardwire in the behavior of agents, which might otherwise be people or software that incorporate non-trivial decision making in their actions. You may also have noticed that we are not actually manipulating any data thus far. Simulation is a good place for us to start because we can test that our process is reasonably well-defined without adding the complexity of agents to it. We are now about to prepare for the next step, incorporating agents and using the interpreter.

We will start with simple "dummy" agents. These agents will do the same thing as our initial simulation script did, starting a step as soon as it is posted and completing it as soon as it is started. We use these agents in our first step of working with the interpreter.

Juliette provides an implementation of agent with just this behavior. The implementation is in laser.juliette.agent.DummyAgent. To start a dummy agent, we will use the following command:

```
jul agent localhost laser.juliette.agent.DummyAgent streamDischargeAgent &
```

localhost tells Juliette to run the agent on your own computer. streamDischargeAgent is the instance name that we used in the vsrm file.

## 11.Running the Interpreter

The following commands are used to run our StreamDischarge process in the interpreter with a dummy agent:

```
jul start
jul agent localhost laser.juliette.agent.DummyAgent streamDischargeAgent &
jul install StreamDischarge.jul
jul run StreamDischarge
```

If everything goes well, you should see output similar to this:

```
-> jul start
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
-> jul agent localhost laser.juliette.agent.DummyAgent streamDischargeAgent &
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
-> jul install StreamDischarge.jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
-> jul run StreamDischarge
```

```
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
Process COMPLETED sucessfully.
```

## 12.Defining Agents with Behavior

To have real processes, we need agents that do more than the dummy agent. You will need to add juliette-client.jar to your project's build path for your agent code to compile. To do this, right-click on your project and select Build Path and then Configure Build Path. In the window that comes up, select Libraries. Then click on the Add External JARs button. Browse to the folder where juld is installed. Go into lib and select juliette-client.jar.

Please refer to [https://collab.cs.umass.edu/groups/laser_library/wiki/2cee6/Writing_an_agent.html](https://collab.cs.umass.edu/groups/laser_library/wiki/2cee6/Writing_an_agent.html) for information on how to define an agent.

Here is the implementation of our StreamDischargeAgent:

```
package streamDischarge;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;

import laser.juliette.agent.AbstractAgent;
import laser.juliette.agent.ItemHandler;
import laser.juliette.agent.ItemHandlerFactory;
import laser.juliette.agent.Launcher;
import laser.juliette.ams.AgendaItem;

/**
 * @author CTS Agent for reading in file 7/9/09 Copied from earlier CalcQAgent
 *         7/13 Modified to open stream
 */

public class StreamDischargeAgent extends AbstractAgent {

	@Override
	/**
	 * Initializes the agent responsible for reading S values from the file.
	 * This agent should be started by saying:
	 * jul agent localhost streamDischarge.StreamDischargeAgent
streamDischargeAgent <csv file> &
	 * where the name of a csv file containing S data should be provided instead of
	 * <csv file>.
	 */
	protected void configureAgent() {

		// Find out the name of the file to read data from
		String[] args = Launcher.getArguments();
		File file = new File(args[0]);
		System.out.println("File: " + file.toString());

		// Check that the file exists
		if (file.exists()) {
```

```
                        System.out.println("File " + file.toString() + " found");

                        try {
                               // Open the file
                               final BufferedReader in =
                                      new BufferedReader(new FileReader(file));

                               // Identify the step handler for the steps assigned to this
                               // agent.
                               setItemHandlerFactory(new ItemHandlerFactory() {
                                      public ItemHandler createItemHandler(
                                                    AgendaItem associatedItem) {
                                             return new
                                                StreamDischargeHandler(associatedItem, in);
                                      }
                               });

                               // Let us handle both new steps that are posted and ones
                               // that existed before the agent was started.
                               setProcessingMode(ProcessingMode.EXISTING_AND_NEW);

                        } catch (FileNotFoundException e) {
                               // This shouldn't happen since we checked if the file
                               // existed before trying to open it.
                               System.out.println(
                                      "File " + file.toString() + " not found");
                        }

                 }
                 else {
                        // Report an error if we can't open the file.
                        System.out.println("File " + file.toString() + " not found");

                 }
          }
}
```

A handler is the object that actually performs the work of a step. Here is a handler that corresponds with the agent above. The only step for which it does actual work is Read S. For all other steps, it simply completes.

When the handler reaches the end of the file, it signals this to the process by throwing the EndOfSetException. We should therefore modify our Little-JIL process to expect this exception and handle it. In this case, the exception is thrown by the Read S step. Add a declaration of EndOfSetException to the Read S step. Propagate it all the way to the root, where you should add a complete handler. You should also define an EndOfSetException Java class similar to our earlier exception.

Now that we have this way of detecting the end of the input, we should change the cardinality on the connector between Get Q Dataset and Build Q so that it loops indefinitely. To do this, select the connector. Change the cardinality to have a lower bound of Default and an upper bound of Unlimited. The "2" on the edge should be replaced with a "+", meaning 1 or more executions of the substep are expected.

Here is the code for StreamDischargeHandler:

```java
package streamDischarge;

import java.io.BufferedReader;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;

import laser.juliette.agent.StartHandler;
import laser.juliette.ams.AMSException;
import laser.juliette.ams.AgendaItem;
import laser.juliette.ams.IllegalTransition;
import laser.lj.Step;

/**
 * @author CTS 7/9
 */

public class StreamDischargeHandler extends StartHandler{
	// The stream to read data from
	private BufferedReader reader;

	/**
	 * Construct a handler for a step in the stream discharge process
	 * @param associatedItem the agenda item associated with the step
	 * @param in the stream to read data from
	 */
	public StreamDischargeHandler(AgendaItem associatedItem, BufferedReader in) {
		super(associatedItem);
		reader = in;
	}

	@Override
	/**
	 * Called when a step assigned to this agent is started.  The only step that
	 * actually does any work currently is the Read S step.
	 */
	public void started(){

		try {

			Step step = item.getStep();
			// testing: print to console
			System.out.println();
			System.out.println();
			System.out.println("Step: " + step.getName());

			if (step.getName()equals("Read S")){
				readS();

			} else if (step.isLeaf()) {
				item.complete();
			}

		} catch (AMSException e) {
```

```
                e.printStackTrace();
            } catch (IllegalTransition e) {
                e.printStackTrace();
            }

        }

        /**
         * Reads the next value from the input file and sends it back into the process
         * as the s parameter.  It terminates when it reaches the end of the input or
         * if any exception is found during its processing of the input.
         * When this happens, it throws EndOfSetException into the process.
         * @throws AMSException if it can't communicate with its agenda
         * @throws IllegalTransition if the step cannot be completed or terminated
         */
        private void readS() throws AMSException, IllegalTransition {
            try {
                // Read in a line of data
                System.out.print("(ReadSHandler) Assigning SensorData object: ");
                String line;

                // Read in next line of file
                line = reader.readLine();
                if (line == null  || line.equals("")){
                    terminate();
                    return;
                }

                SensorData data = new SensorData (line);
                item.setParameter("s", data);
                item.complete();

            } catch (Exception e) {
                e.printStackTrace();
                terminate();
            }
        }

        /**
         * Terminate the step, throwing the EndOfSetException into the process
         * @throws AMSException if it can't communicate with the agenda
         * @throws IllegalTransition if it can't terminate the step
         */
        private void terminate()
                throws AMSException, IllegalTransition {
            Set<Serializable> exceptions = new HashSet<Serializable>();
            exceptions.add (new EndOfSetException());
            item.terminate(exceptions);
            return;
        }

}
```

This agent code also requires a new SensorData constructor that takes a String parameter and parses the String to create a SensorData object. Here is the code for that constructor:

```
/**
 * Parses a line containing a Gregorian date, a Julian date and a data
 * value. The Gregorian date is ignored.  The 3 data items are separated by
 * commas. Sample line:
 * 2007-12-31T23:45,365.9896,8.69
 *
 * @param line the line of data
 * @return a SensorData object containing the information from the line
 * @throws NumberFormatException
 *                      if the data is not parseable as doubles.
 */
public SensorData (String line) throws NumberFormatException {
      // Split the lines into tokens using a comma as the separator.
      String[] tokens = line.split(",");
      System.out.println("(Parser) Current line:" + line);

      // throws NumberFormatException if it is not a number.
      // Parse the Julian Date, currently as a double
      date = Double.parseDouble(tokens[1]);

      // throws NumberFormatException if it is not a number.
      value = Double.parseDouble(tokens[2]);
}
```

Stop the interpreter using "jul stop". Start it up again, this time using your StreamDischargeAgent. The agent needs to be run out of a jar file. It turns out that jul files are actually jar files, so the simplest way to do this is to create a symbolic link from your jul file to a similarly named jar file. Do this from the Terminal:

-> ln - s StreamDischarge.jul StreamDischarge.jar

To run this process, you will also need a data file. Create a text file with this as its contents:

2007-12-21T00:15,355.0104,4.79
2007-12-21T02:30,355.1042,62
2007-12-21T05:00,355.2083,4.77
2007-12-21T10:00,355.4167,4.75

This is a comma-separated file in the format that SensorData is expecting. Call this file SensorData.csv. Then, when you start your agent, you need to give this as a command line argument. Here is what you should enter and what you should expect to see as output when you run the process. You should see that it reads in 4 data values and then terminates with EndOfSetException.

```
-> jul start
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
-> jul agent localhost StreamDischarge.jar streamDischarge.StreamDischargeAgent
streamDischargeAgent SensorData.csv &
[6] 2372
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
```

```
-> jul install StreamDischarge.jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin
-> jul run StreamDischarge
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin/jul
/Users/barbaralerner/Documents/LJilJarFiles/juld/bin


Step: Get Q Dataset


Step: Build Q


Step: Get Q


Step: Read S
(ReadSHandler) Assigning SensorData object: (Parser) Current line:
2007-12-21T00:15,355.0104,4.79


Step: Check S


Step: Adjust S


Step: Calc Q


Step: Output Q


Step: Build Q


Step: Get Q


Step: Read S
(ReadSHandler) Assigning SensorData object: (Parser) Current line:
2007-12-21T02:30,355.1042,62


Step: Check S


Step: Adjust S


Step: Calc Q
```

Step: Output Q

Step: Build Q

Step: Get Q

Step: Read S
(ReadSHandler) Assigning SensorData object: (Parser) Current line:
2007-12-21T05:00,355.2083,4.77

Step: Check S

Step: Adjust S

Step: Calc Q

Step: Output Q

Step: Build Q

Step: Get Q

Step: Read S
(ReadSHandler) Assigning SensorData object: (Parser) Current line:
2007-12-21T10:00,355.4167,4.75

Step: Check S

Step: Adjust S

Step: Calc Q

Step: Output Q

Step: Build Q

Step: Get Q

```
Step: Read S
(ReadSHandler) Assigning SensorData object: Process COMPLETED successfully.
```

Now, you should try extending this process by providing the agent code for the postrequisite and the other leaf steps. The structure for the agent should be similar to what you already have. Here is a description of what should happen for the other steps:

- Check S: Complete if the sensor data value is between -60 and 60. Throw OutOfRangeException otherwise.
- Adjust S: Change the sensor value to (1.0936 * input sensor value) + 0.5505
- Calc Q: Calculate the DischargeData as
  - If sensor value == 0, set discharge value to 0
  - If sensor value <= 25.4, set discharge value to 0.02391 * sensor value$^{2.5}$
  - Otherwise, set discharge value to 77.74 + 7.0001 * (sensor value - 25.4)$^{1.5}$
- Output Q: Write the discharge data to a file whose name is provided to the agent as a 2$^{nd}$ parameter, after the input file name.

Congratualtions! You have written and executed a Little-JIL process!

## 13.For More Information

There is much more that one could learn about Little-JIL. The most complete reference is the Little-JIL Language manual, available at http://laser.cs.umass.edu/techreports/06-51.pdf. It is not necessary to understand everything in there, especially initially. The features of Little-JIL that did not get used in the tutorial but that you will most likely depend on the most are the various step kinds and the different types of continuation semantics associated with exception handlers. Take a look at those so that you have a sense of what is available. Mostly you will learn things as you need them.

You should also be aware of the Little-JIL wiki, which I have referred to a few times in this tutorial. It is far from complete, but it may have useful tips and is probably something we should contribute to as we gain experience with Little-JIL. It is available at https://collab.cs.umass.edu/groups/laser_library/wiki/debb5/Little-JIL_Tutorial.html.