S. Kirstein, S. Land, D. Halfkann

RapidMiner 7

How to extend RapidMiner

rapidminer

Sabrina Kirstein, Sebastian Land, Dominik Halfkann

RapidMiner 7

How to extend RapidMiner

January 25, 2016

RapidMiner
www.rapidminer.com

Contents

CHAPTER 1

Introduction

You have probably already installed RapidMiner Studio and played around with the enormous set of operators.

And maybe in that learning, even with the huge number of functions provided by RapidMiner, you have stumbled on a problem that is unsolvable or only solvable with what seems to be a too-complex process. Don't despair! You can build your own extension to RapidMiner, providing new operators and new data objects and still have all the functionality of RapidMiner.

To create your own extension, it is best to go through the step-by-step guide:

1. Set up your development environment
2. Individualize the extension template
3. Create your own operator

4. Use input and output ports
5. Use operator parameters
6. Document the behavior of your operator
7. Finally, publish your extension on the Marketplace

To create more advanced enhancements to RapidMiner, the Advanced Enhancements section describes how to:

- Build a super operator
- Create your own data objects
- Make custom configurators
- Create custom graphical elements and individualize the **Preferences** dialog

CHAPTER 2

Setting Up The Environment

There is this great IDE (integrated development environment) vs. text editor debate when it comes to programming.

There are two types of programmers; those who prefer full featured IDE's and those who prefer a lightweight text editor. You can use your favorite when developing your extension.

- If you decide to develop your extension with the IDE **Eclipse**, click here.
- If you want to use your favorite **text editor**, click here.

Don't know which one to choose? Try Eclipse and see how it works for you.

2.1 Using Eclipse

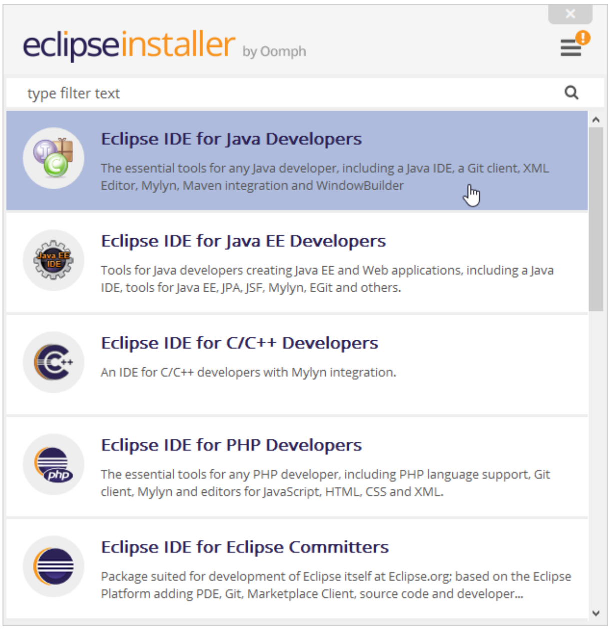
2.1.1 Installing Java

The first step of setting up your development environment is installing **Java**, if you haven't already. You can download Java here.

2.1.2 Downloading and installing Eclipse

To work with Eclipse:

1. Go to the Eclipse website and download the Eclipse installer for your operating system.
2. Execute the installer and select the package `Eclipse IDE for Java Developer`.



Eclipse IDE for Java Developers
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Mylyn, Maven integration and WindowBuilder

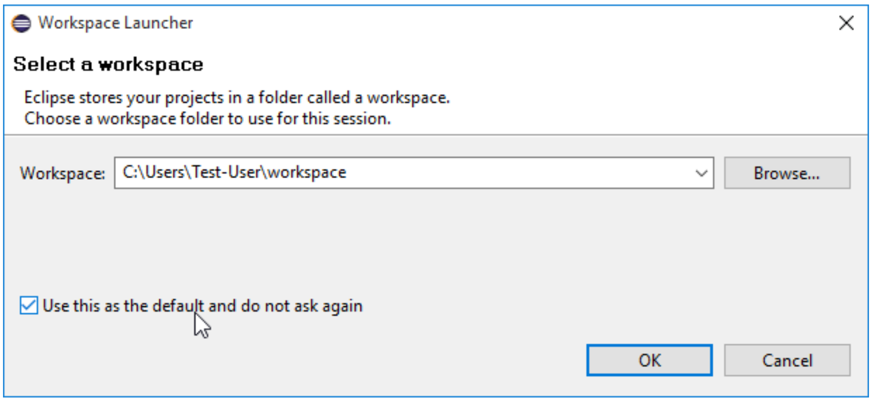
Eclipse IDE for Java EE Developers
Tools for Java developers creating Java EE and Web applications, including a Java IDE, tools for Java EE, JPA, JSF, Mylyn, EGit and others.

Eclipse IDE for C/C++ Developers
An IDE for C/C++ developers with Mylyn integration.

Eclipse IDE for PHP Developers
The essential tools for any PHP developer, including PHP language support, Git client, Mylyn and editors for JavaScript, HTML, CSS and XML.

Eclipse IDE for Eclipse Committers
Package suited for development of Eclipse itself at Eclipse.org; based on the Eclipse Platform adding PDE, Git, Marketplace Client, source code and developer...

3. Select the installation folder and click on `Install`.
4. Click `Launch` to start Eclipse.
5. When Eclipse starts, a window pops up prompting you to select a workspace. Select the folder for storing your Eclipse projects. Click on the checkbox to keep this setting each time you open Eclipse.


Workspace Launcher

Select a workspace

Eclipse stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Users\Test-User\workspace

[✓] Use this as the default and do not ask again

You should now see the Eclipse **Welcome** screen (see Figure 2.1). If you have never used Eclipse before, try the tutorials and play around with Eclipse before proceeding. When you are ready to proceed, switch to the workbench (click the arrow in the upper right corner).

Eclipse tricks you should know:

1. To display a list of all keyboard shortcuts, press Ctrl+Shift+L.
2. Press Ctrl+Space in the Java editor to get a list of suggested completions. Typing some characters before pressing Ctrl+Space will shorten the list.
3. Code completion supports camel case patterns (for example, entering 'NPE' and pressing Ctrl+Space will propose `NullPointerException`).
4. Select an opening or closing bracket and press Ctrl+Shift+P to find its matching bracket.

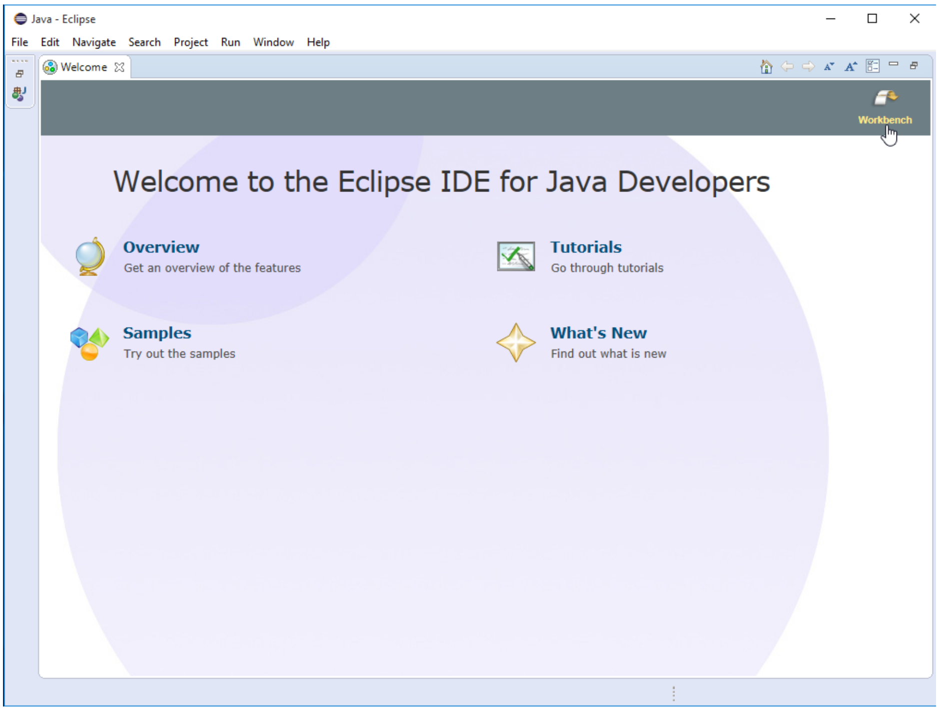
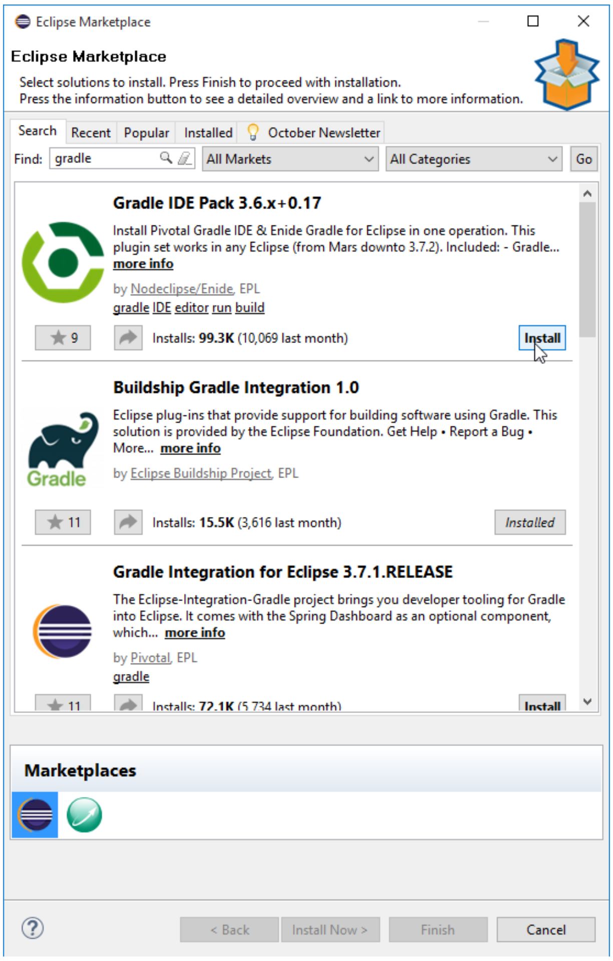

Java - Eclipse
File Edit Navigate Search Project Run Window Help
Welcome
Workbench
Welcome to the Eclipse IDE for Java Developers
Overview
Get an overview of the features
Tutorials
Go through tutorials
Samples
Try out the samples
What's New
Find out what is new

Figure 2.1: The Eclipse start screen

5. Type `/**` and press enter to automatically add a JavaDoc comment stub.
6. Press Ctrl+Shift+O to organize all imports automatically.
7. Press Ctrl+1 to show possible fixes for a problem or possible actions.

2.1.3 Installing the Gradle plugin

Next, install the Gradle plugin for Eclipse. Open the Eclipse **Help** menu and click on **Eclipse Marketplace...**. Search for `Gradle IDE Pack` and install the plugin. You then need to restart Eclipse.



Eclipse Marketplace

Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Search | Recent | Popular | Installed | October Newsletter

Find: gradle | All Markets | All Categories | Go

Gradle IDE Pack 3.6.x+0.17

Install Pivotal Gradle IDE & Enide Gradle for Eclipse in one operation. This plugin set works in any Eclipse (from Mars downto 3.7.2). Included: - Gradle... more info

by Nodeclipse/Enide, EPL

gradle IDE editor run build

9 | Installs: 99.3K (10,069 last month) | Install

Buildship Gradle Integration 1.0

Eclipse plug-ins that provide support for building software using Gradle. This solution is provided by the Eclipse Foundation. Get Help • Report a Bug • More... more info

by Eclipse Buildship Project, EPL

11 | Installs: 15.5K (3,616 last month) | Installed

Gradle Integration for Eclipse 3.7.1.RELEASE

The Eclipse-Integration-Gradle project brings you developer tooling for Gradle into Eclipse. It comes with the Spring Dashboard as an optional component, which... more info

by Pivotal, EPL

gradle

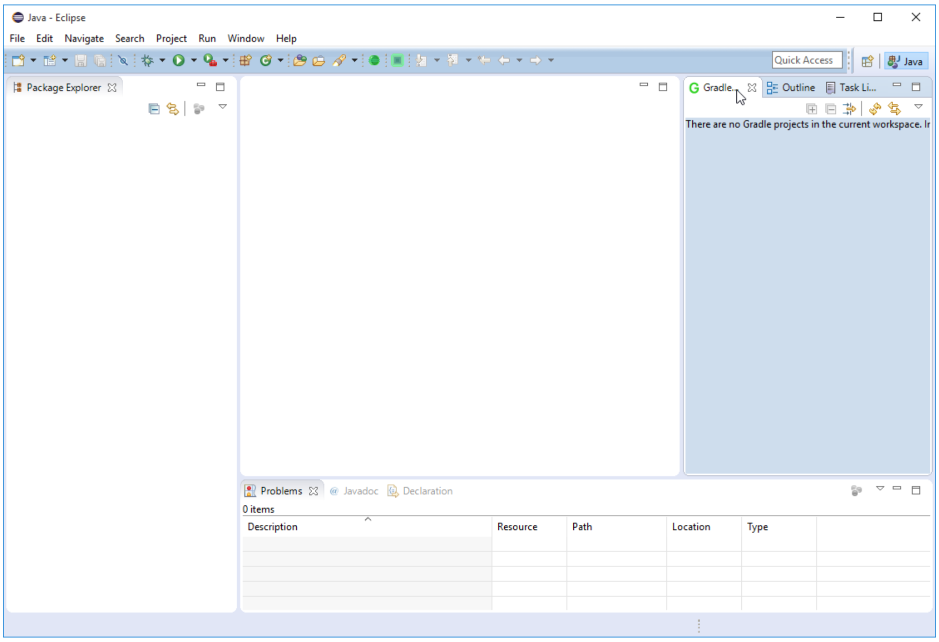
11 | Installs: 72.1K (5,734 last month) | Install

Marketplaces

< Back | Install Now > | Finish | Cancel

Figure 2.2: The Eclipse Marketplace

Open the new **Gradle Tasks** view for later use. To do this, open the **Window > Show View** menu and click on **Other...**. Search for `gradle` and select the **Gradle Tasks** view. This view now opens in your Eclipse. Move it to an appropriate place (Figure 2.3).



There are no Gradle projects in the current workspace. Ir

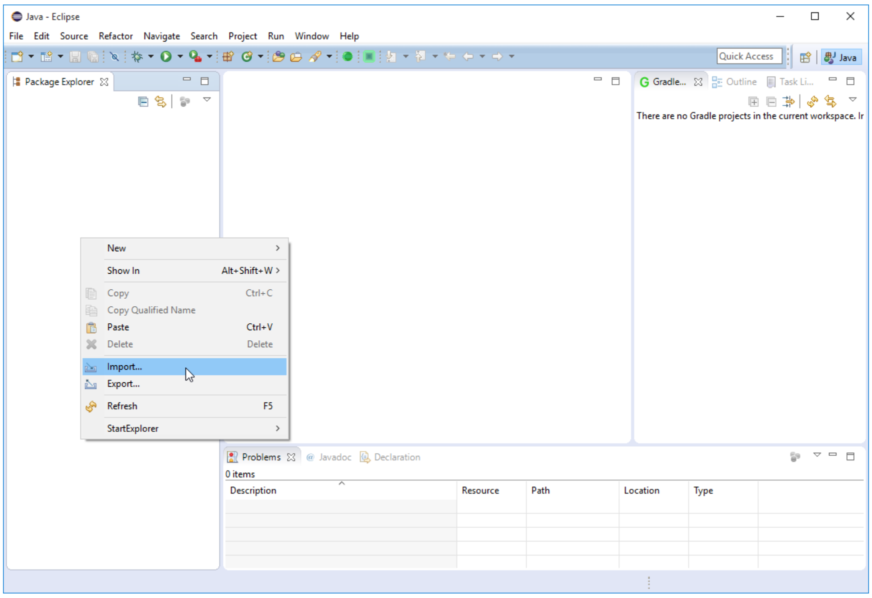
0 items

Figure 2.3: The Eclipse platform

2.1.4 Importing the extension template

To import the template:

1. Open the RapidMiner GitHub page in your browser.
2. Select the repository `rapidminer-extension-template`.
3. Clone the repository or download and unpack the .zip file into your workspace you selected when starting Eclipse the first time.
4. Open Eclipse, right-click on the **Package Explorer** view.
5. Select **Import...**, search for `gradle` and select **Gradle Project**. Then click **Next**.



There are no Gradle projects in the current workspace. Ir

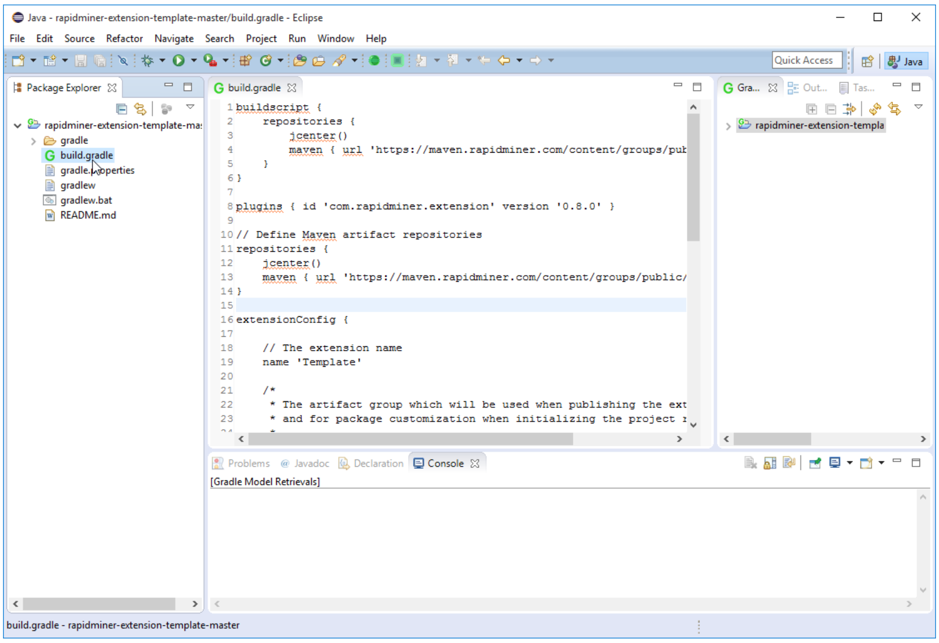
6. Browse for the folder that you just unpacked or cloned and click **Finish**.

(If you use an older version of Eclipse than 4.5.1, you need to browse for the folder, click **Build model**, select the complete project and click **Finish**.)

2.1.5 Using build.gradle

You can now see the imported project in the **Repository Manager** view (see Figure 2.4).

Open the project, double-click the file `build.gradle` to open it, and proceed to the next step - giving your extension an individual name and settings.


Java - rapidminer-extension-template-master/build.gradle - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Java
Package Explorer
rapidminer-extension-template-ma
gradle
build.gradle
gradle.properties
gradlew
gradlew.bat
README.md
build.gradle
Gra...
Out...
Tas...
rapidminer-extension-templa
Problems
Javadoc
Declaration
Console
build.gradle - rapidminer-extension-template-master

```
buildscript {
    repositories {
        jcenter()
        maven { url 'https://maven.rapidminer.com/content/groups/pub
    }
}

plugins { id 'com.rapidminer.extension' version '0.8.0' }

// Define Maven artifact repositories
repositories {
    jcenter()
    maven { url 'https://maven.rapidminer.com/content/groups/public/
}

extensionConfig {

    // The extension name
    name 'Template'

    /*
     * The artifact group which will be used when publishing the ext
     * and for package customization when initializing the project r
```

```
[Gradle Model Retrievals]
```

Figure 2.4: Eclipse with `build.gradle` file opened.

2.2 Using a Simple Editor

2.2.1 Installing Java

The first step of setting up your development environment is installing **Java**, if you haven't already. You can download Java here.

2.2.2 Downloading and installing your favorite text editor

The next step, if necessary, is installing your favorite text editor. For example, the Emacs text editor is popular.

2.2.3 Downloading the extension template

Once you have the tools, open the RapidMiner GitHub page in your browser. Select the repository `rapidminer-extension-template`.

Clone the repository or download and unpack the .zip file.

2.2.4 Using build.gradle

Open the file `build.gradle` of the extension template project in your text editor and proceed to the next step - giving your extension an individual name and settings.

CHAPTER 3

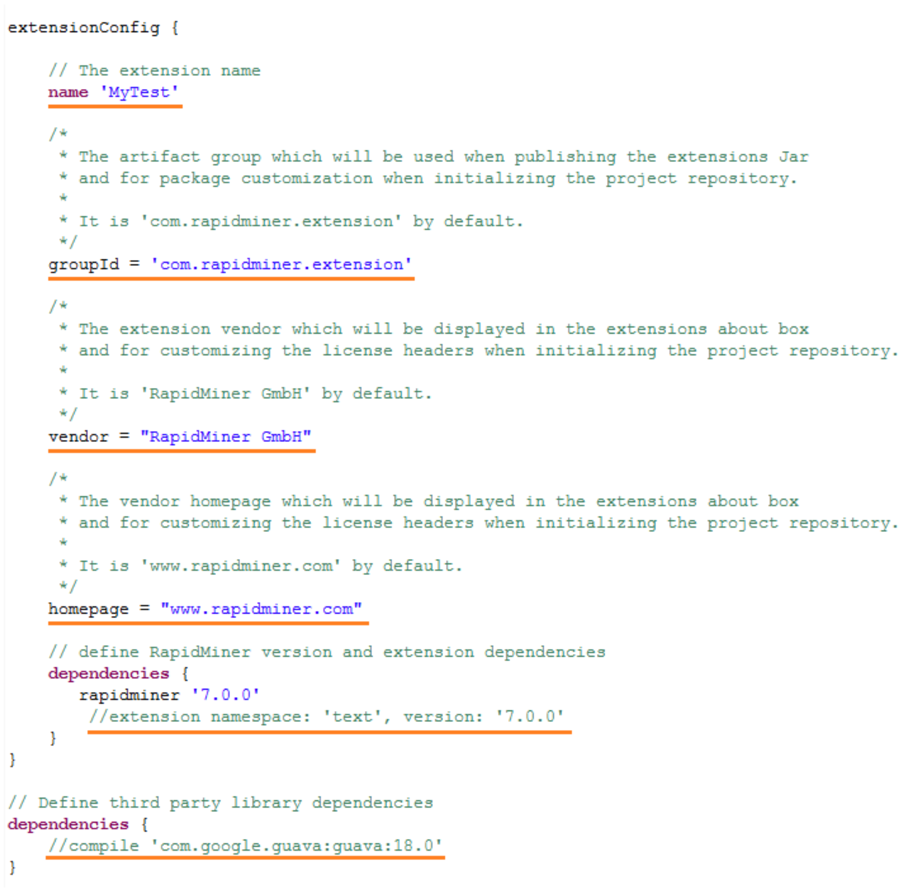
Individualizing Your Extension Settings

If you didn't open the file `build.gradle` in the extension template project yet, open it now.

3.1 Changing settings

First change the marked rows (see Figure 3.1) in the `build.gradle` file to meet your individual needs:

- Give your extension a name
- Define the `groupId` (it will be the name of your Java package)
- Define the vendor



```
extensionConfig {

    // The extension name
    name 'MyTest'

    /*
     * The artifact group which will be used when publishing the extensions Jar
     * and for package customization when initializing the project repository.
     *
     * It is 'com.rapidminer.extension' by default.
     */
    groupId = 'com.rapidminer.extension'

    /*
     * The extension vendor which will be displayed in the extensions about box
     * and for customizing the license headers when initializing the project repository.
     *
     * It is 'RapidMiner GmbH' by default.
     */
    vendor = "RapidMiner GmbH"

    /*
     * The vendor homepage which will be displayed in the extensions about box
     * and for customizing the license headers when initializing the project repository.
     *
     * It is 'www.rapidminer.com' by default.
     */
    homepage = "www.rapidminer.com"

    // define RapidMiner version and extension dependencies
    dependencies {
        rapidminer '7.0.0'
        //extension namespace: 'text', version: '7.0.0'
    }
}

// Define third party library dependencies
dependencies {
    //compile 'com.google.guava:guava:18.0'
}
```

Figure 3.1: `build.gradle` file with marked lines.

- Specify your website
- If your extension depends on other RapidMiner extensions, define them as extension dependencies
- If your extension depends on third-party libraries, define them as dependencies

(You don't know how to use libraries in Gradle? Check out the Gradle User

Guide chapter about dependency management.)

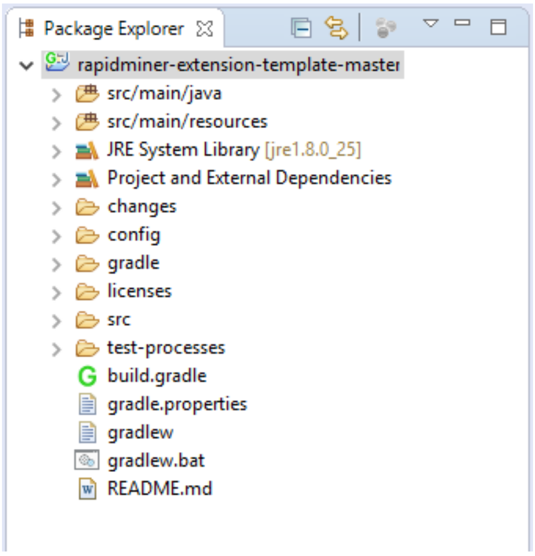
3.2 Initializing the project

To start, open a command prompt (for Windows, it is, for example, the Windows Power Shell). Browse to the extension template project and execute the following command:

```
./gradlew initializeExtensionProject
```

If you are using Eclipse, refresh the project folder. You can then see that the project is initialized and has source folders (which are waiting for your code).

If a simple refresh did not work, try a right-click on your project and select `Gradle > Refresh all`.



3.3 Adding an extension icon

The next step is to select an icon (48x48 pixels) for your extension. Name it `icon.png` and put it into the folder `src/main/resources/META-INF`. If the folder `META-INF` does not yet exist, create it and put your icon inside. If you are using Eclipse, refresh the project folder.

3.4 Installing the extension

If you are using Eclipse, select the project in the `Gradle Tasks` view and double-click the task `installExtension`.

Otherwise, open a command prompt, browse to the extension template project, and execute the following command:

```
./gradlew installExtension
```

3.5 Starting RapidMiner Studio

If RapidMiner Studio is not yet installed, download and install it. Then, open RapidMiner Studio and check whether your extension was loaded successfully. To check, open the `Extensions > About Installed Extensions` menu. You should see an entry `About NAME Extension...` with your extension's name. Select this entry and verify that the icon is loaded successfully and the vendor and URL are shown correctly.

Next, create your own operator.

CHAPTER 4

Creating Your Own Operator

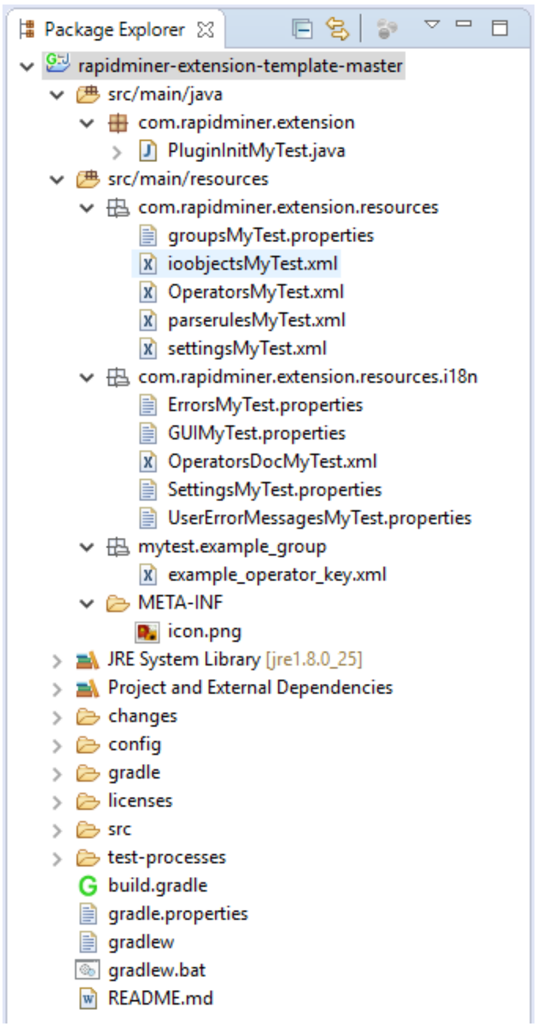
There are two types of operators in RapidMiner - normal and super operators. Super operators contain one or more sub processes. This guide starts with implementing a normal operator, but you can check out the advanced enhancements to learn how to develop a super operator.

In this section:

- Learn about the structure of the extension project and what each file does.
- Create your own operator class.
- Configure settings to load the operator into RapidMiner Studio.

4.1 Extension Project Folder Structure

The folder structure of the extension project looks like this:



```
rapidminer-extension-template-master
  src/main/java
    com.rapidminer.extension
      PluginInitMyTest.java
  src/main/resources
    com.rapidminer.extension.resources
      groupsMyTest.properties
      ioobjectsMyTest.xml
      OperatorsMyTest.xml
      parserulesMyTest.xml
      settingsMyTest.xml
    com.rapidminer.extension.resources.i18n
      ErrorsMyTest.properties
      GUIMyTest.properties
      OperatorsDocMyTest.xml
      SettingsMyTest.properties
      UserErrorMessagesMyTest.properties
    mytest.example_group
      example_operator_key.xml
    META-INF
      icon.png
  JRE System Library [jre1.8.0_25]
  Project and External Dependencies
  changes
  config
  gradle
  licenses
  src
  test-processes
  build.gradle
  gradle.properties
  gradlew
  gradlew.bat
  README.md
```

In the section on individualizing settings, you learned how to change the `build.gradle` file to make your individual extension. Statements like the extension name and the *groupId* are used to create the folder structure of your extension project. The *groupId*, for example, is the root path of your Java packages. The extension name is used to name the files of your project. The name of the example extension is `MyTest`.

The folder `src/main/java` contains your Java code, while `src/main/resources` contains configurations, documentation and the extension icon.

The following table describes the different generated files in your extension project (replace 'NAME' with the name of your extension):

File name	Description
PluginInit*NAME*.java	The Java class PluginInit*NAME*.java loads the extension into RapidMiner Studio. It contains certain methods that you can specify to add actions (for example, during the extension start-up or before closing the application). When using the extension on your RapidMiner Server, only the method *initPlugin()* is called.
groups*NAME*.properties	Specifies operator or data object colors.
ioobjects*NAME*.xml	Defines IOObjects (data objects), how they are called, their implementation class, and the renderer that renders the data object in the **Results** perspective.
Operators*NAME*.xml	Specifies operators and operator groups, as well as their location in the **Operator Tree** of the **Operators** view.
parserules*NAME*.xml	Automatically updates operator parameters (not frequently used).
settings*NAME*.xml	Specifies settings in the **Preferences** dialog.
Errors*NAME*.properties	Stores I18N error messages. You can load an error message in your code with `I18N.getErrorMessage(key, arguments);`.
GUI*NAME*.properties	Stores I18N GUI messages. You can

	load a message in your code with `I18N.getGUIMessage(key, arguments);`
OperatorsDoc*NAME*.xml	Contains translations from operator group keys to names as well as operator keys to names.
Settings*NAME*.properties	Contains user-friendly setting names and descriptions for setting keys defined in `settingsNAME.xml`. The names and descriptions are shown in the **Preferences** dialog.
UserErrorMessages*NAME* .properties	Contains user error messages for operators. They are displayed, when the code in an operator detects wrong parameter settings or otherwise encounters a problem. For each error name, you must define the properties `error.error_name.name` (name of the error), `error.error_name.short` (short error message) and `error.error_name.long` (long error message). Example use of an user error in your code: `throw new UserError(this, exception.getMessage(), "error_name");`.
example_operator_key.xml	Provides an example of operator documentation. The file is placed in the folder `mytest.example_group`. Create one folder, called `NAME.group_name`, per operator group and add one XML file per operator to the respective group folder.

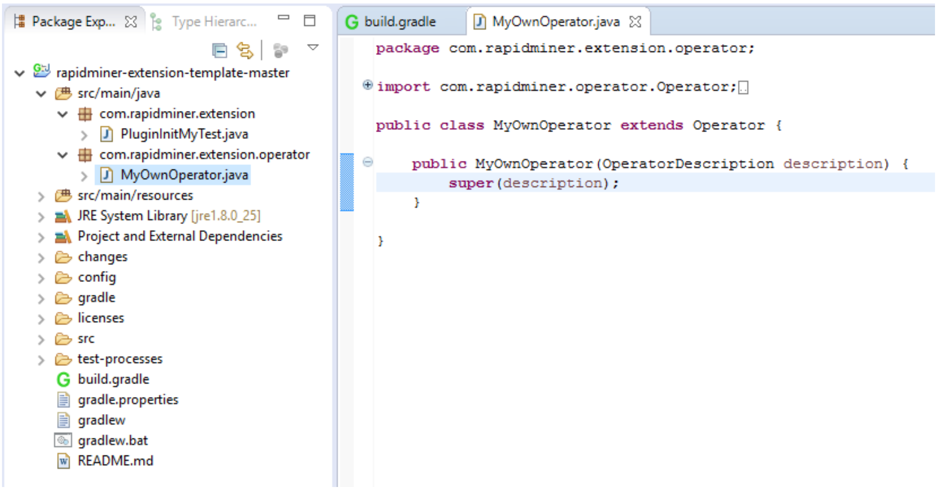
Now, you know how the project is structured. It's time to build your first operator class.

4.2 Creating an Operator Class

Time to create a new class for your operator. Each normal operator has to extend *Operator* or a subclass of *Operator*. There are many subclasses for more specialized operators (e.g., learning operators), but this example uses the simplest case. Frequently used subclasses of *Operator* are *AbstractLearner*, *AbstractReader* and *AbstractWriter*. If you are interested in more, take a look at the type hierarchy of *Operator*.

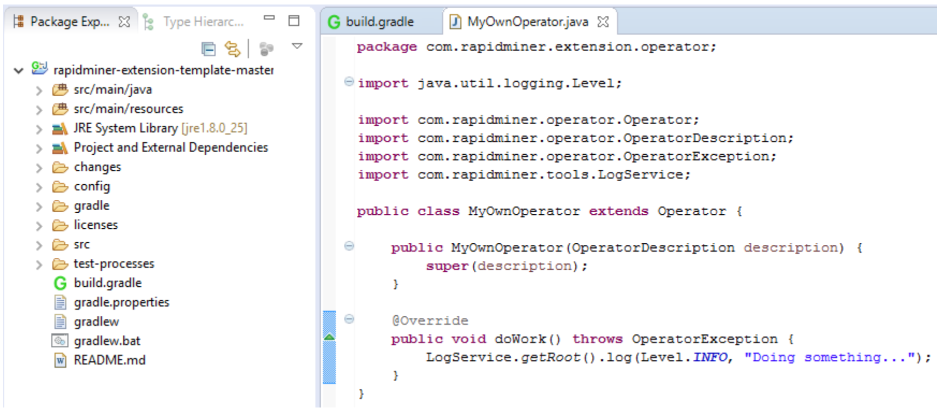
1. Create a new package for your operator class. It should have the name `groupId.operator`.
2. Create a new Java class with a meaningful name. The class has to extend *Operator* from the RapidMiner core. You have to override the constructor.

That's it. Your operator class should look like this:



```
package com.rapidminer.extension.operator;

import com.rapidminer.operator.Operator;

public class MyOwnOperator extends Operator {

    public MyOwnOperator(OperatorDescription description) {
        super(description);
    }

}
```

But wait! You want the operator to do something. Therefore, override the method *doWork()*. The default implementation simply does nothing. In this example, the operator will write logs to the **Log** panel.



```java
package com.rapidminer.extension.operator;

import java.util.logging.Level;

import com.rapidminer.operator.Operator;
import com.rapidminer.operator.OperatorDescription;
import com.rapidminer.operator.OperatorException;
import com.rapidminer.tools.LogService;

public class MyOwnOperator extends Operator {

    public MyOwnOperator(OperatorDescription description) {
        super(description);
    }

    @Override
    public void doWork() throws OperatorException {
        LogService.getRoot().log(Level.INFO, "Doing something...");
    }
}
```

This is the code of the operator structure:

```java
package com.rapidminer.extension.operator;

import java.util.logging.Level;
import com.rapidminer.operator.Operator;
import com.rapidminer.operator.OperatorDescription;
import com.rapidminer.operator.OperatorException;
import com.rapidminer.tools.LogService;

/**
 * Describe what your operator does.
 *
 * @author Insert your name here
 *
 */
public class MyOwnOperator extends Operator {

        /**
         * @param description
         */
        public MyOwnOperator(OperatorDescription description) {
                super(description);
        }

        @Override
        public void doWork() throws OperatorException {
           LogService.getRoot().log(Level.INFO, "Doing something...");
        }

}
```

Once you have implemented an operator, you'll want to test it in RapidMiner. Unfortunately, RapidMiner isn't prophetic. (Actually it could be, but using data mining methods for guessing class usage would be overkill.) Instead, you must specify class use.

4.3 Changing Configurations

To test your operator in RapidMiner Studio, you must first change some configuration in the extension project. Adapt the following two files (replace 'NAME' with the name of your extension):

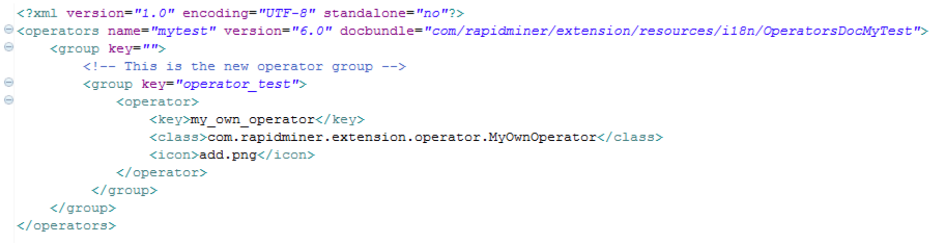
- Operators*NAME*.xml
- OperatorsDoc*NAME*.xml

Here is how to make your operators available to RapidMiner:

1. Decide on a name of the operator group to put the operator in.
2. Open the file `OperatorsNAME.xml` and add the new group with a key.
3. Add an operator tag. The operator has a key, the class (with the complete package path) and, optionally, an icon. If an icon is already part of the RapidMiner Core, you can just use it.

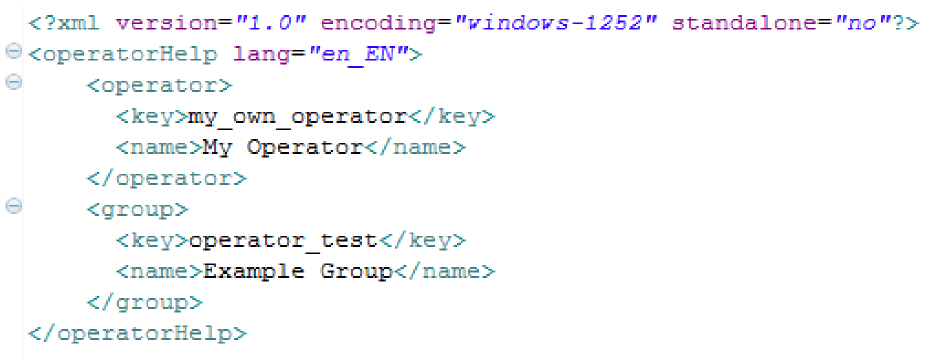
Otherwise:

- Create the folder `com.rapidminer.resources.icons` and subfolders called `16`, `24` and `48`.
- Paste the icon you want to use in the sizes 16x16 pixels, 24x24 pixels and 48x48 pixels in the respective folder.
- Add a 96x96 pixels version of the icon directly to the folder `com.rapidminer.resources.icons`.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<operators name="mytest" version="6.0" docbundle="com/rapidminer/extension/resources/i18n/OperatorsDocMyTest">
    <group key="">
        <!-- This is the new operator group -->
        <group key="operator_test">
            <operator>
                <key>my_own_operator</key>
                <class>com.rapidminer.extension.operator.MyOwnOperator</class>
                <icon>add.png</icon>
            </operator>
         </group>
    </group>
</operators>
```

At this point you can already build the extension and use it in RapidMiner Studio. You could see the operator in the **Operator Tree**, but its name would be the key. Instead, you need to define a translation from the key to the operator name in the file `OperatorsDocNAME.xml`. Define the key and name for every operator and the operator group.



```
<?xml version="1.0" encoding="windows-1252" standalone="no"?>
<operatorHelp lang="en_EN">
    <operator>
      <key>my_own_operator</key>
      <name>My Operator</name>
    </operator>
    <group>
      <key>operator_test</key>
      <name>Example Group</name>
    </group>
</operatorHelp>
```

4.3.1 Coloring your operator

If you want your operator to have a specific color, open the file `groupsNAME.properties` and add a color for your operator group. Use the operator group key to define the operator group and use a Hex code to define the operator color. For example:

```
# Operator group colors
group.operator_test.color = #5FDE35
```

4.3.2 Testing your operator

The following section describes how to test your operator.

Installing the extension

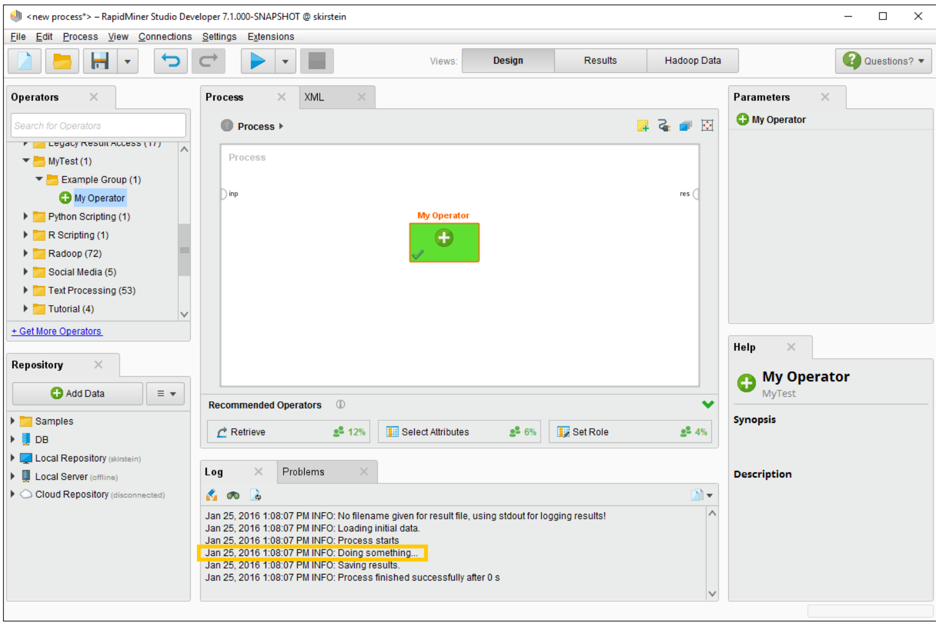
To install the extension:

- *If you are using Eclipse*, select the project in the **Gradle Tasks** view and double-click the task **installExtension**.
- *If using a text editor*, open a command prompt, browse to the extension template project and execute the following command:

  ```
  ./gradlew installExtension
  ```

Starting RapidMiner Studio

Open RapidMiner Studio and find your operator group in the **Operators Tree** panel in the **Extensions** group. Drag the new operator on to the process canvas. Verify that the operator has the color and icon that you defined and that it has a name instead of a key. You can run the process and the operator will produce a log entry, but depending on the purpose of your operator, there might be something missing.



```
Jan 25, 2016 1:08:07 PM INFO: No filename given for result file, using stdout for logging results!
Jan 25, 2016 1:08:07 PM INFO: Loading initial data.
Jan 25, 2016 1:08:07 PM INFO: Process starts
Jan 25, 2016 1:08:07 PM INFO: Doing something...
Jan 25, 2016 1:08:07 PM INFO: Saving results.
Jan 25, 2016 1:08:07 PM INFO: Process finished successfully after 0 s
```

It might be helpful to add input and output ports, for example, if your operator should process an example set. Look at the **Parameters** panel and the **Help** panel, and you'll see that there are not yet parameters that you can set or help for your operator. Adding parameters to your operator helps users to configure how your operator should be executed. Adding documentation to your operator is obviously useful. Use it to explain what your operator does and which parameters can be set. You can even add sample processes.

CHAPTER 5

Input and Output Ports

You define ports to get input from the process and to deliver results. In these pages:

- Learn to add ports to your operator.
- Set preconditions to allow only certain input types in your operator and to enable meta data handling.
- Set generation rules to reflect the changes made by your operator in the meta data.
- Use a PortPairExtender to, for example, create throughput ports that simply pass given data through the operator. (This is mostly used to influence the execution order when the operator itself does not need input and output ports.)

5.1 Adding Input and Output Ports

To define ports, you simply add them as private variables using the following lines of code:

```
private InputPort exampleSetInput = getInputPorts()
                                        .createPort("example set");
private OutputPort exampleSetOutput = getOutputPorts()
                                        .createPort("example set");
```

You can add more input ports, but you must set unique names for input ports and output ports for each operator. To follow the name convention, write the names in lower case and use blanks to separate words.

To fill the `doWork()` method with content, add a new attribute with random values from 1 to 10 to the example set received from the input port. You receive the example set by calling:

```
ExampleSet exampleSet = exampleSetInput.getData(ExampleSet.class);
```

Now, create a new attribute by using the `AttributeFactory`. Define a name for the new attribute and select the type of the new attribute. The class `Ontology` contains constants for the different types.

```
// get attributes from example set
Attributes attributes = exampleSet.getAttributes();
// create a new attribute
String newName = "newAttribute";
// define the name and the type of the new attribute
// valid types are
// - nominal (sub types: binominal, polynominal, string, file_path)
// - date_time (sub types: date, time)
// - numerical (sub types: integer, real)
```

```
Attribute targetAttribute = AttributeFactory
                           .createAttribute(newName, Ontology.REAL);
```

Set the table index of the new attribute and add it to the attributes of the example set.

```
targetAttribute.setTableIndex(attributes.size());
exampleSet.getExampleTable().addAttribute(targetAttribute);
attributes.addRegular(targetAttribute);
```

Add a value for each example of the new attribute:

```
for(Example example:exampleSet){
example.setValue(targetAttribute, Math.round(Math.random()*10+0.5));
}
```

After adding random values to the example set, deliver the resulting example set to the output port:

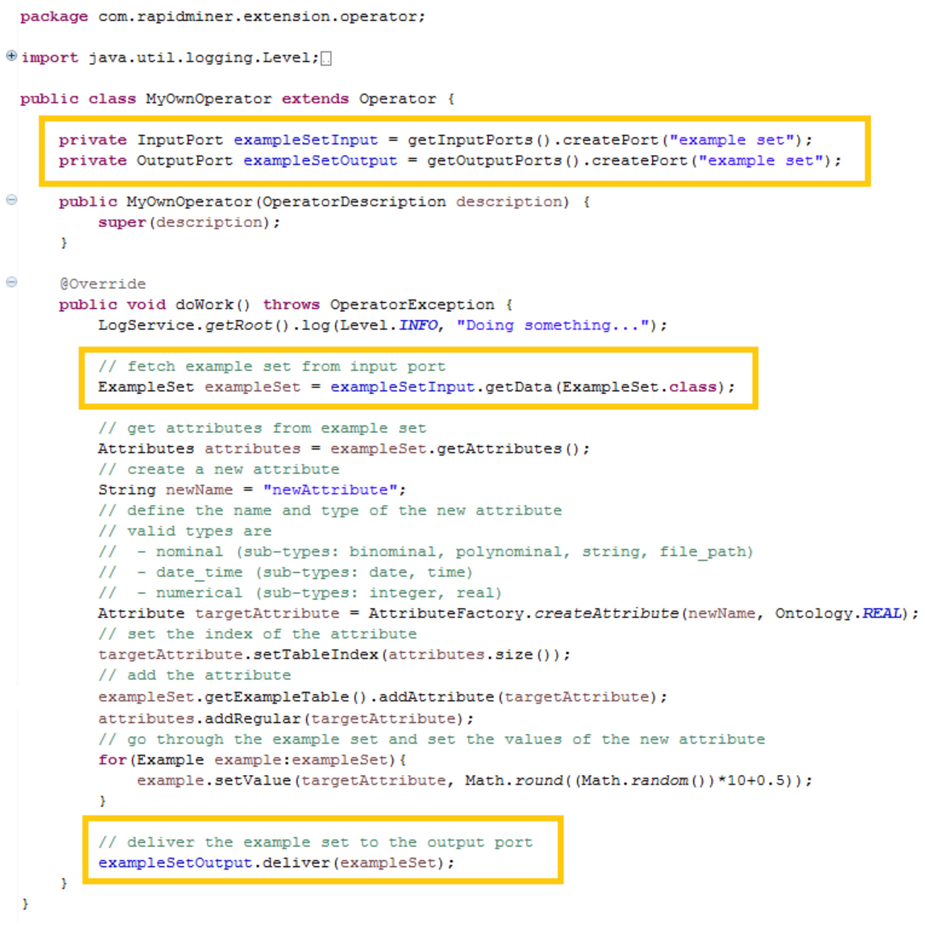
```
exampleSetOutput.deliver(exampleSet);
```

The complete operator class now looks like it is shown in Figure 5.1.

Install the extension again (with the Gradle task) and restart RapidMiner Studio. You can see that the operator now has an input and output port (Figure 5.2).

When you run the process that loads sample data and applies the example operator on it, you can see that the new attribute appears in the resulting example set (Figure 5.3).

The next step is to define preconditions to be sure that you only get the type of input you want to process.



```
package com.rapidminer.extension.operator;

import java.util.logging.Level;

public class MyOwnOperator extends Operator {

    private InputPort exampleSetInput = getInputPorts().createPort("example set");
    private OutputPort exampleSetOutput = getOutputPorts().createPort("example set");

    public MyOwnOperator(OperatorDescription description) {
        super(description);
    }

    @Override
    public void doWork() throws OperatorException {
        LogService.getRoot().log(Level.INFO, "Doing something...");

        // fetch example set from input port
        ExampleSet exampleSet = exampleSetInput.getData(ExampleSet.class);

        // get attributes from example set
        Attributes attributes = exampleSet.getAttributes();
        // create a new attribute
        String newName = "newAttribute";
        // define the name and type of the new attribute
        // valid types are
        //  - nominal (sub-types: binominal, polynominal, string, file_path)
        //  - date_time (sub-types: date, time)
        //  - numerical (sub-types: integer, real)
        Attribute targetAttribute = AttributeFactory.createAttribute(newName, Ontology.REAL);
        // set the index of the attribute
        targetAttribute.setTableIndex(attributes.size());
        // add the attribute
        exampleSet.getExampleTable().addAttribute(targetAttribute);
        attributes.addRegular(targetAttribute);
        // go through the example set and set the values of the new attribute
        for(Example example:exampleSet){
            example.setValue(targetAttribute, Math.round((Math.random())*10+0.5));
        }

        // deliver the example set to the output port
        exampleSetOutput.deliver(exampleSet);
    }
}
```

Figure 5.1: Operator class with ports

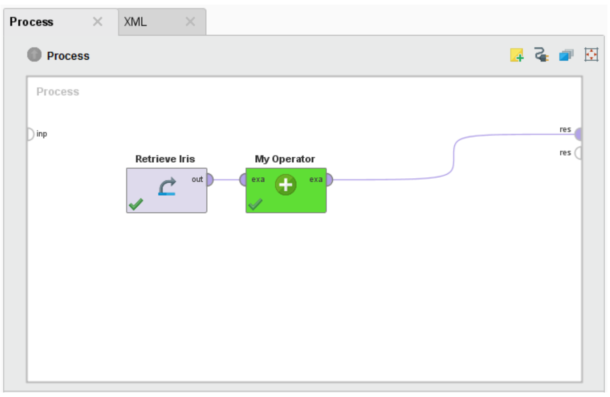
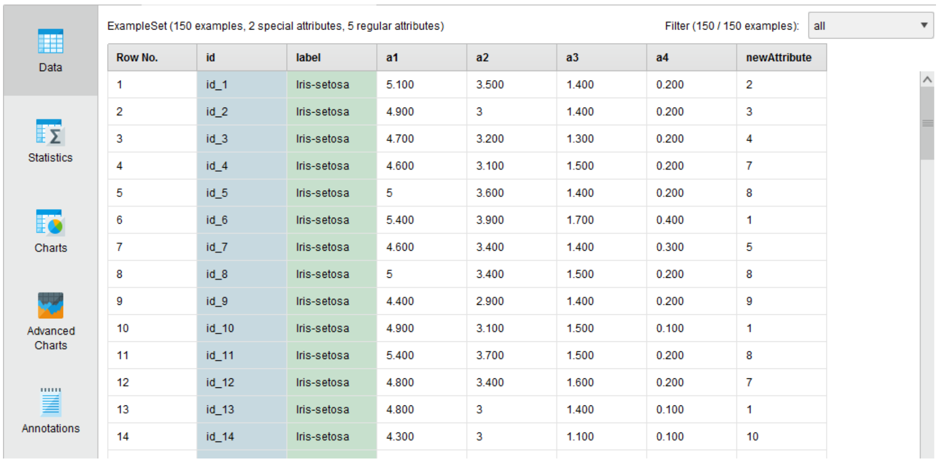


Figure 5.2: Operator with connected input and output ports



ExampleSet (150 examples, 2 special attributes, 5 regular attributes)

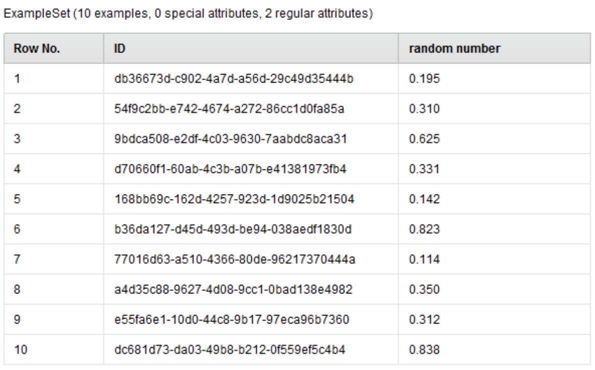
Filter (150 / 150 examples): all

Row No.	id	label	a1	a2	a3	a4	newAttribute
1	id_1	Iris-setosa	5.100	3.500	1.400	0.200	2
2	id_2	Iris-setosa	4.900	3	1.400	0.200	3
3	id_3	Iris-setosa	4.700	3.200	1.300	0.200	4
4	id_4	Iris-setosa	4.600	3.100	1.500	0.200	7
5	id_5	Iris-setosa	5	3.600	1.400	0.200	8
6	id_6	Iris-setosa	5.400	3.900	1.700	0.400	1
7	id_7	Iris-setosa	4.600	3.400	1.400	0.300	5
8	id_8	Iris-setosa	5	3.400	1.500	0.200	8
9	id_9	Iris-setosa	4.400	2.900	1.400	0.200	9
10	id_10	Iris-setosa	4.900	3.100	1.500	0.100	1
11	id_11	Iris-setosa	5.400	3.700	1.500	0.200	8
12	id_12	Iris-setosa	4.800	3.400	1.600	0.200	7
13	id_13	Iris-setosa	4.800	3	1.400	0.100	1
14	id_14	Iris-setosa	4.300	3	1.100	0.100	10

Figure 5.3: Operator result - example set with new attribute

5.1.1 Example operator generating a new example set

This example shows an operator, which creates a completely new example set with one nominal and one numerical attribute.



ExampleSet (10 examples, 0 special attributes, 2 regular attributes)

Row No.	ID	random number
1	db36673d-c902-4a7d-a56d-29c49d35444b	0.195
2	54f9c2bb-e742-4674-a272-86cc1d0fa85a	0.310
3	9bdca508-e2df-4c03-9630-7aabdc8aca31	0.625
4	d70660f1-60ab-4c3b-a07b-e41381973fb4	0.331
5	168bb69c-162d-4257-923d-1d9025b21504	0.142
6	b36da127-d45d-493d-be94-038aedf1830d	0.823
7	77016d63-a510-4366-80de-96217370444a	0.114
8	a4d35c88-9627-4d08-9cc1-0bad138e4982	0.350
9	e55fa6e1-10d0-44c8-9b17-97eca96b7360	0.312
10	dc681d73-da03-49b8-b212-0f559ef5c4b4	0.838

Figure 5.4: Example operator result - generated example set with a nominal and a numerical attribute

Check out the code:

```
package com.rapidminer.operator;

import java.util.LinkedList;
import java.util.List;
import java.util.UUID;
import com.rapidminer.example.Attribute;
import com.rapidminer.example.ExampleSet;
import com.rapidminer.example.table.AttributeFactory;
import com.rapidminer.example.table.DoubleArrayDataRow;
import com.rapidminer.example.table.MemoryExampleTable;
import com.rapidminer.operator.Operator;
import com.rapidminer.operator.OperatorDescription;
import com.rapidminer.operator.OperatorException;
import com.rapidminer.operator.ports.OutputPort;
import com.rapidminer.tools.Ontology;

public class MyOwnOperator extends Operator {

     private OutputPort exampleSetOutput = getOutputPorts()
          .createPort("example set");

     /**
      * @param description
      */
     public MyOwnOperator(OperatorDescription description) {
          super(description);
          getTransformer().addGenerationRule(exampleSetOutput,
               ExampleSet.class);
     }

     @Override
     public void doWork() throws OperatorException {
          // create the needed attributes
          List<Attribute> listOfAtts = new LinkedList<>();
          ExampleSet exampleSet;

          Attribute newNominalAtt = AttributeFactory
               .createAttribute("ID",
                    Ontology.ATTRIBUTE_VALUE_TYPE.NOMINAL);
          listOfAtts.add(newNominalAtt);

          Attribute newNumericalAtt = AttributeFactory
               .createAttribute("random number",
                    Ontology.ATTRIBUTE_VALUE_TYPE.NUMERICAL);
          listOfAtts.add(newNumericalAtt);

          // basis is a MemoryExampleTable, so create one
          // and pass it the list of attributes it
          // should contain
          MemoryExampleTable table = new
               MemoryExampleTable(listOfAtts);

          for (int i = 0; i < 10; i++) {
               // every row is a double array internally;
               //create and fill in data
               double[] doubleArray = new
                    double[listOfAtts.size()];
               doubleArray[0] = newNominalAtt.getMapping()
                    .mapString(UUID.randomUUID().toString());
               doubleArray[1] = Math.random();
               // create an example
               // create a DataRow from our double array
               // and add it to our table
               table.addDataRow(new
                    DoubleArrayDataRow(doubleArray));
          }

          // finally create the ExampleSet from the table
          exampleSet = table.createExampleSet();

          exampleSetOutput.deliver(exampleSet);
     }
}
```

This operator just creates 10 random examples. An enhancement could be to create a parameter that defines how many examples the operator creates.

5.2 Adding Preconditions to Input Ports

As you saw after starting RapidMiner Studio, the operator works. However, it does not alert the user if nothing is connected or an object of wrong type is connected to the input port. To change this behavior and improve operator usability, you can add preconditions to the ports. These preconditions will register errors, if they are not fulfilled and are registered at the time of operator construction. To do so, add a few code fragments to the constructor. For example, this precondition checks whether a compatible *IOObject* is delivered:

```
public MyOwnOperator( OperatorDescription description ){
    super(description);
    exampleSetInput.addPrecondition(
        new SimplePrecondition( exampleSetInput,
            new MetaData(ExampleSet.class) ));
}
```

Since this is one of the most common cases, you can use a shortcut to achieve it. Specify the target *IOObject* class when constructing the input port:

```
private InputPort exampleSetInput =
getInputPorts().createPort("example set", ExampleSet.class);
```

There are many more special preconditions. Some test whether an example set satisfies specific conditions, whether it contains a special attribute of a specific role, whether an attribute with a specific name is inserted, and others. For example, you could add a precondition that tests if the attribute *test* is part of the input example set. The attribute can have any type.

```
exampleSetInput.addPrecondition(
    new ExampleSetPrecondition( exampleSetInput,
        new String[]{"test"}, Ontology.ATTRIBUTE_VALUE) );
```

The *ExampleSetPrecondition* can also check whether the regular attributes are of a certain type, which special attributes have to be contained, and of which type they must be. If the user inserts the operator into a process without connecting an example set output port with the input port, an error is shown. If the user attaches an example set without the *test* attribute, a warning is shown.

The next step is to define generation rules for output ports.

5.3 Adding Generation Rules to Output Ports

At this point, if you connect your operator with another operator that receives an *ExampleSet* object, it alerts that it hasn't receive the correct object. This is because your operator hasn't yet done transformation of the meta data. It makes use of the meta data to check the preconditions, but doesn't deliver it to the output port. You can change this by adding generation rules in the constructor:

```
public MyOwnOperator( OperatorDescription description ){

    super(description);
    exampleSetInput.addPrecondition(
        new ExampleSetPrecondition( exampleSetInput,
            new String[]{"test"}, Ontology.ATTRIBUTE_VALUE ));

    getTransformer().addPassThroughRule(exampleSetInput,
        exampleSetOutput);
}
```

This rule simply passes the received meta data to the output port, which causes the warning to vanish. However, the meta data doesn't reflect the actual delivered data. Remember, you can add an attribute for that. This should be reflected in the meta data, which is why you must implement a special transformation rule. To do so, use an anonymous class so that it looks like this:

```
getTransformer().addRule(
   new ExampleSetPassThroughRule( exampleSetInput, exampleSetOutput,
         SetRelation.EQUAL){

             @Override
             public ExampleSetMetaData modifyExampleSet(
                 ExampleSetMetaData metaData ) throws
                     UndefinedParameterError{
                         return metaData;
             }
});
```

However, this only passes the received meta data to the output port, it doesn't account for changes to the meta data. By adding a hook, you can grab the meta data and change it so that it reflects the changes made on the data during operator execution. After adding the code, the method looks like this:

```
getTransformer().addRule(
   new ExampleSetPassThroughRule( exampleSetInput, exampleSetOutput,
         SetRelation.EQUAL){

             @Override
             public ExampleSetMetaData modifyExampleSet(
                 ExampleSetMetaData metaData ) throws
                     UndefinedParameterError {
                         metaData.addAttribute(
                             new AttributeMetaData("newAttribute",
                                 Ontology.REAL));
                         return metaData;
             }
});
```

If you change the type and name of an attribute, you can also change the meta data like this:

```
AttributeMetaData testAMD = metaData.getAttributeByName("test");
if(testAMD!=null){
    testAMD.setType(Ontology.DATE_TIME);
    testAMD.setName( "date(" + testAMD.getName() + ")" );
    testAMD.setValueSetRelation(SetRelation.UNKNOWN);
}
return metaData;
```

You should change the meta data according to the changes of the data through your operator.

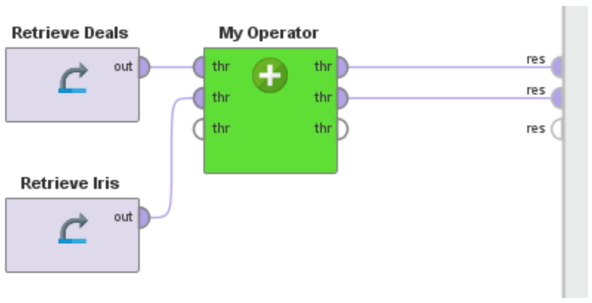
If you don't know the details of the outgoing data but you know the output port type, you can add a very simple generation rule that defines the type delivered at the output port.

```
getTransformer().addGenerationRule(exampleSetOutput,
    ExampleSet.class);
```

In the next step, you will learn how to use the PortExtender.

5.4 Dynamically Adding Ports with the PortPairExtender

Sometimes it's useful to add ports that simply pass data through the operator without changing it - 'through' ports. If your operator does not need input and output ports, best practice is to add through ports to control the process execution order. The *PortPairExtender*, a sub-class of *PortExtender*, ensures that the through ports are pair-wise.



The *PortPairExtender* is also used to pass input data from a super operator to its subprocesses. You can find more details in the advanced chapter about super operators.

5.4.1 Adding through ports

To define through ports, simply add a *PortPairExtender* as a private variable, using the following code:

```
private final PortPairExtender dummyPorts =
    new DummyPortPairExtender("through", getInputPorts(),
        getOutputPorts());
```

Then, initialize the ports and add the pass through rule to handle the meta data:

```
dummyPorts.start();
getTransformer().addRule(dummyPorts.makePassThroughRule());
```

Add the command to pass the data from the input through ports to the output through ports at the end of the `doWork()` method.

```
dummyPorts.passDataThrough();
```

Simple example operator with through ports

```
public class MyOwnOperator extends Operator {

    private PortPairExtender dummyPorts =
        new DummyPortPairExtender("through", getInputPorts(),
            getOutputPorts());

    public MyOwnOperator(OperatorDescription description) {
        super(description);

        dummyPorts.start();
        getTransformer().addRule(dummyPorts.makePassThroughRule());
    }

    @Override
    public void doWork() throws OperatorException {
        LogService.getRoot().log(Level.INFO, "Doing something...");

        // PASS THROUGH PORTS
        dummyPorts.passDataThrough();
    }
}
```

The next step is to add parameters to your operators.

CHAPTER 6

Adding Operator Parameters

Ports handle the data that flows through your operator. You can, however, also define parameters that influence the behavior of the operator.

Parameters are presented in the **Parameters** panel of RapidMiner Studio, where users can alter the parameter's values. There are several types of parameters available for defining real or integer numbers, strings, and collections of strings in combo boxes (either editable or not). Also available are special types for selecting an attribute or several attributes. For the most complex parameter type, you might even define a GUI component as a configuration wizard.

Back to the operator class to add parameters. In fact, you just have to override one method:

```
@Override
```

```
public List<ParameterType> getParameterTypes(){
    return super.getParameterTypes();
}
```

Notice that you have to return a list of *ParameterType*s. It's good practice to call the super method to retrieve the parameters defined in extending operators or abstract classes that provide basic functionality. Otherwise, the functionality provided by the super class might fail because you haven't defined the needed parameters.

For now, add a parameter defining which text should be logged to the console when the operator is executed. It looks like this:

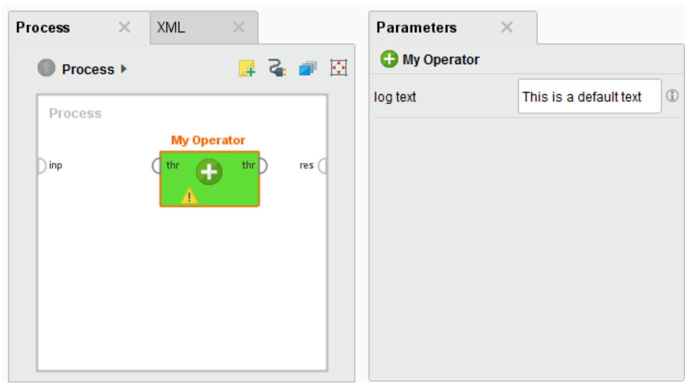
```
@Override
public List<ParameterType> getParameterTypes(){
    List<ParameterType> types = super.getParameterTypes();

    types.add(new ParameterTypeString(
        PARAMETER_TEXT,
        "This parameter defines which text is logged to
        the console when this operator is executed.",
        "This is a default text",
        false));
    return types;
}
```

First, retrieve the list of *ParameterType*s of the super class and then add your own parameter. In this example, the parameter is of type String and is named with the public constant PARAMETER_TEXT. The string that follows should describe the functionality of the parameter type; it is shown in the info tool tip of the parameter. The next string is the default value of the parameter, followed by the last parameter, which determines if the parameter is expert or not. In this example, the parameter is not an expert parameter.

Before looking at the result, you must add the constant to the class. Simply define a public constant:

```
public static final String PARAMETER_TEXT = "log text";
```

The **Parameters** panel now looks like this:



log text: This is a default text

You could use the class *ParameterTypeInt* for integer values or *Parameter-TypeDouble* for double values, but there are also *ParameterType*s for files, dates, category selection and much more. Check out which *ParameterType* constructors are available!

6.1 Expert parameters

Parameters can be either normal or expert. Expert parameters aren't shown until the user switches to expert mode. Therefore, it is good practice to define parameters as expert if their effect is only understandable by those with deeper knowledge of the underlying algorithm. All expert parameters must have default values so that the user is not required to define a parameter he cannot understand. To enable this, simply define in the constructor calls of your *ParameterType*s whether a parameter is an expert parameter or not.

6.2 Using parameters

After defining the parameter, use it to individualize the message that your operator writes to the **Log** panel. First, retrieve the value the user entered and store it in a local variable:

```
String text = getParameterAsString(PARAMETER_TEXT);
```

Then, use the local variable to change the output of the operator. There are several *getParameterAsXXX()* methods that you can call to get the value in the correct type.

6.3 Parameter dependencies

Defining parameter dependencies provides the user with further guidance. For example, some parameters may only be used if other parameters are set. Using these dependencies indicates to the user which parameter will have an effect preventing time spent with irrelevant parameters. If, for example, you are familiar with the large amount of parameters that kernel-based methods like the SVM offer, you probably understand why this is important.

You can add a Boolean parameter determining whether the operator should log custom text. In this case, if checked, the user sees the parameter field for the text. To introduce another parameter with its constant:

```
public static final String PARAMETER_USE_CUSTOM_TEXT =
    "use custom text"
```

Now, build a parameter condition like in the following code:

```
@Override
```

```
public List<ParameterType> getParameterTypes(){
    List<ParameterType> types = super.getParameterTypes();

    types.add(new ParameterTypeBoolean(
        PARAMETER_USE_CUSTOM_TEXT,
        "If checked, a custom text is printed to the log view.",
        false,
        false));

    ParameterType type = new ParameterTypeString(
        PARAMETER_TEXT,
        "This parameter defines which text is logged to
        the console when this operator is executed.",
        "This is a default text",
        false);

    type.registerDependencyCondition(
        new BooleanParameterCondition(
            this, PARAMETER_USE_CUSTOM_TEXT, true, true));

    types.add(type);

    return types;
}
```

For registering the condition, you had to remember the type in a local variable, which is then added to the list separately. Here you added a `BooleanParameterCondition`, which needs to reference a `ParameterHandler`. For operators, this is the operator itself. The second argument is the name of the references parameter. The two *Boolean* values indicate 1) if the parameter becomes mandatory if the condition is satisfied and 2) the value the references parameter must have in order to fulfil the condition.

The resulting parameter panel now looks like this, depending on the parameter settings:

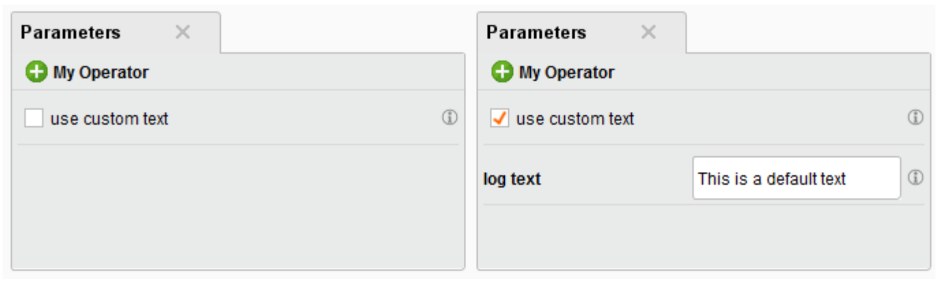


Figure 6.1: Parameter view with parameter condition

The next step is to document the behavior of your operator to provide help.

CHAPTER 7

Providing Operator Documentation

It’s natural to add documentation to program code, but this does not help the end user who never sees any part of the program code.

You have already configured the files `OperatorsNAME.xml` and `OperatorsDocNAME.xml` to provide the operator with an icon and name.

The extension template contains an example of operator documentation. The XML file is placed in the folder `NAME.example_group` (see Figure 7.1), where ’NAME’ is the extension name. You should create one folder, called `NAME.group_name`, per operator group, and add one XML file per operator to the respective group folder.

In this example, the folder is named `mytest.operator_test`, because the extension name is ’MyTest’ and the group ID is ’operator_test’. The file is called `my_own_operator.xml`, because the operator key is ’my_own_operator’.

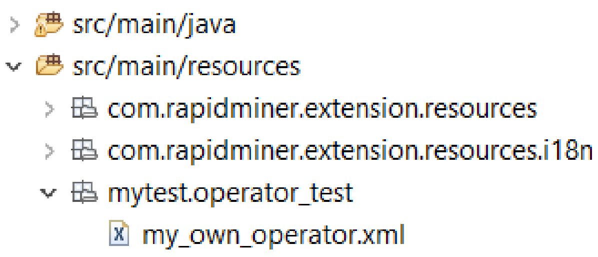

src/main/java
src/main/resources
com.rapidminer.extension.resources
com.rapidminer.extension.resources.i18n
mytest.operator_test
my_own_operator.xml

Figure 7.1: Example folder with operator documentation

Now, copy the sample file content and adapt it to your needs:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="../../../../../
rapidminerreferencemanual/documentation2html.xsl"?>
<p1:documents xmlns:p1="http://rapid-i.com/schemas/
      documentation/reference/1.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://rapid-i.com/schemas/
      documentation/reference/1.0 ">

<operator key="operator.mytest:my_own_operator" locale="en"
    version="6.5.000">
<title>My own operator</title>
<synopsis>This is an example Operator.
  It writes to the Log View when being executed.
</synopsis>
<text>
    <paragraph>First paragraph.</paragraph>
    <paragraph>Second paragraph.</paragraph>
</text>
<inputPorts>
    <port name="input" type="com.rapidminer.example.ExampleSet">
        Input port description.
    </port>
</inputPorts>
<outputPorts>
    <port name="output" type="com.rapidminer.example.ExampleSet">
        Output port description.
    </port>
</outputPorts>

<!-- description of the parameters and the corresponding values -->
<parameters>
    <parameter key="log_text" type="string">
        Description here
    </parameter>
</parameters>

<tutorialProcesses>
    <tutorialProcess key="log.test"
     title="Logging custom text">
        <description>
            <paragraph>
                Tutorial description here.
            </paragraph>
        </description>
        <process version="6.5.000">
            ...
        </process>
    </tutorialProcess>
</tutorialProcesses>

<relatedDocuments>
    <!-- ... -->
</relatedDocuments>
    </operator>
</p1:documents>
```

Change the content between the tags to describe your operator, ports and parameters.

Don't forget to adapt the operator key in this document. Otherwise, your operator icon will not display correctly in the operator help.

You can add a tutorial process. To do so, complete the tutorial description and exchange the process tags with the process XML of your tutorial process.

How to extract a process XML

1. Build the RapidMiner process that you want to add to the documentation.
2. Then, open the **View > Show Panel > XML** menu. The XML panel opens.
3. Move the XML panel to a comfortable place in RapidMiner Studio. You can see the process in XML format in the panel.
4. Copy the complete XML code to extract the process.

Congratulations for your first complete operator, repeat chapters 4 through 7 of this guide for each operator that you want included in your extension.

Then, publish your extension if you want to provide it to the Community or check out which advanced enhancements you can build for RapidMiner Studio.

CHAPTER 8

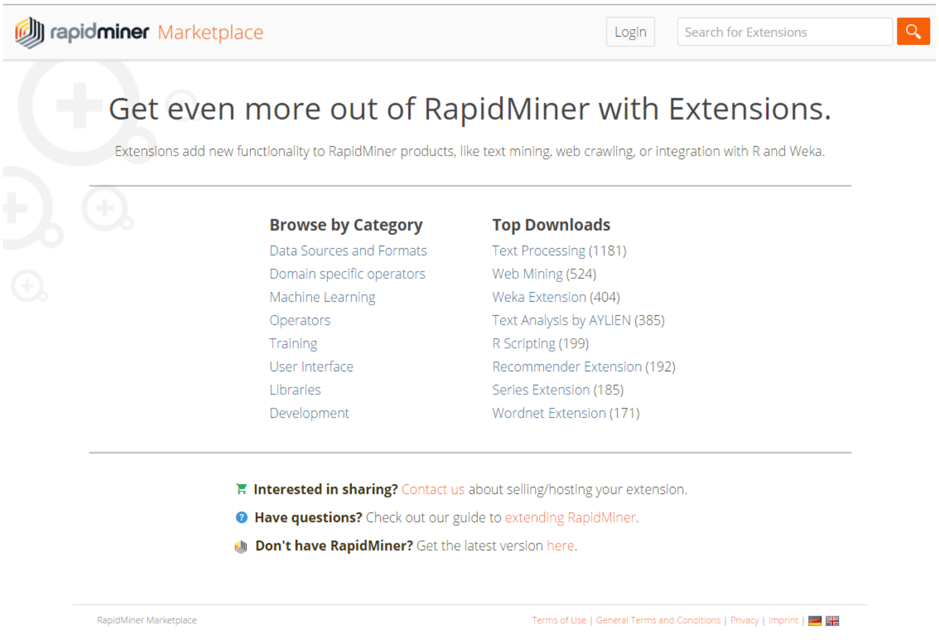
Publishing Your Own Extension

The RapidMiner Marketplace provides many extensions, developed by both RapidMiner and other contributers. To publish your extension, log into the Marketplace (see Figure 8.1).

Click the **Login** button on the top to login. If you don't have an account, click **Create new account**. Fill in the fields of the sign up form and click at **Create new account** again (see Figure 8.2).

You will receive an email with an activation link (see Figure 8.3).

Click the activation link to finish the sign up. Then, check out the options you have in the RapidMiner Marketplace. You can use bookmarks for faster access to extensions that you are interested in. You can search and learn about new extensions. And, most importantly for this document, you can publish your own extension. To publish, click on the **Contact us** link at the



Get even more out of RapidMiner with Extensions.

Extensions add new functionality to RapidMiner products, like text mining, web crawling, or integration with R and Weka.

Browse by Category
Data Sources and Formats
Domain specific operators
Machine Learning
Operators
Training
User Interface
Libraries
Development

Top Downloads
Text Processing (1181)
Web Mining (524)
Weka Extension (404)
Text Analysis by AYLIEN (385)
R Scripting (199)
Recommender Extension (192)
Series Extension (185)
Wordnet Extension (171)

Interested in sharing? Contact us about selling/hosting your extension.
Have questions? Check out our guide to extending RapidMiner.
Don't have RapidMiner? Get the latest version here.

RapidMiner Marketplace
Terms of Use | General Terms and Conditions | Privacy | Imprint |

Figure 8.1: RapidMiner Marketplace

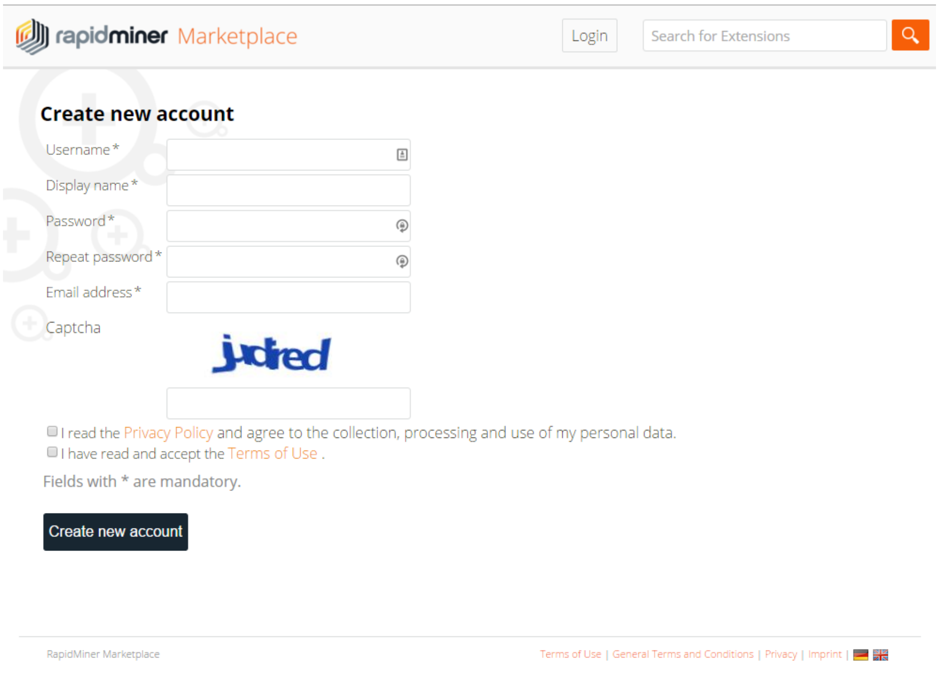
bottom (see Figure 8.4).

Once clicked, a request form opens (see Figure 8.5). In it, you can add information about your extension such as a name, a description, the development stage and more.

When the form is complete, click **Submit Request**.

When received, RapidMiner will check the request and create a product page for your extension. You will be informed by email as soon as it is ready. Once this is done, you can administer your product and upload the first version.

Then, find the **My products** menu (see Figure 8.6) and open it. Click on the product to change the product name, define the extension category, write a short description and upload a new package for this product. This is also the


rapidminer Marketplace

Login

Search for Extensions

Create new account

Username*	
Display name*	
Password*	
Repeat password*	
Email address*	
Captcha	

☐ I read the Privacy Policy and agree to the collection, processing and use of my personal data.
☐ I have read and accept the Terms of Use .

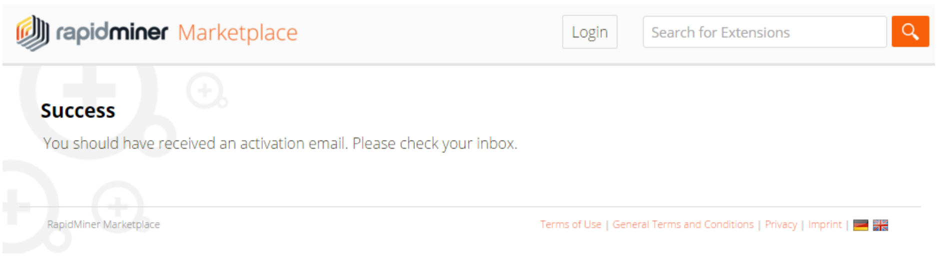
Fields with * are mandatory.

Create new account

RapidMiner Marketplace

Terms of Use | General Terms and Conditions | Privacy | Imprint |

Figure 8.2: RapidMiner Marketplace - Creating a new account


rapidminer Marketplace

Login

Search for Extensions

Success

You should have received an activation email. Please check your inbox.

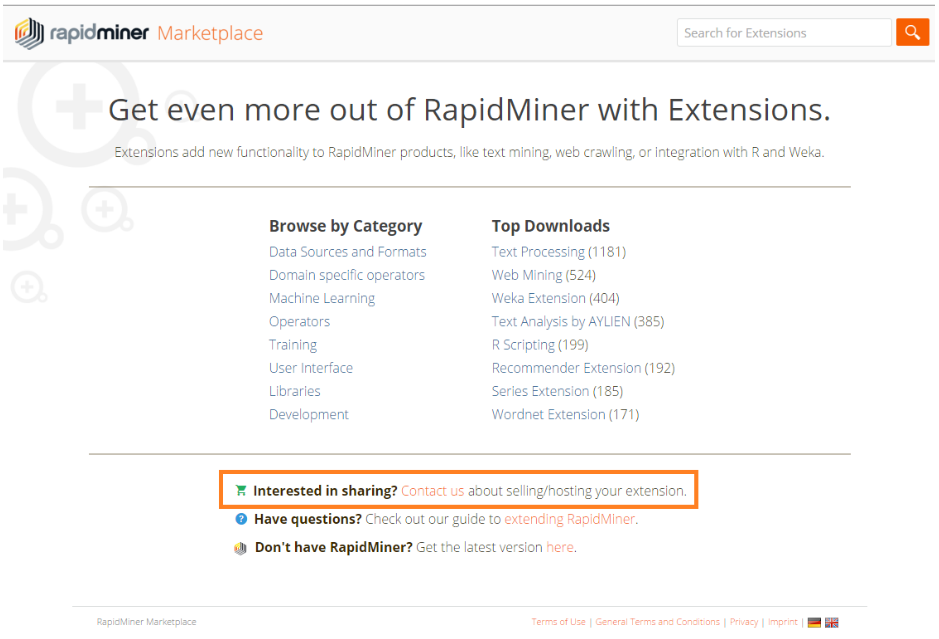
RapidMiner Marketplace

Terms of Use | General Terms and Conditions | Privacy | Imprint |

Figure 8.3: RapidMiner Marketplace - successful creation of an account

place to upload the first version of the JAR file of your own extension.

Now other users can download and use your extension. Congratulations!


rapidminer Marketplace

Search for Extensions

Get even more out of RapidMiner with Extensions.

Extensions add new functionality to RapidMiner products, like text mining, web crawling, or integration with R and Weka.

Browse by Category	Top Downloads
Data Sources and Formats	Text Processing (1181)
Domain specific operators	Web Mining (524)
Machine Learning	Weka Extension (404)
Operators	Text Analysis by AYLIEN (385)
Training	R Scripting (199)
User Interface	Recommender Extension (192)
Libraries	Series Extension (185)
Development	Wordnet Extension (171)

Interested in sharing? Contact us about selling/hosting your extension.

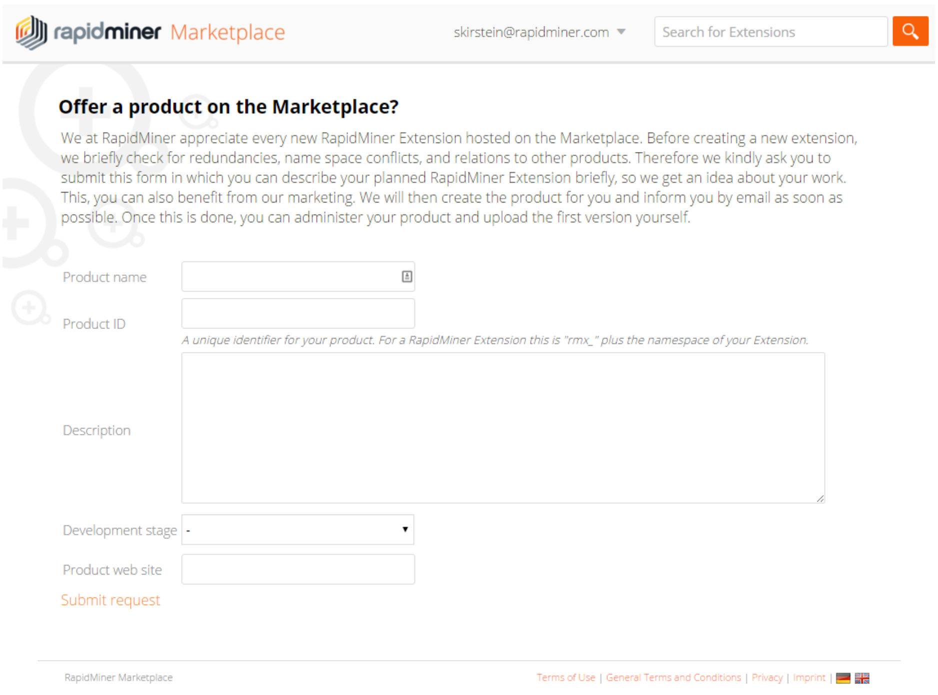
Have questions? Check out our guide to extending RapidMiner.

Don't have RapidMiner? Get the latest version here.

RapidMiner Marketplace

Terms of Use | General Terms and Conditions | Privacy | Imprint |

Figure 8.4: Marketplace - Sharing your extension



Offer a product on the Marketplace?

We at RapidMiner appreciate every new RapidMiner Extension hosted on the Marketplace. Before creating a new extension, we briefly check for redundancies, name space conflicts, and relations to other products. Therefore we kindly ask you to submit this form in which you can describe your planned RapidMiner Extension briefly, so we get an idea about your work. This, you can also benefit from our marketing. We will then create the product for you and inform you by email as soon as possible. Once this is done, you can administer your product and upload the first version yourself.

Product name	
Product ID	*A unique identifier for your product. For a RapidMiner Extension this is "rmx_" plus the namespace of your Extension.*
Description	
Development stage	-
Product web site	

Submit request
RapidMiner Marketplace
Terms of Use | General Terms and Conditions | Privacy | Imprint |

Figure 8.5: RapidMiner Marketplace - Request form for publishing your extension

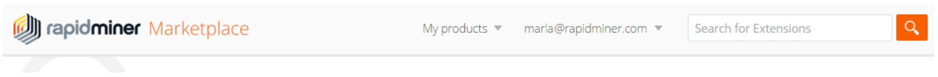


Figure 8.6: RapidMiner Marketplace - menu bar with ‘My products’ menu

CHAPTER 9

Advanced Enhancements

If simple operators do not fit your needs, there are several more advanced possibilities to enhance RapidMiner Studio. Learn how to use the `PluginInit` class to change some of RapidMiner's behavior during startup, before any operator is executed, and how to:

- build a super operator
- create your own data objects
- make custom configurators
- create custom graphical elements

9.1 The PluginInit class

The class offers hooks for changing RapidMiner's behavior during startup. RapidMiner automatically creates the class `PluginInit` when you initialize your extension. It does not have to extend any super class, since its methods are accessed via reflection. There are four methods that are called during startup of RapidMiner. The following might be interesting for you:

```
public static void initPlugin()
```

The `initPlugin` method is called directly after the extension is initialized. It is the first hook during start up. No initialization of the operators or renderers has taken place when this is called.

```
public static void initGui(MainFrame mainframe)
```

The `initGui` method, called before the GUI of the mainframe is created, is called during start up as the second hook. The `MainFrame` is passed as an argument to register GUI elements, while the operators and renderers are registered.

```
public static void initFinalChecks()
```

`initFinalChecks` is the last hook before the splash screen closes.

If you install your extension on RapidMiner Server, the only method called during startup is `initPlugin`.

9.2 Creating Super Operators

There are two types of operators in RapidMiner - normal and super operators. Super operators contain one or more sub processes. You started by implementing a normal operator, but sometimes an operator relies on the execution of other operators. And sometimes these operators should be user defined. Take the cross-validation as an example. The user might specify the learner and the way performance is measured; it then executes these subprocesses as needed.

Assume you have an operator that should loop over values, but the **Loop values** operator in RapidMiner Studio loops over the values of an attribute. You want an operator that loops over values in a given range, with a given step size, and you don't want to create an attribute for this purpose. Instead, build a super operator that re-executes its inner operators for each step of a given range. To do this, create a new class, but this time extend the *OperatorChain* class. As with a simple operator, you must implement a constructor. The empty class looks like this:

```
public class LoopValuesRange extends OperatorChain{

    public LoopValuesRange(OperatorDescription description) {
        super(description, "Executed Process");
    }
}
```

In contrast to the simple operator, you must give the super constructor the names of the subprocesses you are going to create inside your super operator. The number of names you pass to the super constructor determines the number of created subprocesses. If you want to follow the naming convention, start each word uppercase and use blanks to separate words. Later, you might access these subprocesses by index to execute them. But first, define some ports to pass data to the super operator.

9.2.1 Using the PortPairExtender for super operators

You learned earlier how to use the *PortPairExtender* to create throughput ports for a simple operator. You also need this class to pass data from the super operator to the subprocess and back. Do it in a general way so that the user can pass any number and any type of object to the inner process. You might know this behavior from the **Loop** operator of RapidMiner Studio. The code for adding this *PortPairExtender* looks like this:

```
private final PortPairExtender inputPortPairExtender =
    new PortPairExtender("input", getInputPorts(), getSubprocess(0)
        .getInnerSources());
```

In addition to the *PortPairExtender*, there is also a *PortExtender*. Use the *PortPairExtender* to get an equal number of input and output ports. Take a close look at the *PortPairExtender* constructor. In addition to the name, you must specify which input ports the extender should attach to. The *getInputPorts* method delivers the input ports of the current operator (so the port extender is attached on the left side of the operator box). The paired ports are added to the inner sources of the first subprocess. Then, you can access the subprocesses via the *getSubprocess* method.

If you are familiar with RapidMiner's integrated super operators like **Loop**, you know that there are always input ports on the left and output ports on the right of the subprocess. To distinguish these ports from the input and output ports of the super operator, RapidMiner calls them inner sources and inner sinks. In fact, an inner source is technically an output port for the super operator (because it delivers data to this port). The inner sink is an input port for the super operator from where it can retrieve the output of the subprocesses. To deliver outputs from the loop, you could add the following second variant of the *PortPairExtender* to collect the outputs from all iterations and pass them as a collection to the output of our super operator:

```
private final CollectingPortPairExtender outExtender =
    new CollectingPortPairExtender(
      "output", getSubprocess(0).getInnerSinks(), getOutputPorts());
```

This would result in an operator that looks like this:

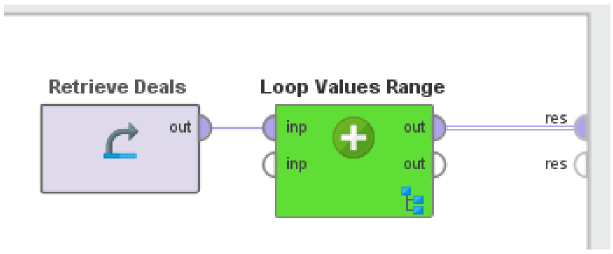


Figure 9.1: Super operator

To make a *PortExtender* work, you must initialize it during construction of the operator. Simply add the following lines in the constructor:

```
inputPortPairExtender.start();
outExtender.start();
```

To have proper meta data available at the output ports, add some rules:

```
getTransformer().addRule(inputPortPairExtender
    .makePassThroughRule());
```

You must add a rule defining when the subprocess' meta data is to be transformed. The ordering of the rule definition is crucial because if the meta data isn't forwarded to the inner ports, there is nothing the meta data transformation of the inner operators can do. This line adds the rule:

```
getTransformer().addRule(new SubprocessTransformRule(
    getSubprocess(0)));
```

Next, you need a rule to pass the meta data from the inner sinks to the output ports:

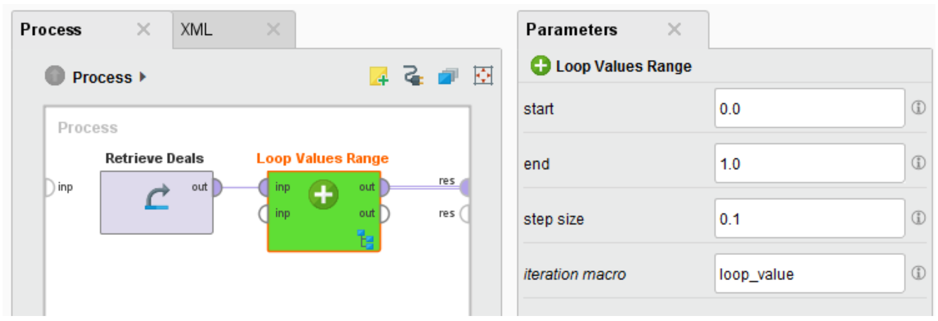
```
getTransformer().addRule(outExtender.makePassThroughRule());
```

The minimal setup of the `doWork()` method looks like this:

```
@Override
public void doWork() throws OperatorException {
     outExtender.reset();
     inputPortPairExtender.passDataThrough();
     getSubprocess(0).execute();
     outExtender.collect();
}
```

First, it resets the *CollectingPortPairExtender*, then it passes data from the input port of the super operator to the inner ports. Next, execute the subprocess and finally collect all outputs.

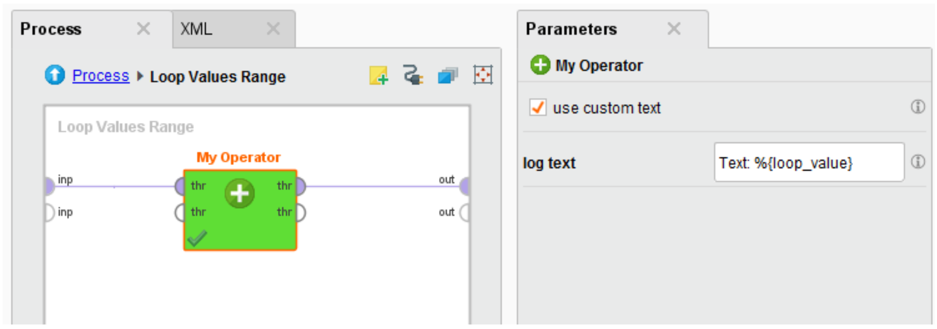
Try this instead. Add four parameters - the start value, the end value, the step size for the range, and a field where you can enter the macro name (which contains the current value during the execution of the loop). Then, adapt the `doWork()` method. Loop over the values in the given range, define the macro value in each iteration and execute the subprocess in each iteration. The final result looks like this:



start: 0.0
end: 1.0
step size: 0.1
iteration macro: loop_value

Figure 9.2: Super operator with parameters

You can see that the operator now has the parameters that define the value range of the loop. Within the subprocess you can read the macro value (the current value in the loop) and print it with the first simple operator that you created at the beginning.

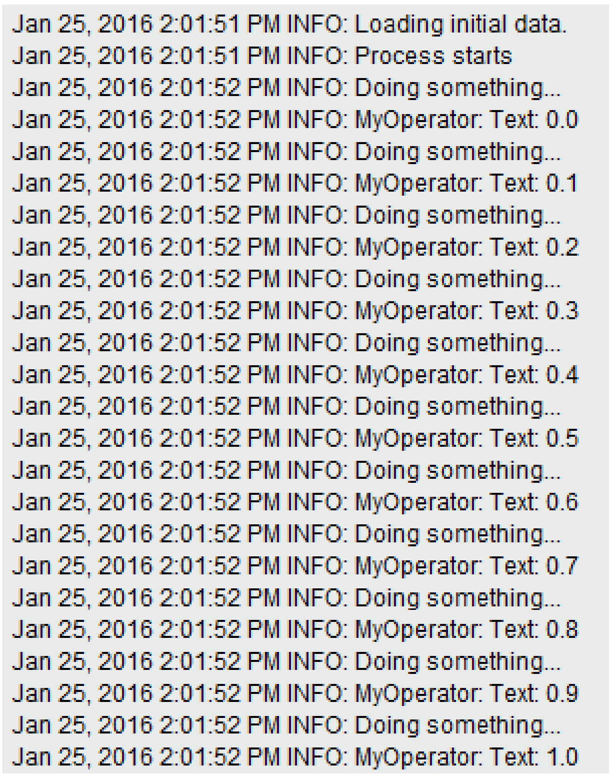


use custom text: ✓

log text: Text: %{loop_value}

Figure 9.3: Super operator's subprocess

The log entries show that, in each iteration, the value of the macro changes and the subprocess is executed.



```
Jan 25, 2016 2:01:51 PM INFO: Loading initial data.
Jan 25, 2016 2:01:51 PM INFO: Process starts
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.0
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.1
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.2
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.3
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.4
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.5
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.6
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.7
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.8
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 0.9
Jan 25, 2016 2:01:52 PM INFO: Doing something...
Jan 25, 2016 2:01:52 PM INFO: MyOperator: Text: 1.0
```

Figure 9.4: Super operator log entries

In the end, the code of the super operator looks like this:

```
/**
 * Example for a super operator, loops over values given
 * by a range and step size.
 */
public class LoopValuesRange extends OperatorChain{

    public static final String PARAMETER_START = "start";
    public static final String PARAMETER_END = "end";
    public static final String PARAMETER_STEP_SIZE = "step size";
    public static final String PARAMETER_MACRO_NAME =
        "iteration macro";

    private final PortPairExtender inputPortPairExtender =
        new PortPairExtender("input", getInputPorts(),
            getSubprocess(0).getInnerSources());

    private final CollectingPortPairExtender outExtender =
        new CollectingPortPairExtender("output",
            getSubprocess(0).getInnerSinks(), getOutputPorts());

    /**
     * Constructor
     * @param description
     */
     public LoopValuesRange(OperatorDescription description) {
         super(description, "Executed Process");
         inputPortPairExtender.start();
         outExtender.start();

         getTransformer().addRule(inputPortPairExtender
             .makePassThroughRule());
         getTransformer().addRule(new SubprocessTransformRule(
            getSubprocess(0)));
        getTransformer().addRule(outExtender
            .makePassThroughRule());

    }

    @Override
    public void doWork() throws OperatorException {

        outExtender.reset();
        inputPortPairExtender.passDataThrough();

        double start= getParameterAsDouble(PARAMETER_START);
        double end= getParameterAsDouble(PARAMETER_END);
        double stepsize= getParameterAsDouble(
            PARAMETER_STEP_SIZE);
        String macro= getParameterAsString(PARAMETER_MACRO_NAME);

        for(double i=start; i<end; i+=stepsize){

            getProcess().getMacroHandler().addMacro(macro,
                Double.toString(Math.round(i*100)/100.0));
            getSubprocess(0).execute();
        }
        outExtender.collect();
    }

    @Override
    public List<ParameterType> getParameterTypes() {
        List<ParameterType> types = super.getParameterTypes();

        types.add(new ParameterTypeDouble(PARAMETER_START,
            "start value of the value range",
            Integer.MIN_VALUE, Integer.MAX_VALUE, 0, false));
        types.add(new ParameterTypeDouble(PARAMETER_END,
            "end value of the value range",
            Integer.MIN_VALUE, Integer.MAX_VALUE, 1, false));

        types.add(new ParameterTypeDouble(PARAMETER_STEP_SIZE,
            "step size of the value range",
            0, Integer.MAX_VALUE, 0.1, false));

        types.add(new ParameterTypeString(PARAMETER_MACRO_NAME,
            "This parameter specifies the name"+
            " of the macro which holds the current value "+
            "of the selected range in each iteration.",
            "loop_value"));

        return types;
    }
}
```

9.3 Adding Your Own Data Objects

You might find that the standard data objects don't fulfil all your requirements. For example, maybe you are going to analyze data recorded from a game engine. The format of the original data can't directly be expressed as a table. Although you could write a single operator that reads in data from a file and does all the translation and feature extraction, you may decide, that it's best to split the task. Instead, create multiple operators - one that can handle the new data object and one that extracts part of the data as example set. With this modularity, it is much easier to extend the mechanism later on and optimize steps separately.

9.3.1 Defining the object class

First, you must define a new class that holds the information you need. This class implements the interface `IOObject`, but it is best to extend `ResultObjectAdapter` instead. The abstract class has already implemented much of the non-special functionality and is suitable for the most cases. In special circumstances, when you already have a class that might hold the data and provide some important functionality, it might be better to extend this class and let it implement the interface. An empty implementation would look like:

```
import com.rapidminer.operator.ResultObjectAdapter;

public class DemoDataIOObject extends ResultObjectAdapter {

    private static final long serialVersionUID = 1725159059345L;
}
```

The above is only an empty object that doesn't hold any information. Now, add some content:

```
import com.rapidminer.operator.ResultObjectAdapter;

public class DemoDataIOObject extends ResultObjectAdapter {

    private static final long serialVersionUID = 1725159059345L;

    private DemoData data;

    public DemoDataIOObject(DemoData data) {
        this.data = data;
    }

    public DemoData getDemoData() {
        return data;
    }
}
```

This class gives access to an object of the class `DemoData`, which is the representative for everything you want to access. While it might be more complex to implement in real-world applications, this example helps to illustrate how things work in general. You want to extract attribute values from the demo data, which an operator can then store in a table. You need a mechanism to add data to the IOObject. (For simplicity, assume you only have numerical attributes to save the effort of remembering the correct types of the data.) Add a map for storing the values with identifier as a local variable:

```
private Map<String, Double> valueMap = new HashMap<String, Double>();
```

Then we extend the DemoDataIOObject with two methods for accessing the map:

```
public void setValue(String identifier, double value) {
    valueMap.put(identifier, value);
}

public Map<String, Double> getValueMap() {
    return valueMap;
}
```

9.3.2 Configuration

To make your data object accessible, adapt the file `ioobjectsNAME.xml` (in the resources folder), which contains some examples of how to define your `IOObject`. This is how the `DemoDataIOObject` is defined:

```
<ioobjects>
```

```
    <ioobject
        name="DemoData"
        class="com.rapidminer.operator.demo.DemoDataIOObject"
        reportable="false">
        <renderer>com.rapidminer.gui.renderer.DefaultTextRenderer
        </renderer>
    </ioobject>
</ioobjects>
```

The renderer is a simple text renderer, which calls the `toString()` method of your `IOObject` to display the object in the `Results` perspective.

Similarly to operators, you can also give your own data objects a color. The connection between two operators that exchange your data object will be colored in the assigned color. To control colors, change the file `groupsNAME.properties` in the resources folder to add a line defining the color:

```
# IOObjects
io.com.rapidminer.operator.demo.DemoDataIOObject.color = #28C68C
```

9.3.3 Processing your own IOObjects

Using these methods, you can now implement your first operator, which extracts values of the `DemoData`.

```
import java.util.List;
import com.rapidminer.operator.Operator;
import com.rapidminer.operator.OperatorDescription;
import com.rapidminer.operator.OperatorException;
import com.rapidminer.operator.ports.OutputPort;

/**
```

```
 * Operator that generates {@link DemoData}
 */
public class GenerateDemoDataOperator extends Operator {

    private OutputPort outputPortDemoData =
        getOutputPorts().createPort("demo data");

    /**
     * Operator to create {@link DemoData}
     *
     * @param description
     *            of the operator
     */
   public GenerateDemoDataOperator(OperatorDescription description)
    {
        super(description);
        getTransformer().addGenerationRule(outputPortDemoData,
            DemoDataIOObject.class);
    }

    @Override
    public void doWork() throws OperatorException {

        DemoDataIOObject ioobject = new DemoDataIOObject(
            new DemoData());
        List<Double> values = ioobject.getDemoData().getValues();
        for (int i = 0; i < values.size(); i++) {
            ioobject.setValue("" + (i + 1), values.get(i));
        }
        outputPortDemoData.deliver(ioobject);
    }
}
```

The example fetches the values from the `DemoData` object and sets the values of the `IOObject`. Then, the output port delivers the `DemoDataIOObject`.

Of course, it's possible to build more complex constructions. You might, for example, use one super operator that handles your `DemoData` with different inner operators. You could build operators that get `DemoData` as input and transform them to an example set. Every method of treating your own IOObjects is possible by combining what we have learned.

9.3.4 Looking into your IOObject

When building a process for your `IOObjects`, it's an excellent idea to set breakpoints with the process and take a look at what's contained in the objects. To continue with the example above, it would be interesting to see which values have been extracted. If you set a breakpoint, RapidMiner will display the result of the `toString` method as the default fallback.

There is plenty of space you could fill with information about the object. How can you do this? The simplest approach is to override the `toString` method of the `IOObject`. However, it is better to override the `toResultString` method, which, by default, only calls the `toString` method.

Although text output has its advantages, writing Courier characters to the screen is a bit outdated. How do you add nice representations to the output as is done with nearly all core RapidMiner `IOObjects`?

RapidMiner uses a renderer concept for displaying the various types of `IOObjects`. So, you should implement a renderer for your `DemoDataIOObject`.

You must implement the `Renderer` interface for this purpose. Extend the `AbstractRenderer`, which has most of the methods already implemented. Most of the methods are used for handling parameters, since renderers might have parameters, just as operators do. They are used during automatic object reporting and control the output. The handling of these parameters and their values is done by the abstract class, you just need to take their values into account when rendering. Here are the methods to implement:

```
import java.awt.Component;
import com.rapidminer.gui.renderer.AbstractRenderer;
import com.rapidminer.operator.IOContainer;
import com.rapidminer.report.Reportable;

public class DemoDataRenderer extends AbstractRenderer {

    @Override
    public String getName() {
        return "DemoData";
    }

    @Override
    public Component getVisualizationComponent(Object renderable,
        IOContainer ioContainer) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
   public Reportable createReportable(Object renderable, IOContainer
        ioContainer, int desiredWidth, int desiredHeight) {
        // TODO Auto-generated method stub
        return null;
    }
}
```

The first method must return an object of a class implementing one of the sub interfaces of `Reportable`, but this should not be handled here. For an example, look at the interfaces and some of the implementations in the core.

The second method returns an arbitrary Java component used for displaying content in Swing. While there is great flexibility, because you want to see the values as a table, render it as such. You don't have to implement everything yourself, though. You can use a subclass of the `AbstractRenderer` - the

AbstractTableModelTableRenderer. As the name indicates, it shows a table based upon a table model. All you need to do is to return this table model:

```
import java.util.ArrayList;
import java.util.List;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import com.rapidminer.gui.renderer.AbstractTableModelTableRenderer;
import com.rapidminer.operator.IOContainer;
import com.rapidminer.operator.demo.DemoDataIOObject;
import com.rapidminer.tools.container.Pair;

public class NewDemoDataRenderer extends
    AbstractTableModelTableRenderer {

    @Override
    public String getName() {
        return "DemoData";
    }

    @Override
    public TableModel getTableModel(Object renderable,
        IOContainer ioContainer, boolean isReporting) {
        if (renderable instanceof DemoDataIOObject) {
            DemoDataIOObject object = (DemoDataIOObject)
                renderable;
            final List<Pair<String, Double>> values =
                new ArrayList<>();
            for (String key : object.getValueMap().keySet()) {
                values.add(new Pair<String, Double>(key,
                    object.getValueMap().get(key)));
            }
            return new AbstractTableModel() {

                private static final long serialVersionUID = 1L;

                @Override
                public int getColumnCount() {
                    return 2;
                }

                @Override
                public String getColumnName(int column) {
                    if (column == 0) {
                        return "Id";
                    }
                    return "Value";
                }

                @Override
                public int getRowCount() {
                    return values.size();
                }

                @Override
                public Object getValueAt(int rowIndex,
                    int columnIndex) {
                    Pair<String, Double> pair =
                        values.get(rowIndex);
                    if (columnIndex == 0) {
                        return pair.getFirst();
                    }
                    return pair.getSecond();
                }
            };
        }
```

```
        return new DefaultTableModel();
    }
}
```

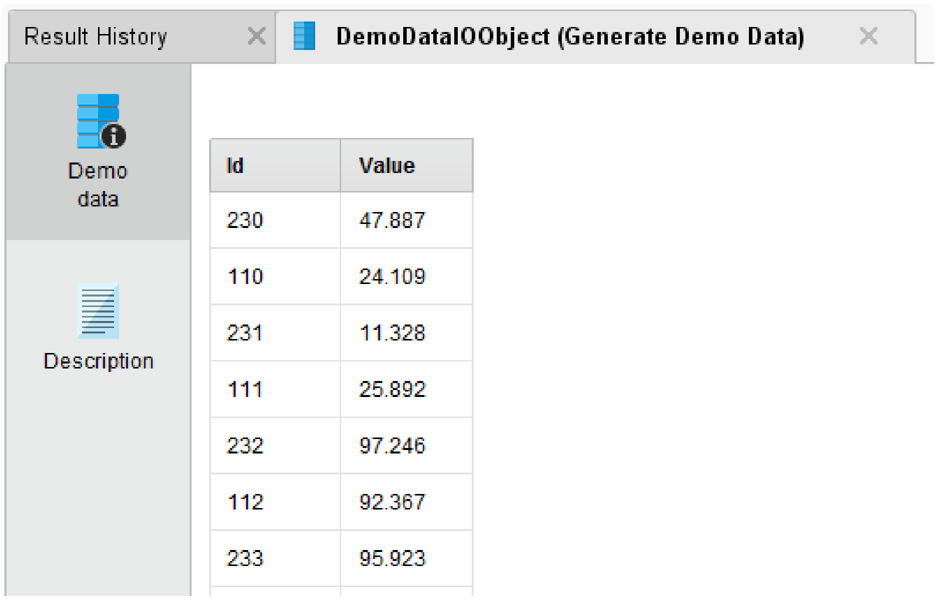
There are other convenience methods in the `AbstractTableModelTableRenderer` for changing the appearance of the table. For example, the following methods change the behavior of the table by enabling or disabling certain features:

```
@Override
public boolean isSortable() {
    return true;
}

@Override
public boolean isAutoresize() {
    return false;
}

@Override
public boolean isColumnMovable() {
    return true;
}
```

To use the new renderer, you must change the file `ioobjectsNAME.xml` in the resources folder again. Just add the renderer before the default renderer:

```
<ioobjects>
    <ioobject
        name="DemoData"
        class="com.rapidminer.operator.demo.DemoDataIOObject"
        reportable="false">
        <renderer>com.rapidminer.gui.renderer.demo.DemoDataRenderer
        </renderer>
        <renderer>com.rapidminer.gui.renderer.DefaultTextRenderer
        </renderer>
    </ioobject>
</ioobjects>
```

You can now see the result of your efforts in building a table representation of the DemoData values.



Id	Value
230	47.887
110	24.109
231	11.328
111	25.892
232	97.246
112	92.367
233	95.923

9.4 Creating Custom Configurators

Imagine that you want to create a RapidMiner extension that offers an operator for reading data from a CRM system. Your operator needs the information for accessing the CRM, such as a URL, a user name, or a password. One approach is to add text fields to the parameters of the operator and let the user type in the required information. Though this may seem convenient, it gets quite redundant if you want to use the same information in other RapidMiner processes or operators, since you would have to enter the information multiple times. Alternatively, you can define the CRM connection globally and let the user select which CRM to get data from.

This is a scenario where `Configurators` come in handy. A configurator globally manages items of a certain type globally and allows you to create, edit and delete them through a custom configuration dialog. For this example, we will implement a configurator for CRM entries that allows us to automatically configure those entries using a dialog accessible through the **Connections** menu. Moreover, a configurator can be used along with a drop-down list, which allows the user to easily select a connection via a parameter of your operator.

9.4.1 Usage

In order to implement your own configurator, you need to know the following classes:

- **Configurable** is an item that can be modified through a *Configurator*.
- **Configurator** instantiates and configures subclasses of *Configurable*.
- **ConfigurationManager** is used to register *Configurators* in RapidMiner.
- **ParameterTypeConfigurable** is a *ParameterType* that creates a drop-down list for configurators and can be used in the parameter settings of operators.

First, create a new class describing a single CRM connection entry, which implements the *Configurable* interface. Best practice is to extend *AbstractConfigurable* instead, because by doing so, you avoid dealing with parameter values. Then, you don't have to write code that deals with the actual configuration:

```
public class CRMConfigurable extends AbstractConfigurable {

    @Override
    public String getTypeId() {
        return "CRM";
    }

    /** Actual business logic of this configurable */
    public CRMConnection connect() {
        String username = getParameter("user name");
        String url = getParameter("URL");
        URLConnection con = new URL(url).openConnection();
        // ...
        // do something with the connection
    }
}
```

Next, we have to extend the *AbstractConfigurator* class. Each configurator has a unique *typeId*, a String in order to identify the configurator in RapidMiner and an I18NBaseKey which will be used as the base key for retrieving localized information from the resource file. Also, we want to add some *ParameterTypes* to our Configurator, because they specify how an entry can be edited through the configuration dialog. In our example, we need *Parameter Types* describing the URL, the user name and password which should be used for the CRM connection. For that matter, you would simply have to overwrite the `getParameterTypes` method and add a new `ParameterTypeString`, as shown in the following implementation:

```
public class CRMConfigurator
    extends AbstractConfigurator<CRMConfigurable> {

    @Override
    public Class<CRMConfigurable> getConfigurableClass() {
        return CRMConfigurable.class;
    }

    @Override
    public String getI18NBaseKey() {
        return "crmconfig";
    }

    @Override
    public List<ParameterType> getParameterTypes(
        ParameterHandler parameterHandler) {

        List<ParameterType> parameters =
            new ArrayList<ParameterType>();
        parameters.add(new ParameterTypeString("URL",
            "The URL to connect to", false));
        parameters.add(new ParameterTypeString("Username",
            "The user name for the CRM", false));
        parameters.add(new ParameterTypePassword("Password",
            "The password for the CRM"));
        return parameters;
    }

    @Override
    public String getTypeId() {
        return "CRM";
    }
}
```

In addition to the methods `getTypeId`, `getI18NBaseKey` and `getParameterTypes`, you must also implement the method `getConfigurableClass`, which simply returns the used Configurable implementation class (in this case, the class `CRMConfigurable`).

Next, add localized information to the resource file *GUIXXX.xml*, where 'XXX' is the extension name.

```
gui.configurable.crmconfig.name = CRM Connection
gui.configurable.crmconfig.description = An entry describing a
    CRM connection.
gui.configurable.crmconfig.icon = data.png
```

To get access to the new configurator, register it in the *ConfigurationManager*. This step is important because it let's RapidMiner know of the new configurator so that the CRM operator and other parts of RapidMiner can access it. To do this, simply call the *register* method within the initialization procedure. This should be done in the *initPlugin* method of the *PluginInit* class:

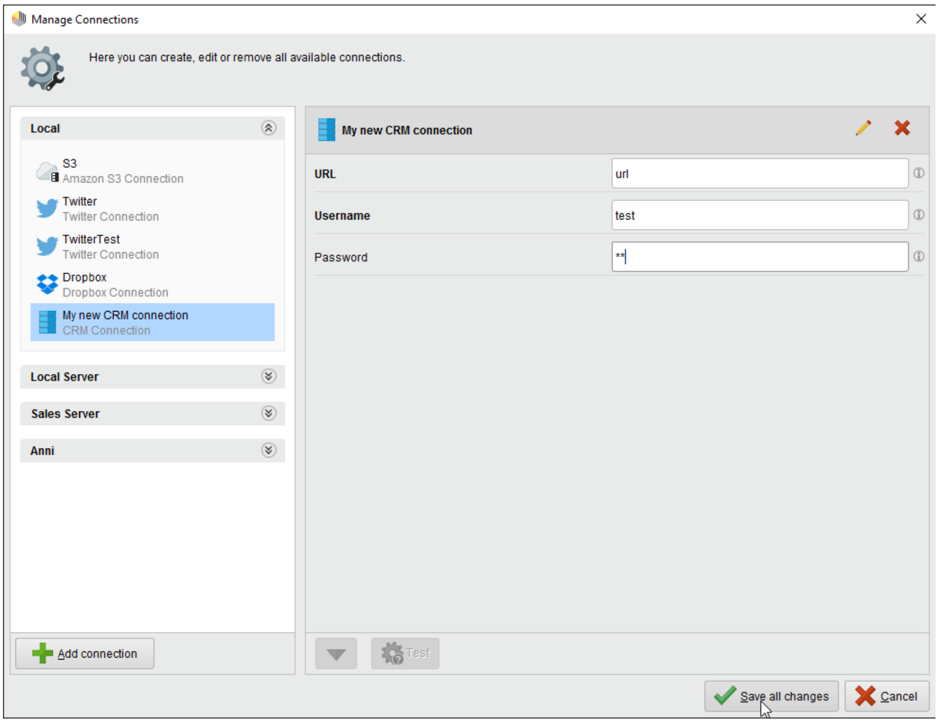
```
public static void initPlugin() {
    ConfigurationManager.getInstance().register(
        new CRMConfigurator());
}
```

You can now open the **Manage Connections** dialog (see Figure 9.5) in the **Connections** menu and create a CRM connection.

Now, use the connection as a parameter of an operator (see Figure 9.6) that connects to the CRM. To do so, override the method `getParameterTypes` and add
`ParameterTypeConfigurable`:

```
@Override
public List<ParameterType> getParameterTypes() {
    List<ParameterType> types = super.getParameterTypes();
    ParameterType type = new ParameterTypeConfigurable(
        PARAMETER_CONFIG, "Choose a CRM connection", "CRM");
    types.add(type);
    return types;
}
```

You have now successfully created your own configurator and can use it to configure CRM entries for your operator.



URL	url
Username	test
Password	**

Figure 9.5: Manage connections dialog

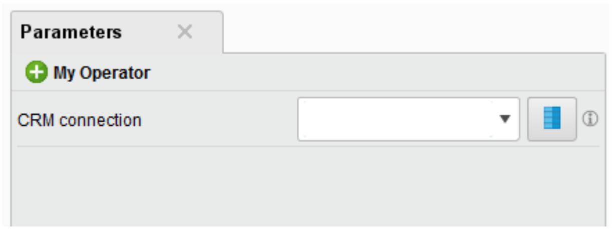


Figure 9.6: Parameter View with Configurable as parameter.

9.5 Adding Custom User Interface Elements

Learn how to add custom panels to the UI of RapidMiner Studio, how to use custom actions, and how to modify the **Preferences** dialog to your needs.

9.5.1 Adding custom panels

The PluginInit class offers the ability to modify the GUI. We will add a single new window here for demonstration purpose. All we have to do is to implement a new class implementing the *Dockable* interface and a component that is delivered by the *Dockable*.

```
public class SimpleWindow extends JPanel implements Dockable {

    private static final long serialVersionUID = 1L;
    private static final DockKey DOCK_KEY = new ResourceDockKey(
        "tutorial.simple_window");
    private JLabel label = new JLabel("Hello user.");

    /**
     * Constructor for a {@link SimpleWindow}.
     */
    public SimpleWindow() {
        // adding content to this window
        setLayout(new BorderLayout());
        add(label, BorderLayout.CENTER);
    }

    /**
     * Sets the simple window label
     *
     * @param labelText
     *            the text to show
     */
    public void setLabel(String labelText) {
        this.label.setText(labelText);
        System.out.println(labelText);
        revalidate();
    }

    @Override
    public Component getComponent() {
        return this;
    }

    @Override
    public DockKey getDockKey() {
        return DOCK_KEY;
    }
}
```

While the content of the window is rather simple and only a variant of the well-known 'Hello world' program, it illustrates the concept of the `ResourceDockKey`. A `DockKey` contains information about a *Dockable*, for example it stores the name and the icon of the window. The `ResourceDockKey` retrieves this information from the GUI resource bundle that is loaded in a language dependent manner from the resource file *GUIXXX.xml* where 'XXX' is the extension name. The following is an example of what might describe the new window:

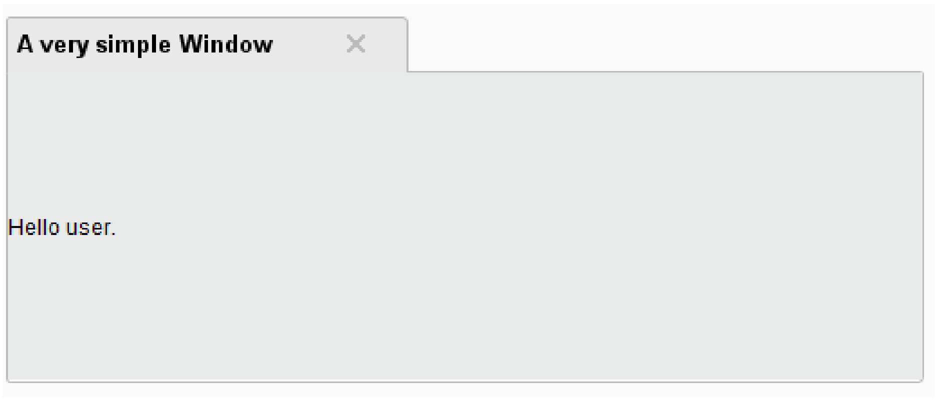
```
gui.dockkey.tutorial.simple_window.name = A simple Window
gui.dockkey.tutorial.simple_window.icon = window2.png
gui.dockkey.tutorial.simple_window.tip = Take a look at what
    RapidMiner has to say.
```

In the example, the icon *window2.png* has been added to the folder `icons/16` in the `resources` folder of your extension, making it available when starting

RapidMiner. The final task before seeing the new window is to register it at RapidMiner's *MainFrame*. You want to do this independent of operator execution, and in fact, want to have the window before any process is executed. To do so, use one of the *PluginInit* hooks, so we are going to fill the *initGui* method:

```
public static void initGui(MainFrame mainframe) {
    mainframe.getDockingDesktop().registerDockable(
        new SimpleWindow(););
}
```

That's it! You can now select the new panel from the **View > Show Panel** menu. The result looks like this:



9.5.2 Adding custom settings to the Preferences dialog

Open the **Settings > Preferences** menu and look at the existing preferences dialog. As you can see, there are several tabs that contain specific settings that the user can make. For your extension, you can create your own tab.

Complete these three steps to build your own tab in the **Preferences** dialog:

1. Adapt the file `settingsNAME.xml`, where 'NAME' is the name of your extension. Then, specify keys for the tab name and the single preferences you want to add. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
   <group key="extension_name">
       <property key="extension_name.url" />
   </group>
</settings>
```

2. Adapt the file `SettingsNAME.properties`, where 'NAME' is again the name of your extension. Specify names for the keys that you defined in the first step.

```
extension_name.title = My Extension
extension_name.description =

extension_name.url.title = A Server URL
extension_name.url.description = Default value of a server URL
```

3. Change the method `initPlugin()` in your `PluginInit` class and register the settings that you want to add to the **Preferences** dialog.

```
public static void initPlugin() {

   ParameterService.registerParameter(
       new ParameterTypeString(
           "extension_name.url", "The server URL",
               "http://localhost:8080"));
}
```

If you want to use a boolean setting (a check box), register it like this:

```
ParameterService.registerParameter(
new ParameterTypeBoolean("extension_name.url","Use a URL?", false));
```