Savant Genome Browser: Plugin Development Guide

February 23, 2012

Authors: Marc Fiume & Eric Smith
Contact: savant@cs.toronto.edu
Website: http://savantbrowser.com

This document applies to Savant SDK version 2.0.0

Contents

Chapter 1

Introduction

Savant is unique in the Genome Browser arena in that it was designed to be extensible through a rich plugin framework, which allows developers to provide functionality in addition to what is provided with the standard distribution.

A wide range of Savant plugins have already been written. GUI plugins, like the Amino Acid plugin and the SNP Finder, enhance the visualisation of data within Savant (see Chapter 4). DataSource plugins, like the SQL plugin and the UCSC plugin, extend Savant by providing access to new sources of data (Chapter 5). Tool plugins, like the GATK tool, extend Savant by providing integrated access to third-party analytical tools (Chapter 6).

Chapter 2

Tutorial: Developing a Savant Plugin

In this tutorial, we describe the steps used to create the Savant Amino plugin using the demo project included in the Savant SDK as a template.

2.1 The Amino Plugin

The Savant Amino plugin was designed to demonstrate just some aspects of the API: in particular, accessing basic information about tracks and rendering it within Savant. It does not cover much of what is offered through the API (for example, creating custom data-sources or visualisations). Most importantly, this tutorial was designed to help you create a foundation for your own plugin project.

2.2 Prerequisites

- JDK

- NetBeans
  Note: for this tutorial, it is recommended to download the Netbeans + JDK Bundle from oracle.com.
- Savant SDK

2.3 Setting Up the NetBeans Project

1. Set up NetBeans and the JDK

Download NetBeans and the JDK and install them. For simplicity it is recommended to download the bundle, but you can certainly download and install them independently if you wish.

2. Set up the Savant SDK

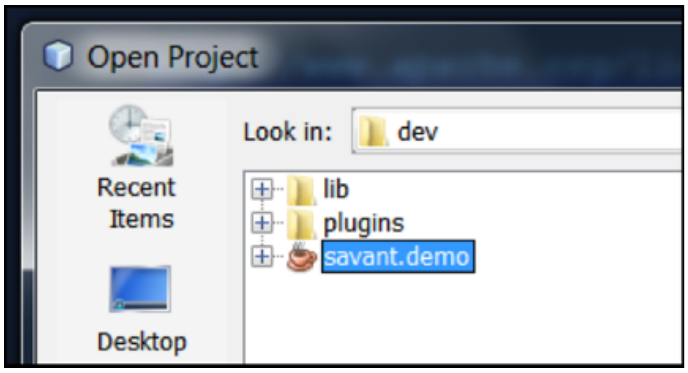
Download the Savant SDK, listed above, and unzip it. The SDK contains the following components:

- api – documentation of the Savant code
- dev – development files
- samples – sample projects
- this file

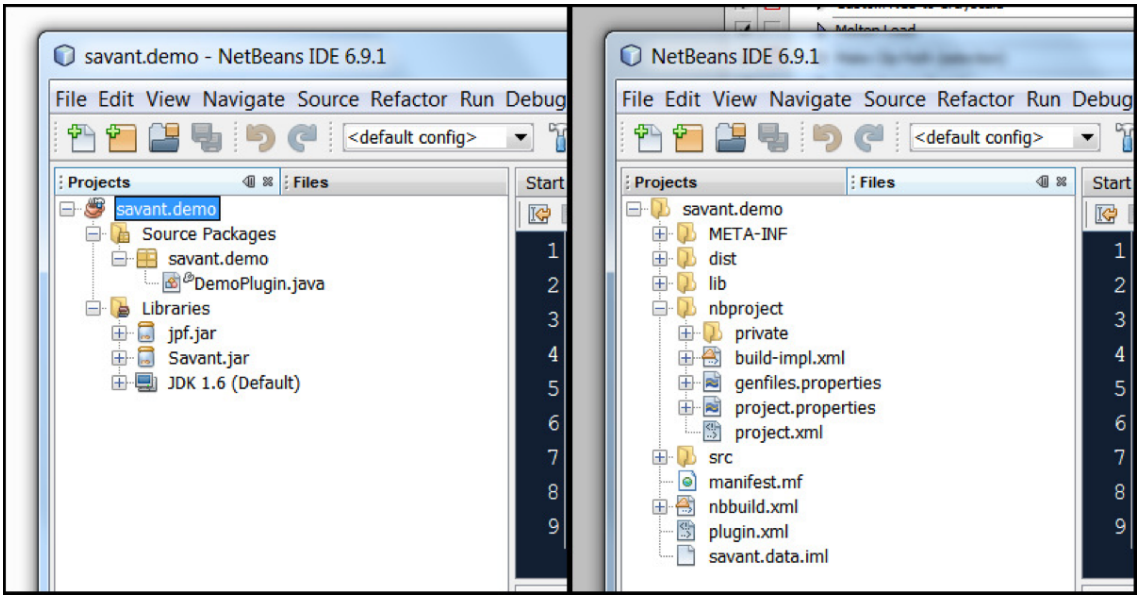
For this tutorial, we will refer to the dev folder in the SDK as dev/.

3. Open the Demo Project

Open NetBeans. Open the project in dev/savant.demo.



The “Project” and “Files” views should look something like this:



Description of Important Project Files

- Project View
  - Source Packages – contains implementation of this plugin
    - Main package (savant.demo) – contains the main class of the plugin
    - Main class (DemoPlugin.java) – the class which implements the Savant Plugin interface
  - Libraries – contains links to external code packages
    - Savant.jar – Savant API, required for accessing Savant functions
- Files View
  - lib/ – contains additional libraries used by this plugin
  - nbproject/project.properties – contains NetBeans project properties
  - nbbuild.xml – contains Ant instructions for building the project
  - plugin.xml – contains information about the plugin and how to run it

2.4 Editing the Demo Project

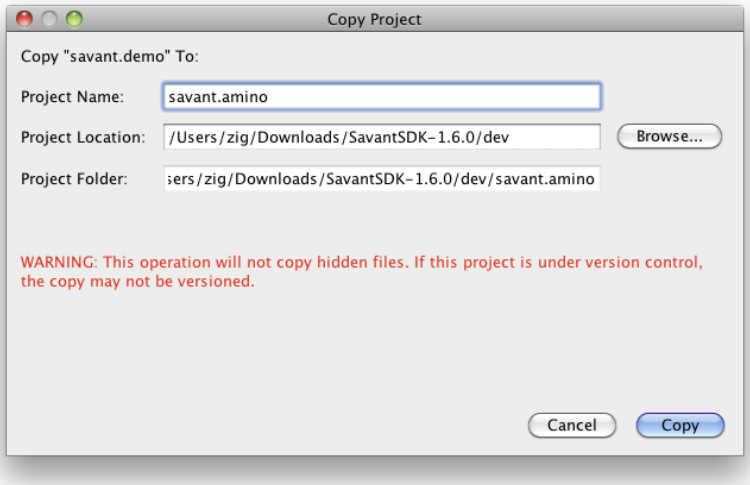
You will need to know the following properties of your plugin:

- *NAME* – the name of the plugin.
- *VERSION* – the version of the plugin. In Savant we use the following scheme for version numbers [major version] . [minor version] . [bug fix / build number].
- *PACKAGE* – the main package of the plugin (which contains the main class).
- *CLASS* – the main class of the plugin.

In this example, *NAME* = "Amino Plugin", *VERSION* = "1.0.0", *PACKAGE* = "savant.amino", and *CLASS* = "AminoPlugin".

1. Copy the Demo Project

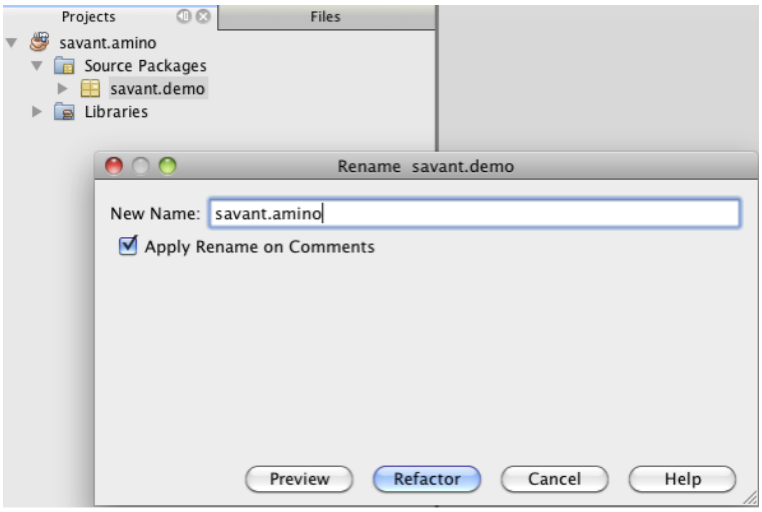
Right-click the savant.demo project icon (looks like a coffee cup) and choose "Copy…". The copy dialog will be presented. In the Project Name field, enter *PACKAGE*. Click the "Copy" button.



This will create a copy of the demo project which we will now edit to create a new plugin.

2. Rename the main package

Right-click the savant.demo package icon (looks like a cardboard box) and choose "Refactor > Rename…". In the New Name field, enter *PACKAGE*. Click the "Refactor" button.



3. Rename the main class

Right-click the DemoPlugin.java icon and choose "Refactor > Rename…". In the New Name field, enter *CLASS*. Click the "Refactor" button.

4. Change the plugin name

Double-click the *CLASS*.java file (formerly DemoPlugin.java) to open it in the Editor. Change the `getTitle()` function to return "*NAME*".

```
@Override
public String getTitle() {
    return "Amino Plugin";
}
```

5. Edit the plugin properties

Switch to the "Files" view (the tab beside the "Projects" tab). Expand the new project. Double-click the plugin.xml file to open it in the editor. There are four places that need to be edited in this file, denoted in red boxes in the following figure:

```
<?xml version="1.0" ?>
<plugin id="savant.demo" version="1.0.1" class="savant.demo.DemoPlugin" sdk-version="2.0.0" name="Demo Plugin"/>
```

Here is how the file will be changed in the savant.amino example:

```
<?xml version="1.0" ?>
<plugin id="savant.amino" version="1.0.0" class="savant.amino.AminoPlugin" sdk-version="2.0.0" name="Amino Acid Plugin"/>
```

6. Change the project name

Open the nbbuild.xml file in the editor (it's in the same directory as plugin.xml). Change the name attribute of the project to *PACKAGE*.

```
<project name="savant.amino" default="default" basedir=".">
    <description>Builds, tests, and runs the project savant.amino.</description>
    <import file="nbproject/build-impl.xml"/>
```

At this point, we have created an independent plugin project which can be built and loaded into Savant. It is a good idea to try to do so now. See the section on building plugins (§3.1).

2.5 Code Away!

You now have an independent plugin project as a foundation for your own plugin. You have two main tasks, outlined below. If you are following the savant.amino tutorial, you should download the complete savant.amino NetBeans Project to a separate location and copy the Java files in the dev/src directory of that package to the dev/src directory of your NetBeans project (overwriting AminoPlugin.java).

In general, the remaining tasks are:

1. Implementing the `init()` function

The `init()` function is called immediately when the plugin is loaded as Savant starts. The init function is responsible for initializing (among other things) the graphical components of the plugin (e.g. buttons, menus, etc.). The graphical "canvas" for plugins is a JPanel provided as an argument to the `init()` function. For help with JPanel and other Swing components visit http://download.oracle.com/javase/tutorial/uiswing.

2. Implementing the rest of your plugin

Go nuts, you are free to do whatever you like in your plugin!

Suggested guidelines:

Thread long running tasks

Your plugin will run in the main Savant thread unless you tell it to do otherwise. This means that if your plugin does something which takes a long time (e.g. downloads files, performs a long computation, etc.) it will lock up the main thread until it has finished. Please perform such tasks in a separate thread.

Try to stick to what's in the Savant API packages

The `savant.api.adapter` and `savant.api.util` packages contains classes and methods which the Savant development team considers useful for developers to have. It is possible a developer requires some more functionality that can be found by getting access to a Savant instance. Please notify us of these situations so that we can consciously include such functionality in subsequent versions of the API, otherwise there is a risk that this functionality will be destroyed (since we're not expecting anyone to be using it directly) and your plugin will no longer work.

2.6 Submit Your Plugin

We strongly encourage you to submit all your developed Savant plugins to us, so that we can make them directly available to all Savant users through the Savant Plugin Repository. This is both a great way to promote your own work while encouraging more users and developers to join the Savant Community! You can submit your plugins to us through our plugin submission form: http://savantbrowser.com/plugins.php#submit.

Chapter 3

Plugin Development Tips and Tricks

3.1 Building and Testing Plugins

To build your plugin, right-click the plugin project icon (the coffee cup) and choose "Build". NetBeans should output something like this:

```
deps-jar:
Created dir: /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/out/production/savant.demo
Created dir: /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/out/empty
Compiling 1 source file to /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/out/production/savant.demo
compile:
Copying 1 file to /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/out/production/savant.demo
Created dir: /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins
Copy libraries to /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins/lib.
Building jar: /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins/savant.demo-1.0.0.jar
Building jar: /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins/savant.demo-1.0.0.jar-all
Deleting directory /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins/lib
Moving 1 file to /Users/zig/Downloads/SavantSDK-2.0.0-prerelease/dev/plugins
jar:
BUILD SUCCESSFUL (total time: 0 seconds)
```

What's important is that it builds successfully. Look in the dev/plugins directory. You should see a .jar file corresponding to your plugin (in the savant.amino example, it is savant.amino-1.0.0.jar).

Now that you've ensured that the plugin was built, you can test out the plugin by running the provided dev/testPlugins.sh script or by running the Savant.jar otherwise (e.g. in Windows, you can just double-click Savant.jar).

3.2 Including Other Libraries

You may include additional libraries in your plugin. To do so:

1. Place the jar file for the external library in the dev/lib folder.
2. In your NetBeans project, right-click "Libraries", then choose "Add JAR/Folder...", then find the jar

file and click “Open”.

3. Open the nbbuild.xml file in the “Files” view in NetBeans, and add a new `zipfileset` entry under the previous `zipfileset` entry (near the bottom of the file) for your jar file:

```
<target name="-post-jar">
    <jar jarfile="${dist.jar}-all">
        <zipfileset src="${dist.jar}" excludes="META-INF/*" />
        <!-- Add this line. Replace "nameofjar" with actual file-name. -->
        <zipfileset src="lib/nameofjar.jar" excludes="META-INF/*" />
    </jar>
    <delete dir="${dist.dir}/lib"/>
    <move file="${dist.jar}-all" tofile="${dist.jar}"/>
</target>
```

Chapter 4

Sample GUI Plugin: savant.amino

In the samples/savant.amino directory of the SDK download, you will find the completed savant.amino plugin. The code is divided into three files: AminoPlugin.java, AminoAcid.java, and AminoCanvas.java.

The Amino Acid plugin is an example of a GUI plugin which renders extra information in a layer on top of one of the existing tracks.

4.1 AminoPlugin.java

The actual class which implements the Savant plugin. Since this is a GUI plugin, the class it extends is `SavantPanelPlugin`. The `AminoPlugin` class has only three methods of note: `init()`, `getTitle()`, and `getAlpha()`. The first two of these methods are required by the `SavantPanelPlugin` interface.

4.1.1 `AminoPlugin.init()`

Initialises the plugin's interface panel. This panel is what the user will see upon selecting the plugin's tab at the bottom of the Savant window. What the plugin puts in this panel is entirely up to you. For some plugins, like the built-in Data Table plugin, the panel is the essential part of the plugin, for others the panel just provides a way of configuring the plugin's settings. Since the savant.amino plugin has no real settings to configure, the panel contains only a legend describing the colour scheme used by the plugin.

The `init()` function also gives the plugin an opportunity to register listeners for Savant events it might be interested in. In this case, the savant.amino plugin listens for `TrackEvents` and `LocationChangedEvents`. The former gives the plugin a chance to create an `AminoCanvas` on which to draw the track's amino acids; the latter lets the plugin know when the layer canvas needs to be repainted.

4.1.2 `AminoPlugin.getTitle()`

This required method provides the title for the plugin's tab in the Savant user interface.

4.1.3 `AminoPlugin.getAlpha()`

An example of how a plugin can access persistent settings stored in the savant.settings file. As an exercise, you can add a `JSlider` to the plugin's panel and use `SettingsUtils.setInt()` to save the value programmatically. For the time being, changing the settings will require adding a line like "savant.amino.ALPHA=40" to your ~/.savant/savant.settings file.

4.2 AminoAcid.java

This enum defines the constants associated with each amino acid, and sets up a lookup table. The colour scheme is taken from RasMol, which assigns colours according to traditional amino acid properties (e.g. aspartic acid and glutamic acid are acidic and coloured red, while arginine, histidine, and lysine are basic and coloured in bluish shades).

4.3 AminoCanvas.java

This class does the work of actually drawing the amino acids. The plugin creates an `AminoCanvas` when it receives a `TrackEvent.Type.ADDED` event from Savant, and adds the newly-created canvas to the track's layer-canvas. Since the `AminoCanvas` will be drawn on top of the track itself, it calls `setOpaque(false)` to ensure that it doesn't completely obscure the track.

The plugin converts between base positions and pixel values using the `TrackAdapter`'s `transformXPos()` method. Information about where the starts and ends of coding regions is retrieved using the track's `getDataInRange()` method, which returns a list of records. In this case, since the plugin only attaches an `AminoCanvas` to tracks with data-format `RICH_INTERVAL`, we can assume that it's safe to cast these records to `RichIntervalRecord`.

The actual sequence data is retrieved using the `GenomeAdapter` class. The rest of the plugin's logic is devoted to figuring out where the codons start and end.

Chapter 5

Sample DataSource Plugin: savant.diff

The savant.diff plugin is intended to demonstrate how to write a data-source plugin. The data-source plugins most familiar to Savant users are probably the UCSC and SQL plugins, which let Savant display data retrieved from a relational database. For the purposes of the SDK, we will present a much simpler plugin, which calculates the difference between two continuous tracks and displays the result as a third such track.

5.1 DiffPlugin.java

The actual class which implements the Savant plugin. Since this is a data-source plugin, it is derived from `SavantDataSourcePlugin`. The `init()` and `getTitle()` methods are required by the plugin interface, but don't do anything particularly interesting. The actual work is done in the two `getDataSource()` methods.

5.1.1 `DiffPlugin.getDataSource()`

The niladic version of `getDataSource()` is called by Savant when the user chooses the *Load Track from Other DataSource* option from the *File* menu. In the case of the Diff plugin, this just presents a dialog to let the user select which two continuous tracks will serve as inputs for the plugin.

More elaborate plugins, such as the UCSC plugin, use this method as an opportunity to log into a database and configure a session. Regardless of the complexity of this configuration process, the results should all be distilled into a single URI which contains sufficient information to save and restore the session. In the case of the Diff plugin this URI consists of the scheme `diff://` followed by the URIs for the two input tracks, enclosed in brackets and separated by a semicolon. Together this provides enough information for the plugin to reconstitute the Diff track later.

5.1.2 `DiffPlugin.getDataSource(URI)`

This version of the `getDataSource()` method is used when opening a project or using the *Recent Tracks* menu. It takes the `diff://` URI and determines which two tracks to use as inputs. If the Diff plugin were production code, it would include logic to open the input tracks if they were not already loaded. However, since this plugin is just intended as sample code, that extra logic has been omitted for the sake of clarity.

5.2 SourceDialog.java

Provides a simple user interface that lets the user select which tracks to use as inputs for the Diff track. The track lists are populated using `TrackUtils.getTracks(DataFormat.CONTINUOUS)`, which provides an array of all loaded continuous tracks, including both ordinary continuous tracks as well as coverage tracks.

5.3 DiffDataSource.java

This class provides the plugin's implementation of the `DataSourceAdapter` interface. Savant uses parameterised types for its data-sources, so the actual interface implemented is `DataSourceAdapter<ContinuousRecord>`. The `ContinuousRecord` interface consists of just a position, a reference (i.e. chromosome), and a floating-point value.

The `DataSourceAdapter` interface requires nine methods to be implemented, but seven of them are trivial. The `getDataFormat()` and `getColumnNames()` methods just return constants which are appropriate for continuous tracks. The `getURI()` method returns the URI which was set by the track's constructor. For want of anything better to return, the plugin just returns the track's URI as the value for `getName()`. The `loadDictionary()` and `lookup()` methods are intended for tracks which have named features (e.g. gene tracks); for a continuous track these methods can just be stubs. Similarly, the Diff plugin has no cleanup to perform, so the `close()` method can also just be a stub.

5.3.1 `DiffDataSource.getReferenceNames()`

The `getReferenceNames()` method is used by Savant to determine which references a track has data for. For a Diff track, this is essentially the intersection of the references for its two inputs. In practice, some tracks store references as "chr1", "chr2", etc., while others use "1", "2", which means that `getReferenceNames()` needs to have a bit of extra logic to homogenise these two representations.

5.3.2 `DiffDataSource.getRecords()`

The key method of any data-source is `getRecords()`. Savant passes the plugin a reference, a range, and a resolution, and the data-source is supposed to return a list of data records, ordered by ascending position.

In the case of the Diff plugin, this just amounts to calling `getRecords()` for the two input tracks and calculating the difference between the values at each position. This is complicated somewhat by the fact that there is no guarantee that the input data-sources will return a data-record for every position (particularly at lower resolutions), hence the need for the `interpolate()` method.

Chapter 6

Sample Tool Plugin: srma.xml

Tool plugins are a new variety of plugins which was introduced in Savant 2. They provide a way to wrap an external application so that it can be invoked from within Savant.

A tool plugin is defined by an XML file which describes how to invoke the external application, how to specify its arguments, and how to interpret its output. The format of this XML file is derived from the plugin.xml file used for other Savant plugins. Other Savant plugins embed the plugin.xml file inside a JAR file, but tool plugins consist solely of the bare XML file.

To the end-user, a tool plugin looks much like an ordinary GUI plugin. A tool has a user-interface which is presented in a tab at the lower left of the Savant window. The various command-line arguments can be configured using that user-interface. Clicking the "Execute" button will launch the external application, and the application's progress and status will be displayed in the tool's panel.

The actual installation of the external application lies outside the scope of this document. The sample tool plugin included with the SDK executes SRMA (Homer and Nelson 2010), which happens to be a Java application. However, a tool plugin can equally well be used to invoke a platform-specific binary.

6.1 <plugin> Element

The top-level element of a tool plugin's XML file is the <plugin> element. Like the <plugin> element of a plugin.xml file, this element has attributes for `id`, `version`, `class`, `sdk-version`, and `name`. The only special requirement for a tool plugin is that the `class` attribute be `savant.plugin.Tool`.

```
<plugin id="savant.srma" version="1.0.0" class="savant.plugin.Tool" sdk-version="2.0.0" name="SR
MA (Short Read Micro re-Aligner)">
```

6.2 <tool> Element

The tool's XML file should contain a single <tool> element, defining the command-line which will be executed to invoke the tool.

In the case of the SRMA sample, the command line launches Java to invoke the srma JAR file. Here, the actual srma-0.1.15.jar[1] file is expected to be in the Savant plugins directory (~/.savant/plugins). Any invariant command-line options can also be specified in the <tool> element.

6.3 <arg> Elements

Following the <tool> element can be any number of <arg> elements. Each <arg> tag describes a single command-line argument.

The `flag` attribute indicates the text of the flag as it will be inserted onto the external application's command-line. To make it clearer, the actual command-line is displayed at the top of the plugin's panel as the user manipulates the user-interface.

The user-interface is generated programmatically using the `name` and `type` attributes. The `name` provides a user-friendly name which will be used when presenting the argument in the user-interface. The argument's appearance in the user-interface is controlled by the `type` attribute, as described in the following table.

Type	Presentation
`BAM_INPUT_FILE`	Menu listing all open alignment tracks
`FASTA_INPUT_FILE`	Menu listing all open sequence tracks
`OUTPUT_FILE`	Text-field
`RANGE`	Savant-style range specification (e.g. "chr1:1-1000")
`BOOL`	Check-box
`INT`	Text-field to enter integer value
`FLOAT`	Text-field to enter floating-point number
`LIST`	Uses <arg>'s `choices` attribute to populate a menu. Choices are expected to be comma-separated.
`MULTI`	Uses `choices` attribute to populate a grid of check-boxes. Allows user to select multiple options.

As it happens, there are no `type="MULTI"` arguments for the SRMA sample, but here is an example of one from the GATK tool, which allows multiple occurrences of the *-G* argument to appear on its command-line:

```
    <arg name="Annotation Groups" flag="-G" type="MULTI" choices="RodRequiringAnnotation, Standa
rdAnnotation, WorkInProgressAnnotation, ExperimentalAnnotation, RankSumTest"/>
```

In addition to the required attributes described so far, the <arg> element has two optional attributes:

[1] This JAR file is not included as part of the Savant SDK. It should be downloaded from http://sourceforge.net/projects/srma/files and copied to your ~/.savant/plugins directory (savant\plugins on Windows).

`required` and `default`. Most arguments are presented with a check-box next to them which lets the user control whether they should appear on the command-line, but if an argument has the `required="true"` attribute, a value must be specified.

6.4 <progress> and <error> Elements

When executing an external application as a plugin tool, Savant reads the application's standard output and standard error streams, in order to display it in the plugin's panel. In many cases, this output can also be parsed to determine the progress and status of the external application. Knowing the expected pattern of output, Savant can optionally apply a regular expression to extract that information. The example below shows the regular expressions which are used by the GATK tool plugin.

```
<progress>INFO.*TraversalEngine.*\s(\d+.\d+)%</progress>
<error>ERROR.*MESSAGE: (.*)$</error>
```