Table of Contents

LHCb Software Training: Printing and Job options

The purpose of this exercise is to make you familiar with using job options to configure algorithms, and with the methods for printing from within a Gaudi application.

Prerequisites

The instructions assume that you have already followed part 1 of the LHCb software basics tutorial. You should also have looked at the slides "Introduction to Gaudi" [.ppt] and "Printing and job options" [.ppt] attached to this topic. Please feel free to update these slides if you modify them for a future tutorial session.

These instructions have last been checked against the DaVinci v29r0 environment. Please use this version of DaVinci or a more recent version.

Setting up the environment

We will be working in the same environment as the DaVinci tutorial.

- Start Eclipse as described in the first tutorial
- Create a local DaVinci project with using version v29r0 (refer to Exercise 3)
- Check out the Tutorial/Analysis package (from the Analysis project), version v10r2 (the default) using the `GetPack Wizard`

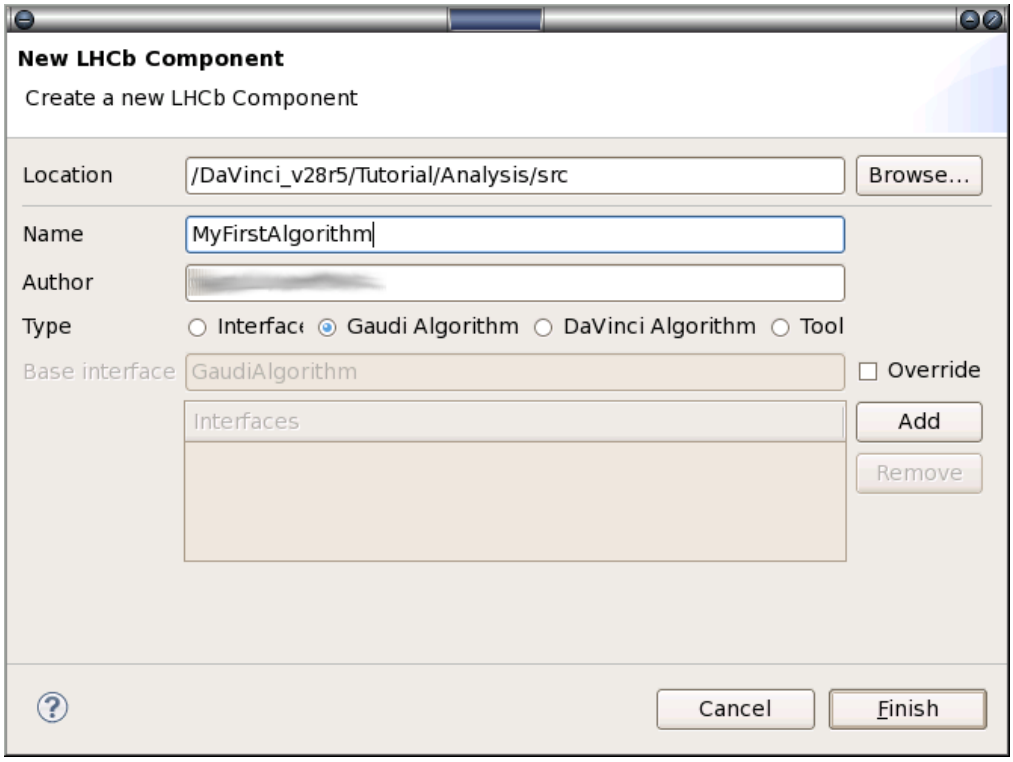
This package contains a `requirements` file already set up for the tutorial, a set of options and solutions for the DaVinci tutorial sessions, and an empty `src` directory

To exploit the full power of the Eclipse C++ editor, you need to ensure that it index is correctly populated (it is not mandatory for the success of this hands-on).

- For new projects it is very useful to enable the automatic discovery of include directories (it should be enable by default, but for the moment you have to do it by hand)
  - right-click on the new project, select *Properties* in the menu, find the entry *Discovery Options* in the menu on the left and check the checkbox *Automate discovery of path and symbols*, then select the *Discovery profile*: *GCC per project scanner* (the configuration could be more precise, but this is enough for most cases)
  - build once the project to let Eclipse collect the paths
- After the update of the paths, the index of classes and files may not be up-to-date
  - right-click on the project, and select `Index->Rebuild`

Creating a GaudiAlgorithm, adding a property and printing its value

- Click with the right button on the directory `DaVinci_v29r0/Tutorial/Analysis/src` the *Project Explorer* view and select `New->LHCb Component` in the menu
- Type `MyFirstAlgorithm` in the *Name* field and select *Gaudi Algorithm* for the *Type*, then click on *Finish*



LHCb Component wizard

- Open the file `MyFirstAlgorithm.h` which has been created in the `src` directory and add a member variable to store the property

```
double m_jPsiMassWin;
```

- Open the file `MyFirstAlgorithm.cpp`
- In the class constructor, declare a name for the property, initialize it to a default value and document it

```
declareProperty( "MassWindow", <memberVariable> = <defaultValue>, "Documentation"
);
```

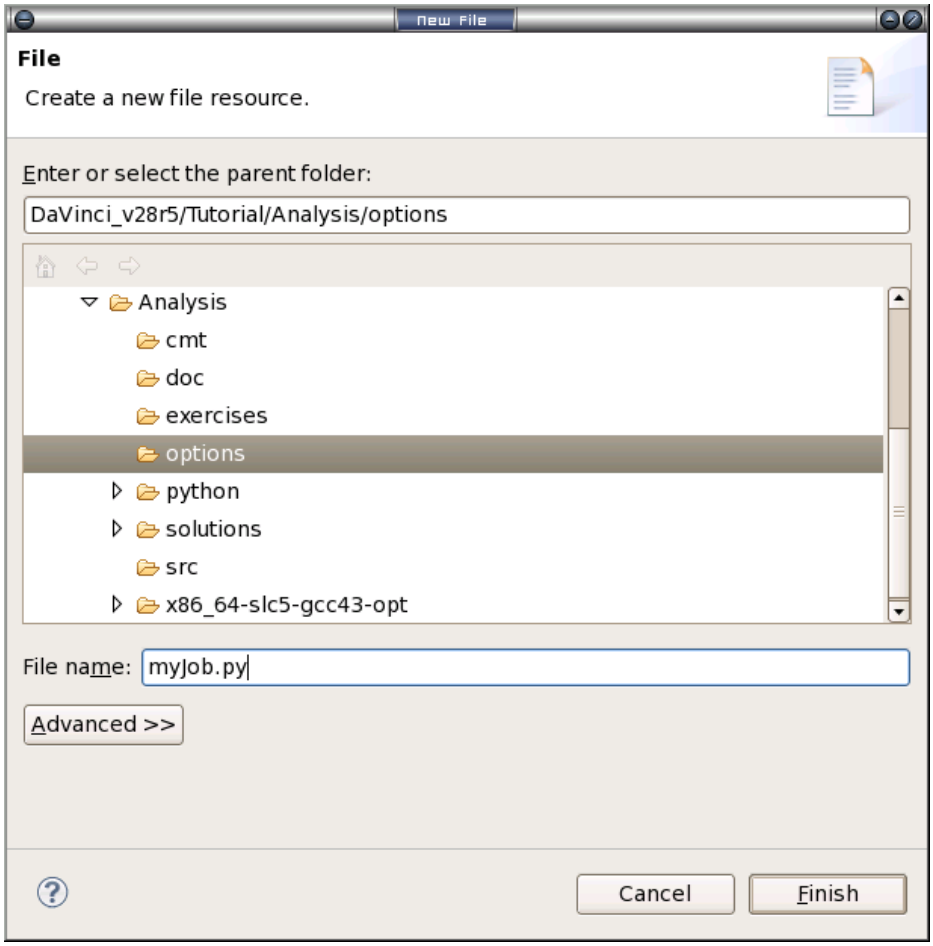
- Print the property's value twice, using two different units

```
info() << "Mass window is " << value << " Units" << endmsg;
```

- Save both files and build the project

Setting up the job options and running the job

- Create a job options file called `myJob.py` (right-click on `DaVinci_v29r0/Tutorial/Analysis/options`, then `New->File`)



New option file

Note

you may be asked to configure the python interpreter, for the moment you can select the automatic configuration and the defaults.

- Tell python about the Gaudi framework
```
from Gaudi.Configuration import *
```
- Tell python about your algorithm
```
from Configurables import MyFirstAlgorithm
```
- Add an instance of your algorithm to the application
```
myAlg = MyFirstAlgorithm()
ApplicationMgr().TopAlg += [myAlg]
```
- Save the file and run the job (see either Exercise 4 or Exercise 6 in part 1 of the tutorial). You should see some printout from your algorithm. If you do not, either your algorithm was not called (check your job options) or you put the printout in the wrong place in the code. Only printout from `initialize()` will be seen; `execute()` is not called in this example (why?)

Modify the job behavior by changing job options

The following examples illustrate how you can change an algorithm's behaviour by changing the job options, without recompiling. Try the following in turn, rerunning the job each time. No need to recompile!

- Modify the value of the algorithm's property
```
from GaudiKernel.SystemOfUnits import GeV, MeV
myAlg.MassWindow  = 1.3 * GeV
```
- One big advantage of using Python for job options is its syntax and type checking. See what happens with each of the following typing errors:

```
myAlg = MyFirstAlgorithm()
myAlg.MassWindow = 1.3 * GeV
myAlg.MassWindow = "some string"
```

- Change the global output level of the application
  `MessageSvc().OutputLevel = DEBUG`
  Try any of the values `VERBOSE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`
- Change the output level of your algorithm
  `myAlg.OutputLevel = DEBUG`
  Try any of the values `VERBOSE`, `DEBUG`, `INFO`, `WARNING`, `ERROR`
- Run two instances of your algorithm, with different values for the cut. Hint:
  `anAlg = SomeAlg("Alg1") #instantiates class SomeAlg with instance name "Alg1" and assigns it to python variable anAlg`
- A more detailed discussion of python configurables, including examples for configuring tools, can be found in the TupleToolsAndConfigurables FAQ.

Modify the job behaviour with `StatusCode`

- Look at what happens when you change the `initialize()` method of your algorithm to
  `return StatusCode::FAILURE`

The LHCb convention is that algorithms should return `StatusCode::FAILURE` from `initialize()` if there is a fatal configuration error which makes it pointless to continue with the job. Algorithms should **never** return `StatusCode::FAILURE` from inside the event loop (`execute()` method), because this will not just stop processing the current event, but will stop the job. Instead, they should trap the error and take any necessary remedial action. There are ways for algorithms to abort processing of the current event only, but these are beyond the scope of this tutorial; detailed instructions are available here

- Play with the `Warning()` and `Error()` methods.
  - What is the difference compared to using the `warning()` and `err()` MsgStream functions?.
  - What happens if you
    `return Error("An error");`
    from your algorithm?
  - And if you
    `return Error("Another error", StatusCode::SUCCESS);`

It is recommended that all errors are reported using the `Warning()` or `Error()` methods, due to the nice feature of printing statistics at the end of the job. It is also good practice that you **always** print a warning or error before returning `StatusCode::FAILURE`

Further reading

A more extensive introduction to job configuration using Python is available here and here

-- MarcoCattaneo - 27-Jan-2010

-- MarcoClemencic - 13-Sep-2011

This topic: LHCb > LHCbSoftwareTrainingPrintingEclipse
Topic revision: r9 - 2011-09-23 - MarcoClemencic