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 v33r0p1 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.

- Choose the DaVinci environment.

```
setenvDaVinci v33r0p1
```

- Check out and configure the Tutorial package

```
getpack Tutorial/Analysis v10r4
cd Tutorial/Analysis
```

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

Creating a Gaudi Algorithm, adding a property and printing its value

- Create a new header file using emacs

```
emacs src/MyFirstAlgorithm.h &
```

Answer "**A**" to emacs, meaning that you want to create a header file for a `(Gaudi)Algorithm`. **Do not** answer "D" for this exercise, DaVinciAlgorithm requires more specialised job options than those we are studying here.

- Add a member variable to store the property

```
double m_jPsiMassWin;
```

- Create the corresponding .cpp file

```
emacs src/MyFirstAlgorithm.cpp &
```

- In the class constructor, declare a property with a name by which the C++ variable can be accessed from Python, initialize it to a default value (pick something Gaudi uses units where MeV = 1); and document it

```
declareProperty( "MassWindow", m_jPsiMassWin = <defaultValue>, "The J/Psi mass window
cut" );
```

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

```
#include "GaudiKernel/SystemOfUnits.h"
...
// The following goes inside the initialize() method
info() << "Mass window: " << m_jPsiMassWin / Gaudi::Units::MeV << " MeV" << endmsg;
info() << "Mass window: " << m_jPsiMassWin / Gaudi::Units::GeV << " GeV" << endmsg;
```

- Save both files and build a library with them. Recall that the cmt/requirements file sets build configuration choices such as this. There should be, by default, two lines

```
...
library AnalysisTutorial ../src/*.cpp
...
apply_pattern component_library library=AnalysisTutorial
```

which set the build rule for the library named AnalysisTutorial and then tell cmt to build this as a component library. Running

```
cmt make
```

in the cmt directory will then build this library.

Setting up the job options and running the job

- Create a job options file

```
emacs options/myJob.py &
```

- 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. Recall the earlier exercises Exercise 4 or Exercise 6 in part 1 of the tutorial. There are two ways of running a job. If you ran "SetupProject DaVinci v33r0p1", then "gaudirun.py" is in your path (you can use the "which" command to check this) and you can directly run

```
gaudirun.py /path/to/myJob.py
```

Alternatively, you can run the job through cmt:

```
cmt run 'gaudirun.py /path/to/myJob.py'
```

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 behaviour 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.MasssWindow = 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. You cannot simply create a second variable using the constructor MyFirstAlgorithm(), because then there will be two objects with different Python variable names but referring to C++ class instantiations with the same variable name (by default, MyFirstAlgorithm() makes an instantiation with the name "MyFirstAlgorithm"). Instead,

```
anAlg = MyFirstAlgorithm("Alg1") #instantiates class MyFirstAlgorithm with instance name "Alg1" and assigns it to python variable anAlg
```

Then you can add this to the list of algorithms to be run by changing the list of algorithms [myAlg] to [myAlg, 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

This topic: LHCb > LHCbSoftwareTrainingPrinting
Topic revision: r57 - 2012-12-12 - JackWimberley