Job Options and Printing

Job Options

- All applications run the same main program (gaudirun.py)
- Job options configure the job:
  - What to run, in what order, with what data, with which cuts
  - Provided by the user in a job options configuration file
- Job options configuration file is written in python
  - Can use full power of python syntax
    - Type checking
    - Expressions, if-then-else, loops etc.
    - Early Validation of configuration
- Job options file is passed to gaudirun.py as argument(s)

```
gaudirun.py MyOpts.py [someMoreOpts.py]
```

Configurables

- Python classes, provided by the framework, used to set the job options of the C++ components

```
from Configurables import MyFirstAlgorithm
```

  - Each C++ component (Algorithm, Tool, Service) has a corresponding python Configurables
  - To set the properties of a component, must first instantiate the corresponding python Configurable

```
myAlg = MyFirstAlgorithm()
```

Instance of the Python class

Python variable holding the instance

  - Then use it to set the properties of the C++ component

```
myAlg.OutputLevel = DEBUG
```

Running the C++ algorithms

- Merely instantiating the python configurable does not instantiate the corresponding C++ component
  - Some special configurables have properties that define sequences of algorithms to be executed
  - Python instances must be added to these sequences

```
ApplicationMgr().TopAlg += [ myAlg ]
```

  - Execute an instance of the C++ MyFirstAlgorithm, as configured on the previous slide, in the TopAlg sequence of the ApplicationMgr

```
DaVinci().UserAlgorithms += [ myAlg ]
```

  - Execute an instance of the C++ MyFirstAlgorithm, as configured on the previous slide, in the UserAlgorithms sequence of DaVinci

Named algorithms

- By default, instance of an algorithm has the same name as the C++ class (and python configurable class)
  - e.g. "MyFirstAlgorithm"
- To run several instances of the same algorithm, give it an an explicit name

```
myFred = MyFirstAlgorithm( name = "Fred" )
myGeorge = MyFirstAlgorithm( name = "George" )
myFred.MassWindow = 3. * GeV
myGeorge.MassWindow = 2500. * MeV
ApplicationMgr().TopAlg += [ myFred, myGeorge ]
```

  - Execute two instances of MyFirstAlgorithm, with different values for the MassWindow property; execute "Fred" before "George"
  - N.B. MassWindow must have been declared as a property in the C++ code

Named Tools

- Tools always have a name, defined in the C++ code. They are created by a named instance of a C++ component (Algorithm, Tool, Service)

```
MyFirstAlgorithm::initialise() {
ICutlery* theTool = tool<ICutlery>("Knife", "MeatKnife");
```

  - In his case an algorithm of type MyFirstAlgorithm creates a tool of type Knife, with interface ICutlery, called "MeatKnife"

- Use the same names in python configuration:

```
theCook = MyFirstAlgorithm( name = "Cook" )
# Create a configurable for a tool named "MeatKnife", of
# type Knife, and associate it to the theCook configurable
theCook.addTool( Knife, name ="MeatKnife" )
# Now set a property of the tool
theCook.MeatKnife.OutputLevel = DEBUG
```

Declaring properties in the C++ code

- Add a member variable to hold the property

```
class MyFirstAlgorithm : public GaudiAlgorithm  {
  private:
    double m_jPsiMassWin; ///< J/Psi mass window cut
      ...
};
```

LHCb coding convention for member data

doxygen documentation string

- Declare as a property in the constructor and initialize it with a default value

```
MyFirstAlgorithm::MyFirstAlgorithm( <args> )
{
  declareProperty( "MassWindow", ///< Property name used in job options file
    m_jPsiMassWin = 0.5*Gaudi::Units::GeV, ///< Variable initialized to default
    "The J/Psi mass window cut" );  ///< Documentation string for Python
}
```

Aside: all member data must always be initialised in the constructor

Printing

- Why not use std::cout, std::cerr, ... ?
  - Yes, it prints, but
    - Do you always want to print to the log file?
    - How can you connect std::cout to the message window of an event display?
    - How can you add a timestamp to the messages?
    - You may want to switch on/off printing at several levels just for one given algorithm, service etc.

Printing - MsgStream

- Using the MsgStream class
  - Usable like std::cout
  - Allows for different levels of printing
    - MSG::VERBOSE (=1)
    - MSG::DEBUG (=2)
    - MSG::INFO (=3)
    - MSG::WARNING (=4)
    - MSG::ERROR (=5)
    - MSG::FATAL (=6)
    - MSG::ALWAYS (=7)
  - Record oriented
  - Allows to define severity level per object instance

MsgStream - Usage

- Send to predefined message stream

```
info() << "PDG particle ID of " << m_partName
       << " is " << m_partID << endmsg;
err() << "Cannot retrieve properties for particle "
      << m_partName << endmsg;
```

- Print error and return bad status

```
return Error("Cannot retrieve particle properties");
```

- String formatting

```
debug() << format("E: %8.3f GeV", energy ) << endmsg;
```

- Set print level in options

```
MessageSvc().OutputLevel    = ERROR
MySvc().OutputLevel         = WARNING
MyAlgorithm().OutputLevel   = INFO
```

Print everything of INFO level or higher

Units

- We use Geant4/CLHEP system of units
  - mm, MeV, ns are defined to have value 1.
  - All other units defined relative to this
  - In header file "GaudiKernel/SystemOfUnits.h"
  - In namespace Gaudi::Units
- Multiply by units to set value:

```
double m_jPsiMassWin = 0.5 * Gaudi::Units::GeV;
```

- Divide by units to print value:

```
info() << "Mass window: " << m_jPsiMassWin / Gaudi::Units::MeV
       << " MeV" << endmsg;
```

- Units can be used also in job options:

```
import GaudiKernel.SystemOfUnits as Units
SomeAlgorithm().MassWindow = 0.3 * Units.GeV
```

StatusCode

- Object returned by many methods
  - Including GaudiAlgorithm::initialize(), GaudiAlgorithm::execute() , etc.
  - Currently, takes two values:
    - StatusCode::SUCCESS, StatusCode::FAILURE
- Should always be tested
  - If function returns StatusCode, there must be a reason
  - Report failures:

```
StatusCode sc = someFunctionCall();
if ( sc.isFailure() )
  { Warning("there is a problem",sc,0).ignore();}
```

- If IAlgorithm methods return StatusCode::FAILURE, processing stops
  - Always return StatusCode::SUCCESS from these methods

Exercise

- Now read the web page attached to this lesson in the agenda and work through the exercise