Table of Contents

Class Description

The GaudiAlgorithm class is an extension to the basic Gaudi Algorithm class. It provides additional shortcut methods for things such as :-

- Simple and safe location of services and tools
- Easy messaging and exception handling
- Easy handling of statistical counters

Locating *Tools* and *Services*

If one inherits the code from the base classes `GaudiAlgorithm` or `GaudiTool`, one can locate tools in a very simple way using the templated method `tool`:

```
1000// locate the private tool using type and name:
1010IMyTool* mytool1  = tool<IMyTool>("MyToolType","MyToolName", this ) ;
1020
1030// locate the private tool using "type/name" semantic:
1040IMyTool* mytool2  = tool<IMyTool>("MyToolType/MyToolName"  , this ) ;
1050
1060// locate the public tool using type and name:
1070IMyTool* mytool3  = tool<IMyTool>("MyToolType","MyToolName" ) ;
1080
1090// locate the public tool using "type/name" semantic:
1100IMyTool* mytool4  = tool<IMyTool>("MyToolType/MyToolName"  ) ;
1110
1120// locate the public tool using ":PUBLIC" keyword
1130IMyTool* mytool5 = tool<IMyTool>("MyToolType","MyToolName:PUBLIC", this ) ;
1140
1150// locate the public tool using "type/name" semantic with ":PUBLIC" keyword:
1160IMyTool* mytool6  = tool<IMyTool>("MyToolType/MyToolName:PUBLIC"  , this ) ;
```

There is no need to release the acquired tools, all acquired tools will be properly released in the `finalize` method of the base class.

Similar methods exist for the easy location of services using similar semantics to tools:

```
1000// locate the service by name, create if needed..
1010IMyService* s = svc<IMyService>( "MyServiceType" , true ) ;
```

These methods throw exception if the required tool or service is not found, therefore there is no need to check the returned pointer - the system ensures its validity.

Using the predefined message streams

The base classes `GaudiAlgorithm` and `GaudiTool` predefine a set of methods for easy control of printout:

```
1000// print as MSG::VERBOSE
1010verbose() << " It is VERBOSE printout " << endmsg ;
1020
1030// print as MSG::DEBUG
1040debug()   << "It is DEBUG   printout" << endmsg ;
1050
1060// print as MSG::INFO
1070info()    << "It is INFO    printout" << endmsg ;
1080
1090// ditto
```

```
msg()    << "Is it INFO    printout" << endmsg ;

// print as MSG::WARNING
warning() << "It is WARNING printout" << endmsg ;

// print as MSG::ERROR
error()   << "It is ERROR   printout" << endmsg ;

// ditto
err()     << "It is ERROR  printout" << endmsg ;

// print as MSG::FATAL
fatal()   << "It is FATAL printout" << endmsg ;

// print as MSG::ALWAYS
always()  << "It is ALWAYS  printout" << endmsg ;

// print with the specified level:
MSG::Level level = ... ;
msgStream ( level ) << "It is custom  printout" << endmsg ;
```

The usage of these predefined methods decreases significantly the CPU penalty for creation/destruction of local `MsgStream` objects. However some penalty for deactivated streams is still visible, e.g. see the discussion around Savannah patch #1196 by Patrick Koppenburg. In short, one should avoid the unchecked usage of the streams, especially for inner loops. It is recommended to check the status of the stream with the `msgLevel` before actual usage of the stream:

```
for ( long i = 0 ; i < 1000000 ; ++ i )
{
    // bad style, not efficient:
    verbose() << " some printout " << .... << endmsg ;  ///< very bad!, not efficient!

    // good style, efficient:
   if ( msgLevel   ( MSG::VERBOSE ) )  {  verbose () << " another printout " << ... << endmsg
}
```

Using the generic statistical counters

The `GaudiAlgorithm` and `GaudiTool` base classes are equipped with the method `counter`, which returns a reference to the local instance of *the named counter* (C++ type `StatEntity&`):

```
const std::size_t nTracks = ... ;

// increment the counter:
counter("#Tracks")+= nTracks ;
```

The current content of the generic counter could be inspected:

```
const StatEntity& cnt = ... ;
always()
    << " #of entries " << cnt.nEntries() << endmsg
    << " Total sum   " << cnt.flag()          << endmsg
    << " Mean value  " << cnt.flagMean () << endmsg
    << " RMS "  << cnt.flagRMS() << endmsg
    << " Minimal value " << cnt.flagMin() << endmsg
    << " Maximal value " << cnt.flagMax () << endmsg ;
```

If the content of the generic counter allows the interpretation as a binominal efficiency counter one can ask for efficiency and its uncertainty:

```
1000always()
1010    << " Efficiency " << cnt.eff()
1020    << " +-"          << cnt.effErr()     << endmsg;
```

All counters will be printed at the end of the job in a table if the property `"StatPrint"` is activated (default):

```
Counter            SUCCESS Number of counters : 8
Counter            SUCCESS      Counter      |      #      |     sum      | mean/eff^* | rms/err^*   |
Counter            SUCCESS  "G"              |     26490 |    -326.309 |  -0.012318 |    0.99484 |
Counter            SUCCESS  "Gneg"           |      2712 |        2712 |     1.0000 |    0.00000 |
Counter            SUCCESS  "Gpos"           |      2688 |        2688 |     1.0000 |    0.00000 |
Counter            SUCCESS  "NG"             |     26490 |       26490 |     1.0000 |    0.00000 |
Counter            SUCCESS *"eff"            |      5400 |        2688 |( 49.77778 +- 0.6804071)%|
Counter            SUCCESS  "executed"       |      5400 |        5400 |     1.0000 |    0.00000 |
Counter            SUCCESS  "g2"             |      5400 |    5341.386 |    0.98915 |     1.3535 | 1
Counter            SUCCESS  "gauss"          |      5400 |   -8.611382 | -0.0015947 |    0.99456 |
```

Why are some counters printed with ------ symbols as min/max values and their names are prepended with asterisks ?

the counter is printed as *the binomial efficiency counter* using the special format if:

1. *the name* of the counter contains (case-insentitive) substrings `"eff"`, `"acc"`, `"pass"`, `"filt"` or `"fltr"`.
2. *the content* of the counter could be interpreted as *the binomial efficiency counter* (see here)
3. the property `UseEfficiencyRowFormat` is activated, see (see here).

How can I change the format of the final table of the counters?

The format of the final table is under the control of following properties:

Property	Type	Description
`StatTableHeader`	string	The string to be used as the header of the table
`RegularRowFormat`	string	The format to be used for the regular counters
`EfficiencyRowFormat`	string	The special format to be used for the binomial efficiency counters
`UseEfficiencyRowFormat`	`true`	Switch on/off the special treatment for the binomial efficiency counters

The formatting is performed using Boost Format Library.

Doxygen Documentation

http://proj-gaudi.web.cern.ch/proj-gaudi/releases/latest/doxygen/class_gaudi_algorithm.html

---

ChrisRJones - 29 Jul 2005

MarcoCattaneo - 20 Mar 2009

This topic: LHCb > GaudiAlgorithm
Topic revision: r4 - 2012-10-10 - MarcoCattaneo

Copyright &© 2008-2020 by the contributing authors. All material on this collaboration platform is the property of the contributing authors.
Ideas, requests, problems regarding TWiki? Send feedback