Plugins with Qt

Qt 3.0

Contents

Qt Plugins Documentation

Qt provides a simple plugin interface which makes it easy to create custom database drivers, image formats, text codecs, styles and widgets as stand-alone components.

Writing a plugin is achieved by subclassing the appropriate plugin base clase, implementing a few functions, and adding a macro.

There are five plugin base classes, and their plugins are stored in the standard plugin directory.

Base Class	Default Path
QImageFormatPlugin	`$QTDIR/plugins/imageformats`
QSqlDriverPlugin	`$QTDIR/plugins/sqldrivers`
QStylePlugin	`$QTDIR/plugins/styles`
QTextCodecPlugin	`$QTDIR/plugins/codecs`
QWidgetPlugin	`$QTDIR/plugins/widgets`

Suppose that you have a new style class called 'MyStyle' that you want to make available as a plugin. The required code is straightforward:

```
class MyStylePlugin : public QStylePlugin
{
public:
    MyStylePlugin() {}
    ~MyStylePlugin() {}

    QStringList keys() const {
        return QStringList() << "MyStyle";
    }

    QStyle* create( const QString& key ) {
        if ( key == "MyStyle" )
            return new MyStyle;
        return 0;
    }
};

Q_EXPORT_PLUGIN( MyStylePlugin )
```

The constructor and destructor do not need to do anything, so are left empty. There are only two virtual functions that must be implemented. The first is keys() which returns a string list of the classes implemented in the plugin. (We've just implemented one class in the example above.) The second is a function that returns an object of the required class (or 0 if the plugin is asked to create an object of a class that it doesn't implement). For QStylePlugin, this second function is called create().

It is possible to implement any number of plugin subclasses in a single plugin, providing they are all derived from the same base class, e.g. QStylePlugin.

For database drivers, image formats, custom widgets and text codecs, no explicit object creation is required. Qt will find and create them as required. Styles are an exception, since you might want to set a style explicitly in code. To apply a style, use code like this:

```
    QApplication::setStyle( QStyleFactory::create( "MyStyle" ) );
```

Some plugin classes require additional functions to be implemented, see the Qt Designer manual's, 'Creating Custom Widgets' section in the 'Creating Custom Widgets' chapter, for a complete example of a QWidgetPlugin, which implements extra functions to integrate the plugin into *Qt Designer*.

See the class documentation for details of the virtual functions that must be reimplemented for each type of plugin.

Qt applications automatically know which plugins are available, because plugins are stored in the standard plugin subdirectories. Because of this applications don't require any code to find and load plugins, since Qt handles them automatically.

The default directory for plugins is `$QTDIR/plugins`, with each type of plugin in a subdirectory for that type, e.g. `styles`. If you want your applications to use plugins and you don't want to use the standard plugins path, have your installation process determine the path you want to use for the plugins, and save the path, e.g. using QSettings, for the application to read when it runs. The application can then call QApplication::addLibraryPath() with this path and your plugins will be available to the application. Note that the final part of the path, i.e. `styles`, `widgets`, etc. cannot be changed.

QImageFormatPlugin Class Reference

The QImageFormatPlugin class provides an abstract base for custom image format plugins.

```
#include <qimageformatplugin.h>
```

Public Members

- **QImageFormatPlugin** ()
- **~QImageFormatPlugin** ()
- virtual QStringList **keys** () const
- virtual bool **installIOHandler** (const QString & format)

Detailed Description

The QImageFormatPlugin class provides an abstract base for custom image format plugins.

The image format plugin is a simple plugin interface that makes it easy to create custom image formats that can be used transparently by applications.

Writing an image format plugin is achieved by subclassing this base class, reimplementing the pure virtual functions keys() and installIOHandler(), and exporting the class with the Q_EXPORT_PLUGIN macro. See the Plugins documentation for details.

See also Plugins.

Member Function Documentation

QImageFormatPlugin::QImageFormatPlugin ()

Constructs an image format plugin. This is invoked automatically by the Q_EXPORT_PLUGIN macro.

QImageFormatPlugin::~QImageFormatPlugin ()

Destroys the image format plugin.

You never have to call this explicitly. Qt destroys a plugin automatically when it is no longer used.

bool QImageFormatPlugin::installIOHandler (const QString & format) [virtual]

Installs a QImageIO image I/O handler for the image format *format*.

See also keys() [p. 6].

QStringList QImageFormatPlugin::keys () const [virtual]

Returns the list of image formats this plugin supports.

See also installIOHandler() [p. 6].

QSqlDriverPlugin Class Reference

The QSqlDriverPlugin class provides an abstract base for custom QSqlDriver plugins.

`#include <qsqldriverplugin.h>`

Public Members

- **QSqlDriverPlugin** ()
- **~QSqlDriverPlugin** ()
- virtual QStringList **keys** () const
- virtual QSqlDriver * **create** (const QString & key)

Detailed Description

The QSqlDriverPlugin class provides an abstract base for custom QSqlDriver plugins.

The SQL driver plugin is a simple plugin interface that makes it easy to create your own SQL driver plugins that can be loaded dynamically by Qt.

Writing a SQL plugin is achieved by subclassing this base class, reimplementing the pure virtual functions keys() and create(), and exporting the class with the Q_EXPORT_PLUGIN macro. See the SQL plugins that comes with Qt for example implementations (in `$QTDIR/plugins/src/sqldrivers`). Read the plugins documentation for more information on plugins.

See also Plugins.

Member Function Documentation

QSqlDriverPlugin::QSqlDriverPlugin ()

Constructs a SQL driver plugin. This is invoked automatically by the Q_EXPORT_PLUGIN macro.

QSqlDriverPlugin::~QSqlDriverPlugin ()

Destroys the SQL driver plugin.

You never have to call this explicitly. Qt destroys a plugin automatically when it is no longer used.

QSqlDriver * QSqlDriverPlugin::create (const QString & key) [virtual]

Creates and returns a QSqlDriver object for the driver key *key*. The driver key is usually the class name of the required driver.

See also keys() [p. 8].

QStringList QSqlDriverPlugin::keys () const [virtual]

Returns the list of driver keys this plugin supports.

These keys are usually the class names of the custom drivers that are implemented in the plugin.

See also create() [p. 8].

QStylePlugin Class Reference

The QStylePlugin class provides an abstract base for custom QStyle plugins.

```
#include <qstyleplugin.h>
```

Public Members

- **QStylePlugin** ()
- **~QStylePlugin** ()
- virtual QStringList **keys** () const
- virtual QStyle * **create** (const QString & key)

Detailed Description

The QStylePlugin class provides an abstract base for custom QStyle plugins.

The style plugin is a simple plugin interface that makes it easy to create custom styles that can be loaded dynamically into applications with a QStyleFactory.

Writing a style plugin is achieved by subclassing this base class, reimplementing the pure virtual functions keys() and create(), and exporting the class with the Q_EXPORT_PLUGIN macro. See the plugins documentation for an example.

See also Plugins.

Member Function Documentation

QStylePlugin::QStylePlugin ()

Constructs a style plugin. This is invoked automatically by the Q_EXPORT_PLUGIN macro.

QStylePlugin::~QStylePlugin ()

Destroys the style plugin.

You never have to call this explicitly. Qt destroys a plugin automatically when it is no longer used.

QStyle * QStylePlugin::create (const QString & key) [virtual]

Creates and returns a QStyle object for the style key *key*. The style key is usually the class name of the required style.

See also keys() [p. 10].

QStringList QStylePlugin::keys () const [virtual]

Returns the list of style keys this plugin supports.

These keys are usually the class names of the custom styles that are implemented in the plugin.

See also create() [p. 10].

QStyleFactory Class Reference

The QStyleFactory class creates QStyle objects.

`#include <qstylefactory.h>`

Static Public Members

- QStringList **keys** ()
- QStyle * **create** (const QString & key)

Detailed Description

The QStyleFactory class creates QStyle objects.

The style factory creates a QStyle object for a given key with QStyleFactory::create(key).

The styles are either built-in or dynamically loaded from a style plugin (see QStylePlugin).

QStyleFactory::keys() returns a list of valid keys. Qt currently ships with "windows", "motif", "cde", "platinum", "sgi" and "motifplus".

Member Function Documentation

QStyle * QStyleFactory::create (const QString & key) [static]

Creates a QStyle object that matches *key*. This is either a built-in style, or a style from a style plugin.

See also keys() [p. 11].

Example: themes/wood.cpp.

QStringList QStyleFactory::keys () [static]

Returns the list of keys this factory can create styles for.

See also create() [p. 11].

QStyleOption Class Reference

The QStyleOption class specifies optional parameters for QStyle functions.

```
#include <qstyle.h>
```

Public Members

- enum **StyleOptionDefault** { Default }
- **QStyleOption** (StyleOptionDefault = Default)
- **QStyleOption** (int in1, int in2)
- **QStyleOption** (int in1, int in2, int in3, int in4)
- **QStyleOption** (QMenuItem * m)
- **QStyleOption** (QMenuItem * m, int in1)
- **QStyleOption** (QMenuItem * m, int in1, int in2)
- **QStyleOption** (const QColor & c)
- **QStyleOption** (QTab * t)
- **QStyleOption** (QListViewItem * i)
- **QStyleOption** (Qt::ArrowType a)
- bool **isDefault** () const
- int **lineWidth** () const
- int **midLineWidth** () const
- int **frameShape** () const
- int **frameShadow** () const
- QMenuItem * **menuItem** () const
- int **maxIconWidth** () const
- int **tabWidth** () const
- const QColor & **color** () const
- QTab * **tab** () const
- QListViewItem * **listViewItem** () const
- Qt::ArrowType **arrowType** () const

Detailed Description

The QStyleOption class specifies optional parameters for QStyle functions.

Some QStyle functions take an optional argument specifying extra information that is required for a paritical primitive or control. So that the QStyle class can be extended, QStyleOption is use to give a variable-argument for these options.

The QStyleOption class has constructors for each type of optional argument, and this set of constructors may be extended in future Qt releases. There are also corresponding access functions that return the optional arguments - these too may be extended.

For each constructor, you should refer to the documentation of the QStyle functions to see the meaning of the arguments.

When calling QStyle functions from your own widgets, you must only pass either the default QStyleOption or the argument that QStyle is documented to accept. For example, if the function expects QStyleOption(QMenuItem *, int), passing QStyleOption(QMenuItem *) leaves the optional integer argument uninitialized.

When subclassing QStyle, you must similarly only expect the default or documented arguments. The other arguments will have uninitialized values.

If you make your own QStyle subclasses and your own widgets, you can make a subclass of QStyleOption to pass additional arguments to your QStyle subclass. You will need to cast the "const QStyleOption&" argument to your subclass, so be sure your style has been called from your widget.

See also Widget Appearance and Style.

Member Type Documentation

QStyleOption::StyleOptionDefault

This enum value can be passed as the optional argument to any QStyle function.

- `QStyleOption::Default`

Member Function Documentation

QStyleOption::QStyleOption (StyleOptionDefault = Default)

The default option. This can always be passed as the optional argument to QStyle functions.

QStyleOption::QStyleOption (int in1, int in2)

Pass two integers, *in1* and *in2*. For example, linewidth and midlinewidth.

QStyleOption::QStyleOption (int in1, int in2, int in3, int in4)

Pass four integers, *in1*, *in2*, *in3* and *in4*.

QStyleOption::QStyleOption (QMenuItem * m)

Pass a menu item, *m*.

QStyleOption::QStyleOption (QMenuItem * m, int in1)

Pass a menu item and an integer, *m* and *in1*.

QStyleOption::QStyleOption (QMenuItem * m, int in1, int in2)

Pass a menu item and two integers, *m*, *in1* and *in2*.

QStyleOption::QStyleOption (const QColor & c)

Pass a color, *c*.

QStyleOption::QStyleOption (QTab * t)

Pass a QTab, *t*.

QStyleOption::QStyleOption (QListViewItem * i)

Pass a QListViewItem, *i*.

QStyleOption::QStyleOption (Qt::ArrowType a)

Pass an Qt::ArrowType, *a*.

Qt::ArrowType QStyleOption::arrowType () const

Returns an arrow type if the appropriate constructor was called, else undefined.

const QColor & QStyleOption::color () const

Returns a color if the appropriate constructor was called, else undefined.

int QStyleOption::frameShadow () const

Returns a QFrame::Shadow value if the appropriate constructor was called, else undefined.

int QStyleOption::frameShape () const

Returns a QFrame::Shape value if the appropriate constructor was called, else undefined.

bool QStyleOption::isDefault () const

Returns whether the option was constructed with the default constructor.

int QStyleOption::lineWidth () const

Returns the line width if the appropriate constructor was called, else undefined.

QListViewItem * QStyleOption::listViewItem () const

Returns a QListView item if the appropriate constructor was called, else undefined.

int QStyleOption::maxIconWidth () const

Returns the maximum width of the menu item check area if the appropriate constructor was called, else undefined.

QMenuItem * QStyleOption::menuItem () const

Returns a menu item if the appropriate constructor was called, else undefined.

int QStyleOption::midLineWidth () const

Returns the mid-line width if the appropriate constructor was called, else undefined.

QTab * QStyleOption::tab () const

Returns a QTabBar tab if the appropriate constructor was called, else undefined.

int QStyleOption::tabWidth () const

Returns the tab indent width if the appropriate constructor was called, else undefined.

QTextCodecPlugin Class Reference

The QTextCodecPlugin class provides an abstract base for custom QTextCodec plugins.

`#include <qtextcodecplugin.h>`

Public Members

- **QTextCodecPlugin** ()
- **~QTextCodecPlugin** ()
- virtual QStringList **names** () const
- virtual QTextCodec * **createForName** (const QString & name)
- virtual QValueList<int> **mibEnums** () const
- virtual QTextCodec * **createForMib** (int mib)

Detailed Description

The QTextCodecPlugin class provides an abstract base for custom QTextCodec plugins.

The text codec plugin is a simple plugin interface that makes it easy to create custom text codecs that can be loaded dynamically into applications.

Writing a text codec plugin is achieved by subclassing this base class, reimplementing the pure virtual functions names(), createForName(), mibEnums() and createForMib(), and exporting the class with the Q_EXPORT_PLUGIN macro. See the Qt Plugins documentation for details.

See the IANA character-sets encoding file for more information on mime names and mib enums.

See also Plugins.

Member Function Documentation

QTextCodecPlugin::QTextCodecPlugin ()

Constructs a text codec plugin. This is invoked automatically by the Q_EXPORT_PLUGIN macro.

QTextCodecPlugin::~QTextCodecPlugin ()

Destroys the text codec plugin.

You never have to call this explicitly. Qt destroys a plugin automatically when it is no longer used.

QTextCodec * QTextCodecPlugin::createForMib (int mib) [virtual]

Creates a QTextCodec object for the mib enum *mib*.

(See the IANA character-sets encoding file for more information)

See also mibEnums() [p. 17].

QTextCodec * QTextCodecPlugin::createForName (const QString & name) [virtual]

Creates a QTextCodec object for the codec called *name*.

See also names() [p. 17].

QValueList<int> QTextCodecPlugin::mibEnums () const [virtual]

Returns the list of mib enums this plugin supports.

See also createForMib() [p. 17].

QStringList QTextCodecPlugin::names () const [virtual]

Returns the list of mime names this plugin supports.

See also createForName() [p. 17].

QWidgetPlugin Class Reference

The QWidgetPlugin class provides an abstract base for custom QWidget plugins.

```
#include <qwidgetplugin.h>
```

Public Members

- **QWidgetPlugin** ()
- **~QWidgetPlugin** ()
- virtual QStringList **keys** () const
- virtual QWidget * **create** (const QString & key, QWidget * parent = 0, const char * name = 0)
- virtual QString **group** (const QString & key) const
- virtual QIconSet **iconSet** (const QString & key) const
- virtual QString **includeFile** (const QString & key) const
- virtual QString **toolTip** (const QString & key) const
- virtual QString **whatsThis** (const QString & key) const
- virtual bool **isContainer** (const QString & key) const

Detailed Description

The QWidgetPlugin class provides an abstract base for custom QWidget plugins.

The widget plugin is a simple plugin interface that makes it easy to create custom widgets that can be included in forms using Qt Designer and used by applications.

Writing a widget plugin is achieved by subclassing this base class, reimplementing the pure virtual functions keys(), create(), group(), iconSet(), includeFile(), toolTip(), whatsThis() and isContainer(), and exporting the class with the Q_EXPORT_PLUGIN macro.

See the Qt Designer manual's, 'Creating Custom Widgets' section in the 'Creating Custom Widgets' chapter, for a complete example of a QWidgetPlugin.

See also the Plugins documentation [p. 3].

See also Plugins.

Member Function Documentation

QWidgetPlugin::QWidgetPlugin ()

Constructs a widget plugin. This is invoked automatically by the Q_EXPORT_PLUGIN macro.

QWidgetPlugin::~QWidgetPlugin ()

Destroys the widget plugin.

You never have to call this explicitly. Qt destroys a plugin automatically when it is no longer used.

QWidget * QWidgetPlugin::create (const QString & key, QWidget * parent = 0, const char * name = 0) [virtual]

Creates and returns a QWidget object for the widget key *key*. The widget key is usually the class name of the required widget. The *name* and *parent* arguments are passed to the custom widget's constructor.

See also keys() [p. 20].

QString QWidgetPlugin::group (const QString & key) const [virtual]

Returns the group (toolbar name) that the custom widget called *key* should be part of when *Qt Designer* loads it.

The default implementation returns a null string.

QIconSet QWidgetPlugin::iconSet (const QString & key) const [virtual]

Returns the iconset that *Qt Designer* should use to represent the custom widget called *key* in the toolbar.

The default implementation returns an null iconset.

QString QWidgetPlugin::includeFile (const QString & key) const [virtual]

Returns the name of the include file that *Qt Designer* and `uic` should use to include the custom widget called *key* in generated code.

The default implementation returns a null string.

bool QWidgetPlugin::isContainer (const QString & key) const [virtual]

Returns TRUE if the custom widget called *key* can contain other widgets, e.g. like QFrame; otherwise returns FALSE.

The default implementation returns FALSE.

QStringList QWidgetPlugin::keys () const [virtual]

Returns the list of widget keys this plugin supports.

These keys are usually the class names of the custom widgets that are implemented in the plugin.

See also create() [p. 19].

QString QWidgetPlugin::toolTip (const QString & key) const [virtual]

Returns the text of the tooltip that *Qt Designer* should use for the custom widget called *key*'s toolbar button.

The default implementation returns a null string.

QString QWidgetPlugin::whatsThis (const QString & key) const [virtual]

Returns the text of the whatsThis text that *Qt Designer* should use when the user requests whatsThis help for the custom widget called *key*.

The default implementation returns a null string.

QLibrary Class Reference

The QLibrary class provides a wrapper for handling shared libraries.

`#include <qlibrary.h>`

Public Members

- **QLibrary** (const QString & filename)
- virtual **~QLibrary** ()
- void * **resolve** (const char * symb)
- bool **load** ()
- virtual bool **unload** ()
- bool **isLoaded** () const
- bool **autoUnload** () const
- void **setAutoUnload** (bool enabled)
- QString **library** () const

Static Public Members

- void * **resolve** (const QString & filename, const char * symb)

Detailed Description

The QLibrary class provides a wrapper for handling shared libraries.

An instance of a QLibrary object can handle a single shared library and provide access to the functionality in the library in a platform independent way. If the library is a component server, QLibrary provides access to the exported component and can directly query this component for interfaces.

QLibrary ensures that the shared library is loaded and stays in memory whilst it is in use. QLibrary can also unload the library on destruction and release unused resources.

A typical use of QLibrary is to resolve an exported symbol in a shared object, and to e.g. call the function that this symbol represents. This is called "explicit linking" in contrast to "implicit linking", which is done by the link step in the build process when linking an executable against a library.

The following code snippet loads a library, resolves the symbol "mysymbol", and calls the function if everything succeeded. If something went wrong, e.g. the library file does not exist or the symbol is not defined, the function pointer will become a null pointer. Upon destruction of the QLibrary object the library will be unloaded, making all references to memory allocated in the library invalid.

```
typedef void (*MyPrototype)();
MyPrototype myFunction;

QLibrary myLib( "mylib" );
myFunction = (MyProtoype) myLib.resolve( "mysymbol" );
if ( myFunction ) {
    myFunction();
}
```

Member Function Documentation

QLibrary::QLibrary (const QString & filename)

Creates a QLibrary object for the shared library *filename*. The library will be unloaded in the destructor.

Note that *filename* does not need to include the (platform specific) file extension, so calling

```
QLibrary lib( "mylib" );
```

is equivalent to calling

```
QLibrary lib( "mylib.dll" );
```

on Windows. Specifying the extension is not recommended, since doing so introduces a platform dependency.

If *filename* does not include a path, the library loader will look for the file in the platform specific search paths.

See also load() [p. 23], unload() [p. 24] and setAutoUnload() [p. 23].

QLibrary::~QLibrary () [virtual]

Deletes the QLibrary object.

The library will be unloaded if autoUnload() is TRUE (the default), otherwise it stays in memory until the application is exited.

See also unload() [p. 24] and setAutoUnload() [p. 23].

bool QLibrary::autoUnload () const

Returns TRUE if the library will be automatically unloaded when this wrapper object is destructed; otherwise returns FALSE. The default is TRUE.

See also setAutoUnload() [p. 23].

bool QLibrary::isLoaded () const

Returns TRUE if the library is loaded; otherwise returns FALSE.

See also unload() [p. 24].

QString QLibrary::library () const

Returns the filename of the shared library this QLibrary object handles, including the platform specific file extension.

For example:

```
QLibrary lib( "mylib" );
QString str = lib.library();
```

will set *str* to "mylib.dll" on Windows, and "libmylib.so" on Linux.

bool QLibrary::load ()

Loads the library. Since resolve() always calls this function before resolving any symbols it is not necessary to call this function explicitly. In some situations you might want the library loaded in advance, in which case you would call this function.

void * QLibrary::resolve (const char * symb)

Returns the address of the exported symbol *symb*. The library is loaded if necessary. The function returns a null pointer if the symbol could not be resolved or the library could not be loaded.

```
typedef int (*avgProc)( int, int );

avgProc avg = (avgProc) library->resolve( "avg" );
if ( avg )
    return avg( 5, 8 );
else
    return -1;
```

void * QLibrary::resolve (const QString & filename, const char * symb) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Loads the library *filename* and returns the address of the exported symbol *symb*. Note that like the constructor, *filename* does not need to include the (platform specific) file extension. The library remains loaded until the process exits.

The function returns a null pointer if the symbol could not be resolved or the library could not be loaded.

void QLibrary::setAutoUnload (bool enabled)

If *enabled* is TRUE (the default), the wrapper object is set to automatically unload the library upon destruction. If *enabled* is FALSE, the wrapper object is not unloaded unless you explicitly call unload().

See also autoUnload() [p. 22].

bool QLibrary::unload () [virtual]

Unloads the library and returns TRUE if the library could be unloaded; otherwise returns FALSE.

This function is called by the destructor if autoUnload() is enabled.

See also resolve() [p. 23].

Index