CHAPTER 8

Files, Streams, and XML

Handling files is a complex problem when it comes to cross-platform applications because even the most basic features can vary across platforms. For instance, Unix systems use the slash (`/`) as a separator in paths, whereas the Windows platform uses a backslash (`\`). And this is just the beginning; you'll also encounter an unnerving array of fundamental differences such as different line endings and encodings, each of which can cause all sorts of strange problems to crop up when you attempt to coax your application into running on multiple platforms.

To overcome this problem, Qt offers a range of classes to handle paths, files, and streams. Qt also handles XML files—a format structuring the contents in a portable way.

Working with Paths

The `QDir` class is the key to handling paths and drives in Qt applications. When specifying paths to a `QDir` object, the slash (`/`) is used as a separator and is automatically converted to whatever separator is used on the current platform. Drive letters are allowed, and paths starting with a colon (`:`) are interpreted as references to resources embedded into the application.

The `QDir` static methods make it possible to easily navigate the file system. First, `QDir::current()` returns a `QDir` that refers to the application's working directory. `QDir::home()` returns a `QDir` for the user's home directory. `QDir::root()` returns the root, and `QDir::temp()` returns the directory for temporary files. `QDir::drives()` returns a `QList` of `QFileInfo` objects, representing the roots of all the available drives.

Note Unix systems are considered to have a single drive `/`, whereas a Windows machine's drive space can be configured to have several drives.

`QFileInfo` objects are used to hold information about files and directories. It has a number of useful methods, some of which are listed here:

- `isDir()`, `isFile()`, and `isSymLink()`: Return `true` if the file information object represents a directory, file, or symbolic link (or a shortcut on Windows).
- `dir()` and `absoluteDir()`:Return a `QDir` object represented by the file information object. The `dir` method can return a directory relative to the current directory, whereas `absoluteDir` returns a directory path starting with a drive root.
- `exists()`: Returns `true` if the object exists.
- `isHidden()`, `isReadable()`, `isWritable()`, and `isExecutable()`:Return information about the file's state.
- `fileName()`:Returns the file name without the path as a `QString`.
- `filePath()`:Returns the file name including the path as a `QString`. The path can be relative to the current directory.
- `absoluteFilePath()`:Returns the file name including the path as a `QString`. The path starts with a drive root.
- `completeBaseName()` and `completeSuffix()`:Return `QString` objects holding the name of the file and the suffix (extension) of the file name.

Let's use these methods to create an application listing all drives and folders in the root of each drive. The trick is to find the drives using `QDir::drives` and then find the directories of each drive's root (see Listing 8-1).

Listing 8-1. *Listing the drives with the root directories*

```
#include <QDir>
#include <QFileInfo>

#include <QtDebug>

int main( int argc, char **argv )
{
  foreach( QFileInfo drive, QDir::drives() )
  {
    qDebug() << "Drive: " << drive.absolutePath();

    QDir dir = drive.dir();
    dir.setFilter( QDir::Dirs );

    foreach( QFileInfo rootDirs, dir.entryInfoList() )
      qDebug() << "  " << rootDirs.fileName();
  }

  return 0;
}
```

The QDir::drives method returns a list of QFileInfo objects that are iterated using foreach. After having printed the drive's root path through qDebug, the QDir object for each root is retrieved using the dir method.

■ **Note** To use qDebug in a Windows environment, you must add the line CONFIG += console to your project file.

One nice aspect of QDir objects is that they can be used to get a directory listing. By using the filter() method, you can configure the object to return only directories. The directories are then returned as a QList of QFileInfo objects from the entryInfoList method. These QFileInfo objects represent directories, but the fileName method still returns the directory name. The isDir and isFile methods make it possible to confirm that the file name is a directory name or the name of a file. This is easier to understand if you consider directories to be files containing references to their contents.

The setFilter(Filters) method can be used to filter out directory entries based on a number of different criteria. You can also combine the filters criteria to get the entry list you want. The following values are supported:

QDir::Dirs: Lists directories that are matched by the name filter.

QDir::AllDirs: Lists all directories (does not apply the name filter).

QDir::Files: Lists files.

QDir::Drives: Lists drives. It is ignored on Unix systems.

QDir::NoSymLinks: Does not list symbolic links. It is ignored on platforms in which symbolic links not are supported.

QDir::NoDotAndDotDot: Does not list the special entries . and ..

QDir::AllEntries: Lists directories, files, drives, and symbolic links.

QDir::Readable: Lists readable files. It must be combined with Files or Dirs.

QDir::Writeable: Lists writable files. It must be combined with Files or Dirs.

QDir::Executable: Lists executable files. It must be combined with Files or Dirs.

QDir::Modified: Lists files that have been modified. It is ignored on Unix systems.

QDir::Hidden: Lists files that are hidden. On Unix systems, it lists files starting with .

QDir::System: Lists system files.

QDir::CaseSensitive: The name filter should be case sensitive if the file system is case sensitive.

The filter method is combined with the setNameFilters() method, which takes a QStringList of file name–matching patterns such as *.cpp. Notice that the name filter is a list of patterns, so it is possible to filter for *.cpp, *.h, *.qrc, *.ui, and *.pro files with one name filter.

Working with Files

You can use QDir to find files and QFileInfo to find out more about files. To take it one step further to actually open, read, modify and create files, you have to use the QFile class.

Let's start looking at QFile by checking out Listing 8-2. The application checks whether the file testfile.txt exists. If it does, the application attempts to open it for writing. If that is allowed, it simply closes the file again. Along the way, it prints status messages using qDebug.

The highlighted lines in the listing show the interesting QFile operations. First, the file name is set in the constructor. The file name can be set using the setFileName(const QString&) method, which makes it possible to reuse a QFile object. Next, the application uses the exists method to see whether the file exists.

The last highlighted line attempts to open the file for writing because it is easy to write-protect a file on all platforms supported by Qt. The open method returns true if the file is successfully opened.

The rest of the listing consists of code for outputting debug messages and exiting the main function (using return). Make sure to close the file before exiting if the opening of the file was successful.

Listing 8-2. *Basic* QFile *operations*

```
#include <QFile>

#include <QtDebug>

int main( int argc, char **argv )
{
  QFile file( "testfile.txt" );

  if( !file.exists() )
  {
    qDebug() << "The file" << file.fileName() << "does not exist.";
    return -1;
  }

  if( !file.open( QIODevice::WriteOnly ) )
  {
    qDebug() << "Could not open" << file.fileName() << "for writing.";
    return -1;
  }

  qDebug() << "The file opened.";
```

```
    file.close();

    return 0;
}
```

The previous listing opened the file for writing. You can use other flags when opening files to control how the file is read and modified:

- `QIODevice::WriteOnly`: Opens the file for writing.
- `QIODevice::ReadWrite`: Opens the file for reading and writing.
- `QIODevice::ReadOnly`: Opens the file for reading.

The preceding three flags can be combined with the following flags to control the file access mode in detail:

- `QIODevice::Append`: Appends all written data to the end of the file.
- `QIODevice::Truncate`: Empties the file when it is opened.
- `QIODevice::Text`: Opens the file as a text file. When reading from the file, all line endings are translated to `\n`. When writing to the file, the line endings are converted to a format appropriate for the target platform (for example, `\r\n` on Windows and `\n` on Unix).
- `QIODevice::Unbuffered`: Opens the file without any buffering.

You can always tell which mode is used for a given `QFile` object by calling the `openMode()` method. It returns the current mode. For closed files, it returns `QIODevice::NotOpen`.

Working with Streams

After you have opened a file, it is more convenient to access it using a stream class. Qt comes with two stream classes: one for text files and one for binary files. By opening a stream to access a file, you can use redirect operators (`<<` and `>>`) to write and read data to and from the file. With streams, you also get around platform differences such as endianess and different line-ending policies.

Text Streams

With text streams, you can interface a file as you can from the C++ standard library—but with a twist. The twist is that the file is handled in a cross-platform manner so that line endings and other such details do not mess up the results when you move applications and files between different computers.

To create a text stream for a file, create a `QFile` object and open it as usual. It is recommended that you pass the `QIODevice::Text` flag with your read and write policy. After you open the file, pass a pointer to the file object to the constructor of a `QTextStream` object. The `QTextStream` object is now a stream to and from the file, depending on how the file was opened.

Listing 8-3 shows a `main` function that opens a file called `main.cpp` for reading as text. If the file is opened successfully, a text stream is created. At the end of the function, the file is closed.

Listing 8-3. *Opening a text stream for reading*

```
int main( int argc, char **argv )
{
  QFile file( "main.cpp" );
  if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) )
    qFatal( "Could not open the file" );

  QTextStream stream( &file );

...

  file.close();

  return 0;
}
```

Listing 8-4 shows a simple loop meant to be used in the `main` function from the previous listing. The loop uses `atEnd` to see whether the end of the file is reached. If not, a `QString` is read from the stream using the >> operator and then printed to the debug console.

The result of executing the loop shown will not look like the contents of the `main.cpp` file. Operator >> reads until the first white space is encountered. So the line `#include <QFile>` would be split into `#include` and `<QFile>`. Because `qDebug` adds a line break after each call, the example line would be printed over two lines on the debug console.

Listing 8-4. *Reading from a text stream word by word*

```
  while( !stream.atEnd() )
  {
    QString text;
    stream >> text;
    qDebug() << text;
  }
```

The solution is to either read the entire file, including both text and line breaks, by using the `readAll()` method on the stream object or to read it line by line. Reading with `readAll()` works in most cases, but because the entire file is loaded into memory at once, it can easily use up the entire memory.

To read the file line by line, use the `readLine()` method, which reads a complete line at a time. Listing 8-5 shows the loop from the previous listing, but with `readLine` instead. Executing the loop gives a result on the debug console, showing the contents of the `main.cpp` file.

Listing 8-5. *Reading from a text stream line by line*

```
while( !stream.atEnd() )
{
  QString text;
  text = stream.readLine();
  qDebug() << text;
}
```

Data Streams

Sometimes you can't rely on using a text file for your data. For instance, you might want to support an already existing file format that is not text-based or you might want to produce smaller files. By storing the actual data in a machine-readable, binary format instead of converting it to human-readable text, you can save both file size and complexity in your save and load method.

When you need to read and write binary data, you can use the `QDataStream` class. There are two important matters you need to keep in mind when using data streams, however: data types and versioning.

With data types, you must ensure that you use exactly the same data type for the >> operator as for the << operator. When dealing with integer values, it is best to use `qint8`, `qint16`, `qint32`, or `qint64` instead of the `short`, `int`, and `long` data types that can change sizes between platforms.

The second issue, versioning, involves making sure that you read and write the data using the same version of Qt because the encoding of the binary data has changed between the different versions of Qt. To avoid this problem, you can set the version of the `QDataStream` with the `setVersion(int)` method. If you want to use the data stream format from Qt 1.0, set the version to `QDataStream::Qt_1_0`. When creating a new format, it is recommended to use the highest possible version (for Qt 4.2 applications, use `QDataStream::Qt_4_2`).

All the basic C++ types and most Qt types—such as `QColor`, `QList`, `QString`, `QRect`, and `QPixmap`—can be serialized through a data stream. To make it possible to serialize a type of your own, such as a custom `struct`, you need to provide << and >> operators for your type. Listing 8-6 shows the `ColorText` structure and the redirect operators for it. The structure is used for keeping a string and a color.

■ **Tip** When an object or data is *serialized*, it means that the object is converted into a series of data suitable for a stream. Sometimes this conversion is natural (for example, a string is already a series of characters); in other cases it requires a conversion operation (for example, a tree structure can't be mapped to a series of data in a natural way). When conversion is needed, a serialization scheme must be designed that defines how to serialize a structure and also how to restore the structure from the serialized data.

In this context, *type* means any type—a class, a structure, or a union. By providing the << and >> operators for such a type, you make it possible to use the type with a data stream without requiring any special treatment. If you look at the stream operators in the listing, you see

that they operate on a reference to a QDataStream object and a ColorText object, and return a reference to a QDataStream object. This is the interface that you must provide for all custom types that you want to be able to serialize. The implementation is based on using existing << and >> operators to serialize the type in question. Also remember to place the data on the stream in the same order in which you plan to read it back in.

If you want to write stream operators for a type of variable size—for example, a string-like class—you must first send the length of your string to the stream in your << operator to know how much information you need to read back using your >> operator.

Listing 8-6. *The* ColorText *structure with its* << *and* >> *operators*

```
struct ColorText
{
  QString text;
  QColor color;
};

QDataStream &operator<<( QDataStream &stream, const ColorText &data )
{
  stream << data.text << data.color;

  return stream;
}

QDataStream &operator>>( QDataStream &stream, ColorText &data )
{
  stream >> data.text;
  stream >> data.color;

  return stream;
}
```

Now that the custom type ColorText is created, let's try to serialize a list of ColorText objects: a QList<ColorText>. Listing 8-7 shows you how to do this. First, a list object is created and populated. Then a file is opened for writing before a data stream is created in the same manner as a text stream. The last step is to use setVersion to ensure that the version is properly set. When everything is set up, it is just a matter of sending the list to the stream by using the << operator and closing the file. All the details are sorted out by the different layers of << operators being called directly and indirectly for QList, ColorText, QString, and QColor.

Listing 8-7. *Saving a list of* ColorText *items*

```
  QList<ColorText> list;
  ColorText data;

  data.text = "Red";
  data.color = Qt::red;
  list << data;
```

```
...

  QFile file( "test.dat" );
  if( !file.open( QIODevice::WriteOnly ) )
    return;

  QDataStream stream( &file );
  stream.setVersion( QDataStream::Qt_4_2 );

  stream << list;

  file.close();
```

Loading the serialized data back is just as easy as serializing it. Simply create a destination object of the right type; in this case, use `QList<ColorText>`. Open a file for reading and then create a data stream. Ensure that the data stream uses the right version and reads the data from the stream using the >> operator.

In Listing 8-8, you can see that the data is loaded from a file, and the contents of the freshly loaded list are dumped to the debug console using `qDebug` from a `foreach` loop.

Listing 8-8. *Loading a list of* `ColorText` *items*

```
  QList<ColorText> list;

  QFile file( "test.dat" );
  if( !file.open( QIODevice::ReadOnly ) )
    return;

  QDataStream stream( &file );
  stream.setVersion( QDataStream::Qt_4_2 );

  stream >> list;

  file.close();

  foreach( ColorText data, list )
    qDebug() << data.text << "("
             << data.color.red() << ","
             << data.color.green() << ","
             << data.color.blue() << ")";
```

XML

XML is a meta-language that enables you to store structurized data in a string or text file (the details of the XML standard are beyond the scope of this book). The basic building blocks of an XML file are tags, attributes, and text. Take Listing 8-9 as an example. The `document` tag

contains the `author` tag and the text that reads `Some text`. The `document` tag starts with the opening tag `<document>` and ends with the closing tag `</document>`.

Listing 8-9. *A very simple XML file*

```
<document name="DocName">
  <author name="AuthorName" />
  Some text
</document>
```

Both tags have an attribute called `name` with the values `DocName` and `AuthorName`. It is possible for a tag to have any number of attributes, ranging from none to infinity.

The `author` tag has no contents and is opened and closed at once. Writing `<author />` is equivalent to writing `<author></author>`.

■ **Note** This information is the very least you need to know about XML. The XML file presented here is not even a proper XML file—it lacks a document type definition. And you haven't even started to learn about namespaces and other fun details of XML. But you do know enough now to start reading and writing XML files using Qt.

Qt supports two ways of handing XML files: DOM and SAX (described in the following sections). Before you get started, you need to know that the XML support is part of the Qt module `QtXml`, which means that you are required to add a line reading `QT += xml` to your project file to include it.

DOM

The document object model (DOM) works by representing the entire XML document as a tree of node objects in memory. Although it is easy to parse and modify the document, the entire file is loaded into memory at once.

Creating an XML File

Let's start by creating an XML file using the DOM classes. To make things easier, the goal is to create the document shown in Listing 8-9. The process is divided into three parts: creating the nodes, putting the nodes together, and writing the document to a file.

The first step—creating the nodes—is shown in Listing 8-10. The different building blocks of the XML file include a `QDomDocument` object representing the document, `QDomElement` objects representing the tags, and a `QDomText` object representing the text data in the `document` tag.

The elements and text object are not created using a constructor. Instead, you have to use the `createElement( const QString&)` and `createTextNode( const QString &)` methods of the `QDomDocument` object.

Listing 8-10. *Creating the nodes for a simple XML document*

```
QDomDocument document;

QDomElement d = document.createElement( "document" );
d.setAttribute( "name", "DocName" );

QDomElement a = document.createElement( "author" );
a.setAttribute( "name", "AuthorName" );

QDomText text = document.createTextNode( "Some text" );
```

The nodes created in Listing 8-10 are not ordered in any way. They can be considered to be independent objects, even though they all were created with same document object.

To create the structure shown in Listing 8-9, the `author` element and text have to be put in the document element by using the `appendChild( const QDomNode&)` method, as shown in Listing 8-11. In the listing, you can also see that the `document` tag is appended to the document in the same manner. It builds the same tree structure, as can be seen in the file that you are trying to create.

Listing 8-11. *Putting the nodes together in the DOM tree*

```
document.appendChild( d );
d.appendChild( a );
d.appendChild( text );
```

The last step is to open a file, open a stream to it, and output the DOM tree to it, which is what happens in Listing 8-12. The XML string represented by the DOM tree is retrieved by calling `toString(int)` on the `QDomDocument` object in question.

Listing 8-12. *Writing a DOM document to a file*

```
QFile file( "simple.xml" );
if( !file.open( QIODevice::WriteOnly | QIODevice::Text ) )
{
  qDebug( "Failed to open file for writing." );
  return -1;
}

QTextStream stream( &file );
stream << document.toString();

file.close();
```

Loading an XML File

Knowing how to create a DOM tree is only half of what you need to know to use XML through DOM trees. You also need to know how to read an XML file into a `QDomDocument` and how to find the elements and text contained in the document.

This is far easier than you might think. Listing 8-13 shows all the code it takes to get a QDomDocument object from a file. Simply open the file for reading and try to use the file in a call to the setContent member of a suitable document object. If it returns true, your XML data is available from the DOM tree. If not, the XML file was not valid.

Listing 8-13. *Getting a DOM tree from a file*

```
QFile file( "simple.xml" );
if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) )
{
  qDebug( "Failed to open file for reading." );
  return -1;
}

QDomDocument document;
if( !document.setContent( &file ) )
{
  qDebug( "Failed to parse the file into a DOM tree." );
  file.close();
  return -1;
}

file.close();
```

The root element of a DOM tree can be retrieved from the document object by using the documentElement() method. Given that element, it is easy to find the child nodes. Listing 8-14 shows you how to use firstChild() and nextSibling() to iterate through the children of the document element.

The children are returned as QDomNode objects—the base class of both QDomElement and QDomText. You can tell what type of node you are dealing with by using the isElement() and isText() methods. There are more types of nodes, but text and element nodes are most commonly used.

You can convert the QDomNode into a QDomElement by using the toElement() method. The toText() method does the same thing, but returns a QDomText instead. You then get the actual text using the data() method inherited from QDomCharacterData.

For the element object, you can get the name of the tag from the tagName() method. Attributes can be queried using the attribute(const QString &, const QString &) method. It takes the attribute's name and a default value. In Listing 8-14, the default value is "not set."

Listing 8-14. *Finding the data from the DOM tree*

```
QDomElement documentElement = document.documentElement();

QDomNode node = documentElement.firstChild();
while( !node.isNull() )
{
  if( node.isElement() )
  {
```

```
    QDomElement element = node.toElement();
    qDebug() << "ELEMENT" << element.tagName();
    qDebug() << "ELEMENT ATTRIBUTE NAME"
             << element.attribute( "name", "not set" );
  }

  if( node.isText() )
  {
    QDomText text = node.toText();
    qDebug() << text.data();
  }

  node = node.nextSibling();
}
```

Listing 8-14 simply lists the child nodes of the root node. If you want to be able to traverse DOM trees with more levels, you have to use a recursive function to look for child nodes for all element nodes encountered.

Modifying an XML File

Being able to read and write DOM trees is all you need to know in many applications. Keeping your application's data in a custom structure and translating your data into a DOM tree before saving and then extracting your data from the DOM tree when loading is usually enough. When the DOM tree structure is close enough to your application's internal structure, it is nice to be able to modify the DOM tree on the fly, which is what happens in Listing 8-15.

To put the code in the listing in a context, you need to know that the document has been loaded from a file before this code is run. When the code has been executed, the document is written back to the same file.

You find the root node using `documentElement`, which gives you a starting point. Then you ask the root node for a list of all `author` tags (all elements with the `tagName` property set to `author`) by using the `elementsByTagName(const QString &)` method.

If the list is empty, add an author element to the root node. The freshly created element is added to the root node using `insertBefore(const QDomNode &, const QDomNode &)`. Because you give an invalid `QDomNode` object as the second parameter to the method, the element is inserted as the first child node.

If the list contains an author element, you add a revision element to it. The revision element is given an attribute named `count`, whose value is calculated from the number of revision elements already in the author element.

That's all it takes. Because the nodes have been added to the DOM tree, you just need to save it again to get an updated XML file.

Listing 8-15. *Modifying an existing DOM tree*

```
QDomNodeList elements = documentElement.elementsByTagName( "author" );
if( elements.isEmpty() )
{
  QDomElement a = document.createElement( "author" );
```

```
    documentElement.insertBefore( a, QDomNode() );
  }
  else if( elements.size() == 1 )
  {
    QDomElement a = elements.at(0).toElement();

    QDomElement r = document.createElement( "revision" );
    r.setAttribute( "count",
                    QString::number(
                      a.elementsByTagName( "revision" ).size() + 1 ) );

    a.appendChild( r );
  }
```

Reading XML Files with SAX

The simple API for XML (SAX) can be used only to read XML files. It works by reading the file and locating opening tags, closing tags, attributes, and text; and calling functions in the handler objects set up to handle the different parts of an XML document. The benefit of this approach compared with using a DOM document is that the entire file does not have to be loaded into memory at once.

To use SAX, three classes are used: `QXmlInputSource`, `QXmlSimpleReader`, and a handler. Listing 8-16 shows the `main` function of an application using SAX to parse a file. The `QXmlInputSource` is used to provide a predefined interface between the `QFile` and the `QXmlSimpleReader` object.

The `QXmlSimpleReader` is a specialized version of the `QXmlReader` class. The simple reader is powerful enough to be used in almost all cases. The reader has a content handler that is assigned using the `setContentHandler` method. The content handler must inherit the `QXmlContentHandler`, and that is exactly what the `MyHandler` class does. Having set everything up, it is just a matter of calling the `parse(const QXmlInputSource *, bool)` method, passing the XML input source object as a parameter, and waiting for the reader to report everything worth knowing to the handler.

Listing 8-16. *Setting up a SAX reader with a custom handler class*

```
int main( int argc, char **argv )
{
  QFile file( "simple.xml" );
  if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) )
  {
    qDebug( "Failed to open file for reading." );
    return -1;
  }

  QXmlInputSource source( &file );

  MyHandler handler;
```

```
  QXmlSimpleReader reader;
  reader.setContentHandler( &handler );
  reader.parse( source );

  file.close();

  return 0;
}
```

The declaration of the handler class `MyHandler` can be seen in Listing 8-17. The class inherits from `QXmlDefaultHandler`, which is derived from `QXmlContentHandler`. The benefit of inheriting `QXmlDefaultHandler` is that the default handler class provides dummy implementations of all the methods that you otherwise would have had to implement as stubs.

The methods in the handler class get called by the reader when something is encountered. You want to handle text and tags and know when the parsing process starts and ends, so the methods shown in the class declaration have been implemented. All methods return a `bool` value, which is used to stop the parsing if an error is encountered. All methods must return `true` for the reader to continue reading.

Listing 8-17. *The* `MyHandler` *SAX handler class*

```
class MyHandler : public QXmlDefaultHandler
{
public:
  bool startDocument();
  bool endDocument();

  bool startElement( const QString &namespaceURI,
                     const QString &localName,
                     const QString &qName,
                     const QXmlAttributes &atts );
  bool endElement( const QString &namespaceURI,
                   const QString &localName,
                   const QString &qName );

  bool characters( const QString &ch );
};
```

All methods except `startElement` look more or less like the method shown in Listing 8-18. A simple text is printed to the debug console, and then `true` is returned. In the case of `endElement` (shown in the listing), an argument is printed as well.

Listing 8-18. *A simple handling class method*

```
bool MyHandler::endElement( const QString &namespaceURI, const QString &localName,
  const QString &qName )
{
  qDebug() << "End of element" << qName;
  return true;
}
```

The `startElement` method, shown in Listing 8-19, is slightly more complex. First, the element's name is printed; then the list of attributes passed through an `QXmlAttributes` object is printed. The `QXmlAttributes` is not a standard container, so you must iterate through it using an index variable instead of just using the `foreach` macro. Before the method ends, you return `true` to tell the reader that everything is working as expected.

Listing 8-19. *The* `startElement` *method lists the attributes of the element.*

```
bool MyHandler::startElement( const QString &namespaceURI, const QString &localName,
  const QString &qName, const QXmlAttributes &atts )
{
  qDebug() << "Start of element" << qName;
  for( int i=0; i<atts.length(); ++i )
    qDebug() << "  " << atts.qName(i) << "=" << atts.value(i);

  return true;
}
```

The reason for printing the `qName` instead of the `namespaceURI` or `localName` is that the `qName` is the tag name that you expect. Namespaces and local names are beyond the scope of this book.

It is not very complicated to build an XML parser by implementing a SAX handler. As soon as you want to convert the XML data into custom data for your application, you should consider using SAX. Because the entire document is not loaded at once, the memory requirements of the application are reduced, which might mean that your application runs more quickly.

Files and the Main Window

You learned in Chapter 4 that the setup with a `isSafeToClose` and the `closeEvent` method was a good starting point for giving the user the option to save the file when a window with a modified document is closed. Now the time has come to add support for that functionality to the SDI application (the same concept also applies to the MDI application).

Starting with Listing 8-20, you can see the changes made to the `SdiWindow` class declaration. The highlighted lines were added to handle the load and save functionality.

The change is made to add the menu items Open, Save, and Save As to the File menu. The changes to the class declaration consist of four parts: actions for handling the menu entries, slots for the actions, the functions `loadFile` and `saveFile` for loading and saving the document to an actual file, and the private variable `currentFilename` for keeping the current file name. All methods that have to do with saving documents return a `bool` value, telling the caller whether the document was saved.

Listing 8-20. *Changes made to the* `SdiWindow` *class to enable loading and saving documents*

```
class SdiWindow : public QMainWindow
{
  Q_OBJECT
```

```
public:
  SdiWindow( QWidget *parent = 0 );

protected:
  void closeEvent( QCloseEvent *event );

private slots:
  void fileNew();
  void helpAbout();

  void fileOpen();
  bool fileSave();
  bool fileSaveAs();

private:
  void createActions();
  void createMenus();
  void createToolbars();

  bool isSafeToClose();

  bool saveFile( const QString &filename );
  void loadFile( const QString &filename );
  QString currentFilename;

  QTextEdit *docWidget;

  QAction *newAction;
  QAction *openAction;
  QAction *saveAction;
  QAction *saveAsAction;
  QAction *closeAction;
  QAction *exitAction;

  QAction *cutAction;
  QAction *copyAction;
  QAction *pasteAction;

  QAction *aboutAction;
  QAction *aboutQtAction;
};
```

Creating the actions and then adding them to the appropriate menu is done in exactly the same way as for the already existing actions. The `fileOpen` method, connected to the open action, is shown in Listing 8-21. It uses the static `getOpenFileName` method from the `QFileDialog` class to get a file name. If the user has closed the dialog without choosing a file, the resulting string's `isNull` method returns `true`. In that case, you return from the slot without opening a file.

If an actual file name is retrieved, you can try to load the file using `loadFile`. However, if the current document has not been given a file name and is unchanged, the file is loaded into the current document. If the current document has a file name or has been modified, a new `SdiWindow` instance is created and then the file is loaded into it.

All `SdiWindows` are given file names when they are saved or loaded, so only new files do not have valid file names.

Listing 8-21. *Implementing the slot connected to the open action*

```
void SdiWindow::fileOpen()
{
  QString filename = QFileDialog::getOpenFileName( this );
  if( filename.isEmpty() )
    return;

  if( currentFilename.isEmpty() && !docWidget->document()->isModified() )
    loadFile( filename );
  else
  {
    SdiWindow *window = new SdiWindow();
    window->loadFile( filename );
    window->show();
  }
}
```

The `loadFile(const QString&)` method is used to load the contents from a given file into the document of the current window. The source code of the method is shown in Listing 8-22. The function attempts to open the file. If the file cannot be opened, a message box is shown for the user. If the file is opened, a `QTextStream` is created, and the entire file content is loaded by using `readAll`. The document is then assigned the new text with the `setPlainText` method. When the document has been updated, the `currentFilename` variable is updated, the modified flag is set to `false`, and the window's title is updated.

Listing 8-22. *Source code actually loading file contents into the document*

```
void SdiWindow::loadFile( const QString &filename )
{
  QFile file( filename );
  if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) )
  {
    QMessageBox::warning( this, tr("SDI"), tr("Failed to open file.") );
    return;
  }

  QTextStream stream( &file );
  docWidget->setPlainText( stream.readAll() );
```

```
  currentFilename = filename;
  docWidget->document()->setModified( false );
  setWindowTitle( tr("%1[*] - %2" ).arg(filename).arg(tr("SDI")) );
}
```

The opposite method of `loadFile` is `saveFile(const QString &)`. (You can see its implementation in Listing 8-23.) Despite their different tasks, the two functions' implementations look very similar. The concept is the same: attempt to open the file, send the document as plain text to a stream and update the `currentFilename`, reset the modified bit, and update the window title. When a file is actually saved, the `saveFile` function returns `true`; if the file is not saved, the function returns `false`.

Listing 8-23. *Source code for saving the document to a file*

```
bool SdiWindow::saveFile( const QString &filename )
{
  QFile file( filename );
  if( !file.open( QIODevice::WriteOnly | QIODevice::Text ) )
  {
    QMessageBox::warning( this, tr("SDI"), tr("Failed to save file.") );
    return false;
  }

  QTextStream stream( &file );
  stream << docWidget->toPlainText();

  currentFilename = filename;
  docWidget->document()->setModified( false );
  setWindowTitle( tr("%1[*] - %2" ).arg(filename).arg(tr("SDI")) );

  return true;
}
```

The return value from the `saveFile` method is used in the implementation of the `fileSaveAs` method shown in Listing 8-24. The Save As slot looks very much like the Open slot. It uses the `getSaveFileName` method to ask the user for a new file name. If a file name is selected, the `saveFile` method is called to try to save the document.

Notice that `false` is returned if the file dialog is canceled, and the return value from the `saveFile` method is returned when an attempt to save the document is made. The `saveFile` returns `true` only if the document actually has been written to the file.

Listing 8-24. *Source code for the Save As action*

```
bool SdiWindow::fileSaveAs()
{
  QString filename =
    QFileDialog::getSaveFileName( this, tr("Save As"), currentFilename );
```

```
  if( filename.isEmpty() )
    return false;

  return saveFile( filename );
}
```

The `fileSave` method tries to save the document to the same file as before—the name kept in `currentFilename`. If the current file name is empty, the file has not been given a file name yet. In this case, the `fileSaveAs` method is called, showing the user a File dialog to pick a file name. It is shown as source code in Listing 8-25.

The `fileSave` method returns the return value from either `saveFile` or `fileSaveAs`, depending on which method is used to save the file.

Listing 8-25. *Source code for the Save action*

```
bool SdiWindow::fileSave()
{
  if( currentFilename.isEmpty() )
    return fileSaveAs();
  else
    return saveFile( currentFilename );
}
```

The final option needed to make the dialog behave as expected is to let the user save the file from the warning dialog shown when a modified document is being closed. The new implementation of the `isSafeToClose` method is shown in Listing 8-26, in which the lines containing the actual changes are highlighted.

The first change is the addition of the Save option to the warning dialog using the `QMessageBox::Save` enumerated value. The other change consists of a case for handling the Save button. If the button is pressed, a call is made to `fileSave`. If the file is not saved (that is, `false` is returned), the close event is aborted. This makes it impossible for the user to lose a document without actually having chosen to do so (or experiencing some sort of power failure).

Listing 8-26. *Source code for checking whether to close a document*

```
bool SdiWindow::isSafeToClose()
{
  if( isWindowModified() )
  {
    switch( QMessageBox::warning( this, tr("SDI"),
      tr("The document has unsaved changes.\n"
         "Do you want to save it before it is closed?"),
          QMessageBox::Save | QMessageBox::Discard | QMessageBox::Cancel ) )
    {
    case QMessageBox::Cancel:
      return false;
    case QMessageBox::Save:
      return fileSave();
```

```
    default:
      return true;
    }
  }

  return true;
}
```

Adding these saving and loading capabilities fits well into the SDI structure presented earlier. By confirming that the document actually has been saved (by using the return value from all methods involved), you can build a waterproof protection, making it impossible to close an unsaved document without confirming to do so.

Summary

Using files on different platforms usually means trouble. The incompatibilities are found on all levels: file names, directory paths, line breaks, endianess, and so on. You can avoid problems with paths, drives, and file names by using the `QDir` and `QFileInfo` classes.

After you locate a file, you can open it by using `QFile`. Qt has streams to read and write data. If you use the `QTextStream` class, you can handle text files with ease; if you use the `QDataStream` class, it is easy to serialize and read back your data from binary files. Just think about the potential stream-versioning problem. Even if you use the same Qt versions for all your application deployments, you will get more versions in the future. A simple `setVersion` call can save days of frustration.

One alternative to storing your data as text or in a custom binary format is to use XML. Qt enables you to use DOM, which allows you to read an entire XML document into memory, modify it, and then write it back to a file. If you want to read an XML file without having to load it all at once, you can use Qt's SAX classes.

When you use XML, you need to add the line `QT += xml` to your project file because the XML support is implemented in a separate module. This module is not included in all editions of Qt, so verify that you have access to it before trying to use it.

Finally, you saw the missing piece of the SDI application. Adding the methods covered in the final section of this chapter makes it easy to build applications that support file loading and saving.