Chapter 3

C++, Qt, and Cascades

I have avoided discussing C++ until now and given you mostly a “QML/JavaScript” perspective of Cascades programming. My goal was to show you how easily you could quickly build applications using QML and JavaScript only, without even writing a single line of C++ code. By now you know enough about the Cascades programming model and it is time to look at what’s happening behind the scenes in the C++ world.

QML and JavaScript provide a quick and efficient way of declaratively designing your application’s UI and wiring some behavior for event handling. You will, however, reach a point where you will need to do some heavy lifting and implement core business logic in C++. The reasons for this can be manifold but they will almost certainly revolve around the following:

- Your application’s business logic is complex and you don’t want it scattered in QML documents.
- You need to achieve maximum performance, and JavaScript will simply not scale as well as C++ (for example, it would make no sense writing a physics engine in JavaScript).
- You need tight platform integration provided by Qt modules or BPS.
- You need to reuse third-party libraries written in C/C++.

C++ has the reputation of being a large and complex language, but the purpose of this chapter is to teach just enough so that you can efficiently build Cascades applications. I am actually going to make the bold assertion that all that you need to build Cascades applications is entirely covered in this chapter. The only prerequisite to understanding the material presented here is that you already have some OOP knowledge by having written applications in Java or Objective-C, and I will show you the equivalent C++/Qt way.

Note that the material will also strongly focus on the Qt C++ language extensions for writing applications. Cascades is heavily based on the Qt framework and therefore it is important that you have a good understanding of the underlying Qt framework. For example, I will tend to favor the Qt types, memory management, and container classes even if the standard C++ library provides equivalent functionality. (Another important reason is that the Qt containers are tightly integrated with QML and this will save us the pain of writing glue code to access standard C++ containers from QML.)

After having read this chapter, you will have a C++ perspective of Cascades programming and a good understanding of

- The Qt object model.
- Qt memory management techniques.
- The Qt container classes that you can access from QML.
- The different mechanisms for exposing C++ classes to QML.

C++ OOP 101

C++ has naturally evolved a great deal over the years and its current incarnation includes all the features required for modern software design. For example, memory management has been greatly simplified with smart pointers, and frameworks such as Qt drastically improve a programmer's productivity. The purpose of this section is to get you up and running with the OOP aspects of C++—namely support for classes, member functions, inheritance and polymorphism—so that you can quickly build Cascades applications without spending a couple of hours on a C++ tutorial.

C++ Class

Just like Java and Objective-C, C++ is a class-based language. A class serves as an abstraction for encapsulating functions and data (or in other words, a class is used to create new types in C++). Instances of the class are the objects that you pass around in your application and act upon by calling their methods. Usually, the class is separated between a header file providing the class definition, which includes the class's public interface, and an implementation file, which provides member function definitions (for example, in Cascades the application delegate definition is given by `applicationui.hpp`, and its implementation is given by `applicationui.cpp`). To illustrate C++ classes, let's consider the case of a financial instrument's pricing library. Pricing libraries are usually used by investment banks on Wall Street in order to price financial products such as options, bonds, and other kinds of derivative instruments (the pricing problem can actually become quite complex and is done by "rocket scientists" called *quants*). Quite naturally, the very first abstraction provided by a pricing library is the `Instrument` class, which will be the root abstraction for managing all financial products (see Listing 3-1).

Listing 3-1. Instrument.h

```
#ifndef INSTRUMENT_H_
#define INSTRUMENT_H_
#include <QObject>

class Instrument : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString symbol READ symbol WRITE setSymbol NOTIFY symbolChanged)
    Q_PROPERTY(double price READ price NOTIFY priceChanged)
public:
    Instrument(QObject* parent = 0);
    virtual ~Instrument();

    QString symbol() const;
    void setSymbol(const QString& symbol);

    virtual double price() const=0;
signals:
    void symbolChanged();
    void priceChanged();

private:
    QString m_symbol;
};
```

> **Note** C++, unlike Java, does not define or mandate a base class from which all classes must derive. However, in the following examples, I will be using QObject as a base class in order to illustrate its properties and emphasize its central role in Cascades programming.

Listing 3-1 is called a class definition. As mentioned previously, a class definition is provided in a header file (ending with an `.h` or `.hpp` extension) that declares the class's member functions and variables, as well as their visibility (`private`, `protected`, or `public`). Note that the `Instrument` class declares a *constructor* and a *destructor*. The `Instrument(QObject* parent=0)` constructor is used to initialize a class instance and the `~Instrument()` destructor is where you release resources owned by the object (such as dynamically allocated objects managed by the class instance). (Note that unlike Java, where the garbage collector handles memory management, in C++ you are in charge of memory management, and you must make sure that dynamically allocated resources are released when no longer needed.)

Besides the constructor and destructor, the class's public interface also includes:

- The `virtual double Instrument::price()=0` function, which is used to return the instrument's fair price. I will tell you more about this strange looking function in a moment.
- The `symbol` property, which is defined using the `Q_PROPERTY` macro. I will tell you more about the macro shortly. For the moment, simply keep in mind that it makes the corresponding property accessible from QML.
- The `symbolChanged()` signal, which is emitted when the corresponding symbol property is updated.
- The `priceChanged()` signal, which is emitted when the instrument's price changes.

Finally, the `Instrument` class inherits from `QObject`, which is part of the Qt framework (also note the presence of the `Q_OBJECT` macro, which tells the MOC compiler to generate additional code in order to support the signals and slots mechanism; see Chapter 1).

The `Instrument` class member function definition is given in a separate file, usually ending with the `.cpp` extension (see Listing 3-2).

Listing 3-2. Instrument.cpp

```
#include "Instrument.h"

Instrument::Instrument(QObject* parent) : QObject(parent), m_symbol(""){

}

Instrument::~Instrument() {
    // TODO Auto-generated destructor stub
}

void Instrument::setSymbol(const QString& symbol){
    if(m_symbol == symbol) return;
    m_symbol = symbol;
    emit symbolChanged();
}

QString Instrument::symbol() const{
    return m_symbol;
}
```

We first include the `Instrument.h` header file and then proceed by defining the member functions. The constructor first calls the QObject(QObject* parent) base class constructor and then initializes the class members using a *member initialization list* (in this case, there is only one class member, `m_symbol`, to initialize). As you can see, the file also defines the accessor functions for the `m_symbol` member variable. Finally, note how the `symbolChanged()` signal is emitted when `m_symbol` is updated. As you will see later in this chapter, the signal is used by the QML declarative engine to update properties bound to `Instrument`'s `symbol` property.

We can now try to use the newly created instrument class by creating a simple test application with a main function, which is the entry point of all C/C++ applications (see Listing 3-3).

Listing 3-3. main.cpp

```
int main()
{
    Instrument instrument;
}
```

If you try to compile the previous code, the compiler will complain with the following message:

```
../src/main.cpp:15:16: error: cannot declare variable 'instrument' to be of abstract type
'Instrument'
../src/Instrument.h:13:7: note: because the following virtual functions are pure within
'Instrument':
../src/Instrument.h:21:17: note: virtual double Instrument::price()
```

The compiler essentially tells you that it cannot instantiate the `Instrument` class because it contains a *pure virtual function*. You must be wondering what kind of a beast this is! Well, it is just a fancy way of saying that the method is abstract and that we have not provided an implementation. Also, marking a member function `virtual` tells the C++ compiler that a child class can override it. This is very important. By default, methods are statically resolved in C++. If you intend *polymorphic behavior*, then you need to flag the function as `virtual`. By appending the `=0` to the method declaration, you are telling the compiler that the method is abstract and you are not providing a default implementation. In effect, the class also becomes an abstract base class.

Note Listing 3-3 creates the Instrument instance on the stack as an automatic variable (in other words, the instrument will be automatically deleted as soon it runs out of scope). In C++ you can also dynamically allocate an object using the new operator. In that case, you will have to reclaim the memory when the object is no longer needed using the delete operator.

C++ Inheritance

So far so good; the `Instrument` class provides us with a convenient abstraction for managing financial instruments. However, for the pricing library to be useful, you need to extend it by building a hierarchy of concrete types. In finance you can literarily synthesize any instrument with a desired payoff (that's what quants do). However, the basic building blocks are bonds, stocks, and money accounts. You can use these instruments to create more or less complex derivatives such as options and swaps (that's why they are called derivatives, because their price derives from an underlying instrument). Let's extend the hierarchy to include stocks (see Listing 3-4).

Listing 3-4. Stock.h

```
#define STOCK_H_
#include "Instrument.h"

class Stock: public Instrument {
Q_OBJECT
Q_PROPERTY(double spot READ spot WRITE setSpot NOTIFY spotChanged)
public:
    Stock(QObject* parent = 0);
    virtual ~Stock();

    double spot();
    void setSpot(double spot);

    double price() const;
signals:
    void spotChanged();
private:
    double m_spot;
};

#endif /* STOCK_H_ */
```

As illustrated in the previous code, `Stock` inherits from the `Instrument` class and adds a new `spot` property, which corresponds to the stock's market price. The member function definitions are given by `Stock.cpp` (see Listing 3-5).

Listing 3-5. Stock.cpp

```
#include "Stock.h"

Stock::Stock(QObject* parent) : Instrument(parent), m_spot(0) {

}

Stock::~Stock() {
    // for illustration purposes only. Show that the destructor is called
    std::cout << "~Stock()" << std::endl;
}

double Stock::price() const{
    return spot();
}

double Stock::spot() const{
    return m_spot;
}

void Stock::setSpot(double spot){
    if(m_spot == spot) return;
    m_spot = spot;
    emit spotChanged();
}
```

The `Stock` constructor calls the `Instrument` base class constructor in order to initialize the base class object correctly (and once again, a constructor initialization list is used in order to initialize the `Stock` object's member variables). The `Stock.cpp` file also includes a concrete implementation of the `Instrument::price()` method, which simply returns the current spot or market price of the stock.

An option is a slightly more complex beast. A vanilla equity option gives you the right, but not the obligation, to buy (in the case of a call option) or sell (in the case of a put option) the underlying stock for a specific agreed-upon price sometime in the future. The parameters defining the current price of the option (i.e., the right to buy or sell the underlying stock in the future according to the terms of the option contract) are given by the following:

- The current spot price of the stock.
- The future agreed-upon *strike* price of the stock.
- The stock's *volatility*, which is a measure of its riskiness.

- The time to *maturity* of the contract expressed in years.
- The *risk-free rate*, which usually represents the interest rate on a three month US Treasury bill.

Using the previous input parameters, a neat little thing called the Black-Scholes formula gives you the option's fair value. Putting all of this together, our `Option` class definition is given in Listing 3-6.

Listing 3-6. Option.h

```
#ifndef OPTION_H_
#define OPTION_H_
#include "Instrument.h"

class Option: public Instrument {
    Q_OBJECT
    Q_ENUMS(OptionType)

    Q_PROPERTY(OptionType type READ optionType WRITE setOptionType NOTIFY typeChanged)
    Q_PROPERTY(double riskfreeRate READ riskfreeRate WRITE setRiskfreeRate NOTIFY
               riskfreeRateChanged)
    Q_PROPERTY(double spot READ spot WRITE setSpot NOTIFY spotChanged)
    Q_PROPERTY(double strike READ strike WRITE setStrike NOTIFY strikeChanged)
    Q_PROPERTY(double maturity READ timeToMaturity WRITE setTimeToMaturity
               NOTIFY maturityChanged)
    Q_PROPERTY(double volatility READ volatility WRITE setVolatility NOTIFY volatilityChanged)

public:
    enum OptionType {
        CALL, PUT
    };

    Option(QObject* parent = 0);
    virtual ~Option();

    double price() const;

    double riskfreeRate() const;
    void setRiskfreeRate(double riskfreeRate);

    double spot() const;
    void setSpot(double spot);

    double strike() const;
    void setStrike(double strike);
    double timeToMaturity() const;
    void setTimeToMaturity(double timeToMaturity);

    OptionType optionType() const;
    void setOptionType(OptionType type);
    double volatility() const;
    void setVolatility(double volatility);

signals:
    void priceChanged();
    void typeChanged();
    void spotChanged();
    void volatilityChanged();
    void strikeChanged();
    void riskfreeRateChanged();
    void maturityChanged();

private:
    OptionType m_type;
    double m_strike;
    double m_spot;
    double m_volatility;
    double m_riskfreeRate;
    double m_timeToMaturity;
};

#endif /* OPTION_H_ */
```

Here again, the `Option` class adds its own set of properties and notification signals. An option type is also defined using the `OptionType` enumeration, which is used to differentiate between put and call options (depending on the option type, the Black-Scholes price is different). Also note how the `Q_ENUMS` macro is used to export the enumeration to QML. Here again, the virtual `price` method is overridden to provide the option's Black-Scholes fair value (see Listing 3-7). You can simply skim over the implementation, which is only provided to illustrate how the different option parameters are used in the pricing. (Note that the `CND` function, which is an implementation of the cumulative distribution function, is not shown here.)

Listing 3-7. Option::price()

```
double Option::price() const {
    double d1, d2;

    d1 = (log(m_spot / m_strike)
                 + (m_riskfreeRate + m_volatility * m_volatility / 2)
                 * m_timeToMaturity)
                 / (m_volatility * sqrt(m_timeToMaturity));
    d2 = d1 - m_volatility * sqrt(m_timeToMaturity);

    switch (m_type) {
    case CALL:
        return m_spot * CND(d1)
               - m_strike * exp(-m_riskfreeRate * m_timeToMaturity) * CND(d2);
    case PUT:
        return m_strike * exp(-m_riskfreeRate * m_timeToMaturity) * CND(-d2)
               - m_spot * CND(-d1);
    default:
            //
            return 0;
    }
}
```

The methods used for updating the option's properties are straightforward. For example, Listing 3-8 illustrates how the `spot` property is updated:

Listing 3-8. Spot Property

```
double Option::spot() const {
    return m_spot;
}

void Option::setSpot(double spot) {
    if(m_spot == spot) return;
    m_spot = spot;
    emit spotChanged();
    emit priceChanged();
}
```

Also note that when a property is updated, besides emitting the corresponding property change signal, the `priceChanged()` signal is also emitted. This will play an important role when you will use the `Option` instance in QML bindings.

At this point, we have defined three abstractions in our class hierarchy: `Instrument`, `Stock`, and `Option`. Let's try to use them in practice. A small test program is given in Listing 3-9.

Listing 3-9. main.cpp

```
#include <iostream>
#include "Stock.h"
#include "Option.h"

int main()
{

    Stock stock;
    stock.setSymbol("orcl");
    stock.setSpot(50);

    Option option;

    option.setSymbol("myOption");
    option.setSpot(50);
    option.setStrike(55);
    option.setMaturity(0.5);
    option.setVolatility(0.2);
    option.setRiskfreeRate(.05);
    std::cout << "Stock price is: "  << stock.price() << std::endl;
    std::cout << "Option price is: " << option.price() << std::endl;

}
```

To display the program's output, I am using the standard C++ library by including the `iostream` header (`std::cout` is the standard output stream, which displays characters in a text console by default). The program's output is given as follows:

```
Stock price is: 50
Option price is: 1.45324
```

Polymorphism

We defined the `Stock` and `Option` class, but for our class library to be truly useful, we need to be able to manipulate them using the common base class `Instrument` interface. In practice, we care about being able to price instruments no matter the concrete type; whether it is a `Stock` or an `Option`. In other words, we want to be able to manipulate financial instruments using the base class `Instrument` abstraction. If the instrument is a `Stock`, it will return its market spot price, and if it's an `Option`, it will return the Black-Scholes price. This is exactly what we imply by polymorphism: the ability to implement the pricing logic differently depending on the underlying concrete type and being able to call at runtime the correct implementation using the `Instrument` base class abstraction. In C++, runtime polymorphic behavior is achieved using two mechanisms: references and pointers.

Using References

A reference is essentially an alias to an existing variable. Listing 3-10 shows you how to use a reference when pricing an option.

Listing 3-10. Using References

```
Option option;
option.setOptionType(Option::CALL);
option.setSymbol("myOption");
option.setSpot(50);
option.setStrike(55);
option.setTimeToMaturity(0.5);
option.setRiskfreeRate(.05);
option.setVolatility(.2);

Instrument& instr = option;

std::cout << "Instrument symbol is: " << instr.symbol().toStdString() << std::endl;
std::cout << "Instrument price is: " << instr.price() << std::endl;
```

As shown in Listing 3-10, `instr` is defined as a reference to an `Instrument` by adding an ampersand (&) after the type declaration (note that because a reference is an alias to an existing object, the definition must also include the referenced `Option` object). Finally, the `price()` method is called polymorphically using the `Instrument` base class interface (remember that price is a pure virtual function in `Instrument`'s class definition). The program's output is given as follows:

```
Instrument symbol is: myOption
Instrument price is: 1.45324
```

Another way of using references is by taking them as function parameters. For example, Listing 3-11 defines a `showInstrumentPrice()` function taking a reference to an `Instrument` (note the & indicating a pass-by-reference of the `instrument` parameter).

Listing 3-11. showInstrumentPrice

```
void showInstrumentPrice(const Instrument& instrument) {
    std::cout << "Instrument symbol is: " << instrument.symbol().toStdString() <<
                 " Instrument price is: " << instrument.price() << std::endl;
}

int main(){
    Stock stock;
    stock.setSymbol("myStock");
    stock.setSpot(50);

    Option option;
    option.setOptionType(Option::CALL);
    option.setSymbol("myOption");
    option.setSpot(50);
    option.setStrike(55);
    option.setTimeToMaturity(0.5);
    option.setRiskfreeRate(.05);
    option.setVolatility(.2);

    showInstrumentPrice(stock);
    showInstrumentPrice(option);
}
```

The `showInstrumentPrice` function takes a reference to an Instrument object. It does not know if the actual object is a `Stock` or an `Option`, but it knows that it can call the base class `Instrument::price()` method in order to get the instrument's price. Because `Instrument::price()` has been declared as virtual, the C++ runtime determines the correct price method to call using virtual function dispatch. The output of the application is given as follows:

```
Instrument symbol is: myStock, Instrument price is: 50
Instrument symbol is: myOption, Instrument price is: 1.45324
```

In other words, the `Instrument::price()` call is polymorphic and returns a different price depending on whether you pass a `Stock` or an `Option`. This only works because you are passing a reference to the `showInstrumentPrice()` method. If you try to change the `showInstrumentPrice` signature by removing the reference operator to `showInstrumentPrice(Instrument instrument)`, the C++ compiler will try to pass the `Instrument` parameter *by value*. The value semantics imply that a *copy* of the variable is passed to the function. The copy operation is done by calling a *copy constructor*, which is a special class constructor used for making a copy of a class instance. If you don't specify a copy constructor, the C++ compiler will generate one implicitly for you, which will do a member-wise copy of the source object.

There are several reasons why this will not work in the previous case:

- As explained, the compiler will try to generate a copy constructor. However, because `Instrument` is an abstract class, the C++ compiler cannot generate a copy.
- Let's suppose that Instrument did provide a default implementation for the `price()` method, always returning 0. Something more serious, called *object slicing*, would occur: only the base `Instrument` part of the object, whether it is a `Stock` or an `Option`, would be copied and passed to the `showInstrumentPrice()` function (the overridden `price` method would therefore be "sliced-off" and you would lose all polymorphic behavior. In other words, the function call would always return 0, no matter the concrete type passed to the function).
- There is a third reason why you can't pass an Instrument instance by value: `Instrument`'s base class is `QObject`, which does not support value semantics. (I will tell you more about value semantics when we discuss `QObject` identities. For the moment, suffice to say that because a `QObject`'s copy constructor is `private`, you cannot use it in order to make a copy of the class instance.)

Using Pointers

Now let's look at how polymorphism can be achieved using pointers. Listing 3-12 gives you an updated version of the test application using pointers (in other words, the objects are dynamically allocated on the heap).

Listing 3-12. Pointers

```
Stock* stock = new Stock;
stock->setSymbol("myStock");
stock->setSpot(50);

Option* option = new Option;
option->setSymbol("myOption");
option->setSpot(50);
option->setStrike(55);
option->setTimeToMaturity(0.5);
option->setVolatility(.2);
option->setRiskfreeRate(.05);

Instrument* instrument;

instrument = stock;
std::cout << "Instrument symbol is: " << instrument->symbol().toStdString() << std::endl;
std::cout << "Instrument price is: " << instrument->price() << std::endl;
```

```
delete instrument;

instrument = option;
std::cout << "Instrument symbol is: " << instrument->symbol().toStdString() << std::endl;
std::cout << "Instrument price is: " << instrument->price() << std::endl;

delete instrument;
```

This time we allocate the `Stock` and the `Option` on the *heap* using the `new` operator, which returns a pointer to the dynamically allocated object (in all of the examples until now we were allocating *automatic* objects on the *stack*). We also use an `Instrument` pointer (`Instrument*`) in order to polymorphically call the price method, which is resolved at runtime. The program's output is given as follows:

```
Instrument symbol is: myStock
Instrument price is: 50
Instrument symbol is: myOption
Instrument price is: 1.45324
```

Also note that the objects must be deleted when no longer needed, otherwise you will face a memory leak.

Note As illustrated in Listing 3-11, you must use the -> operator when accessing a class member with a pointer (accessing class members of a stack variable is done using the dot (.) operator).

This concludes our condensed overview of C++'s OOP features. The next sections will further concentrate on the Qt extensions to C++.

Qt Object Model

I very briefly mentioned the Qt object model when I presented the signals and slots mechanism in Chapter 1. The model extends standard C++ with runtime type introspection and metatype information, among other things.

Note C++ provides a limited form of runtime introspection with the `typeid` and `dynamic_cast` keywords. The Qt framework extensions provide a much richer version based on QObject and the MOC compiler.

The Qt object model adds the following features to standard C++ (your class must inherit from `QObject` and declare thee `Q_OBJECT` macro):

- Runtime type introspection using the `QMetaObject` class.
- A dynamic property system giving you the possibility to add properties at runtime to an instance of a `QObject` class.

- The signals and slots notification and interobject communication mechanism.
- A form of memory management using parent-child relationships. At any point you can set a child object's parent (this will effectively add the object to the parent's list of children). The parent will then take *ownership* of the child object and whenever the parent is deleted, it will also take care of deleting all of its children.

Meta-Object Compiler (MOC)

I already mentioned the MOC tool in Chapter 1, but I will do a quick recap here. The MOC parses a C++ header file and if it finds a class declaration containing the `Q_OBJECT` macro, generates additional code in order to add runtime introspection, signals and slots, and dynamic properties to that class (note that you have also encountered other macros such as `Q_PROPERTY`, `Q_ENUMS` and `Q_INVOKABLE` used by the MOC compiler in order to "enrich" a class's functionality). Note that when using the Momentics IDE, you don't need to take any additional steps to use the MOC tool, which is automatically called during the build process; it will scan all the header files located in the source folder of your project. (You can see this happening if you carefully inspect the console view during the build phase: if the class declaration is in a header file called `MyClass.h`, the MOC generated output will be created in `moc_MyClass.cpp` and dropped in a folder of your project tree. On the Mac, it's a hidden folder.)

QObject

`QObject` is essential in Qt/Cascades programming because it implements most of the functionality at the heart of the Qt object model discussed in the previous section. You have already informally encountered the `QObject::connect()` method in Chapter 1 in order to connect signals to slots. The purpose of this section is to give you additional details by reviewing other important `QObject` methods.

QObject::connect()

The `bool QObject::connect(const QObject* sender, const char* signal, const QObject* receiver, const char* slot, ConnectionType = AutoConnection)` method connects a `sender`'s `signal` to the receiver's slot. As you can see, the signal and slot parameters are C strings. You will therefore have to use the corresponding `SIGNAL()` and `SLOT()` macros in order to convert function signatures into strings. Behind the scenes, `QObject::connect()` compares the strings with introspection data generated by the MOC tool. Here is a simple example illustrating how to use the connect method:

```
QObject::connect(sender, SIGNAL(valueChanged(int)), receiver,
SLOT(setValue(int)).
```

Note that the connect method returns a `bool` value that you should always check to make sure that the connection was successful. During development, a best practice is to pass `QObject::connect()`'s return value to the `Q_ASSERT(bool test)` macro (the macro is enabled in debug builds; prints a warning message if the test fails and halts program execution). In practice, you should never ignore a failed connection because your application might behave erratically or crash in release versions.

As you might have guessed, the `QObject::connect()` mechanism happens at runtime without any type checking during the compilation process. In practice, this can be quite frustrating when you have to debug silently failing connections. As a general rule of thumb, if a `QObject::connect()` fails, check the following points:

- Make sure that the signal and slot parameter types correspond. A slot can take fewer parameters than an emitting signal and the extra parameters will be dropped; however, it is essential that the parameter types match.
- If a parameter type is defined in a namespace, make sure to use the fully qualified type name by including the namespace (see Listing 3-13).

Listing 3-13. QObject::connect()

```
QObject::connect(myImageView,
                 SIGNAL(imageChanged(bb::cascades::Image*)),
                 myHandler,
                 SLOT(onImageChanged(bb::cascades::Image*)));
```

Finally, you can also disconnect a signal from a slot using `QObject::disconnect(const QObject* sender, const char* signal, const QObject* receiver, const char* slot)`.

QObject::setProperty()

You can update `QObject` properties defined with the `Q_PROPERTY()` macro using the `QObject::setProperty(const char* propertyname, const QVariant& value)` method. A `QVariant` is a union of common Qt data types; however, at any time the `QVariant` can contain a single variable of a given type. If the property was not defined with the `Q_PROPERTY()` macro, `QObject::setProperty()` will create a new dynamic property and add it to the `QObject` instance. Similarly, you can get a `property`'s value using `QVariant QObject::property(const char* propertyname)`. As you will see later in this chapter, properties are a fundamental aspect of exchanging data between C++ and QML by using bindings (a binding can update a Cascades control's property when a corresponding C++ property changes or vice-versa, depending on the binding target).

QObject::deleteLater()

The `QObject::deleteLater()` method queues up your object for deletion in the Qt event thread. As a general rule of thumb, you should never delete heap-based objects in a slot if it has been passed as a parameter to the slot by the emitting signal (otherwise your application might crash because the object might still be required by other slots, for example). You might, however, face the situation where it is the slot's responsibility to discard the passed object when it is no longer needed. In that case, you can use `QObject::deleteLater()` to make sure that the object will be eventually deleted once control returns to the event loop (I will not get into the details of the Qt event loop, but if you apply the above-mentioned rule by not deleting heap-based objects in slots, you will always be on the safe side of the fence).

You will see examples of how to use `QObject::deleteLater()` in the section discussing `QThread` and you will also have ample opportunity to use the method in Chapter 7 when discarding `QNetworkReply` objects.

QObject::objectName()

The `objectName` property identifies an object by name. In practice, you can set a Cascades control's `objectName` in QML and then retrieve the object from the scene graph in C++ using the `QObject::findChild<T>()` method. For example, this is how the C++ code in Chapter 1 updated the TextView in Listing 1-3.

Note It is considered bad practice to directly access from C++ Cascades controls by objectName. The reason is that you will be introducing tight coupling between the UI controls and your C++ code. Instead, as you will see in the section dedicated to the model-view-controller pattern, the preferable way to interact between C++ and QML is to use signals and properties.

QObject Memory Management

QObjects organize themselves in parent-child relationships. You can always set a `QObject`'s parent either during construction or by explicitly calling the `QObject::setParent(QObject* parent)`. The parent then takes ownership of the `QObject` and adds it to its list of children. Whenever the parent is deleted, so are its children. This technique works particularly well for GUI objects, which tend to naturally organize themselves as object trees. Here are a few things to keep in mind when using the parent-child memory management technique:

- If you delete a `QObject`, its destructor will automatically remove itself from its parent's list of children.
- Signal and slots are also disconnected so that a deleted object cannot receive signals previously handled by the object.
- You should never mix memory management techniques when managing an object. For example, you should not manage the same object using parent-child relationships and a smart pointer (both techniques use separate reference counts and will conflict if used with the same object). You can, however, use smart pointers and parent-child relationships in the same application as long as they manage different objects (you can even use a smart pointer as a member variable of a `QObject` instance).

To further illustrate parent-child memory management, let's extend the Instrument class hierarchy by adding composite instruments. In finance, we usually use aggregates of instruments in order to represent things such as indices, portfolios, and funds. Let's therefore introduce a new type called `CompositeInstrument` (see Listing 3-14).

Listing 3-14. CompositeInstrument.h

```
#ifndef COMPOSITEINSTRUMENT_H_
#define COMPOSITEINSTRUMENT_H_

#include "Instrument.h"

class CompositeInstrument : public Instrument {
    Q_OBJECT

public:
    CompositeInstrument(QObject* parent=0);
    virtual ~CompositeInstrument();

    void addInstrument(Instrument* instrument);
    bool removeInstrument(Instrument* instrument);
    const QList<Instrument*>& instruments();
    double price() const;

signals:
    void instrumentAdded();
    void instrumentRemoved();

private:
    QList<Instrument*> m_instruments;
};

#endif /* COMPOSITEINSTRUMENT_H_ */
```

If you are into design patterns, you must have recognized an implementation of the Composite pattern, which lets you manage an aggregation of objects as a single object. Quite interestingly, these aggregate instruments are also called composites in finance. (Note that another good example of a composite class is the Cascades `Container`. Also in the example given in Listing 3-14, I am supposing that each instrument part of the composite is equally weighted. In practice, you could have different weights attributed to the instruments. For example, the Dow Jones Industrial Average is price weighted.)

Listing 3-15 gives you the `CompositeInstrument` member function definitions.

Listing 3-15. CompositeInstrument.cpp

```
#include "CompositeInstrument.h"
#include <iostream>
using namespace std;

CompositeInstrument::CompositeInstrument(QObject* parent) : Instrument(parent) {

}

CompositeInstrument::~CompositeInstrument() {
    // for illustration purposes only to show that the destructor is called
    cout << "~CompositeInstrument()" << endl;
}
```

```
void CompositeInstrument::addInstrument(Instrument* instrument){
    if(!m_instruments.contains(instrument)){
        m_instruments.append(instrument);
        instrument->setParent(this);
        emit instrumentAdded();
    }
}

bool CompositeInstrument::removeInstrument(Instrument* instrument){
    if(m_instruments.contains(instrument)){
        m_instruments.removeOne(instrument);
        instrument->setParent(0);
        emit instrumentRemoved();
        return true;
    }
    return false;
}

const QList<Instrument*>& CompositeInstrument::instruments(){
    return m_instruments;
}

double CompositeInstrument::price() const {
    double totalPrice = 0;
    for(int i = 0; i < m_instruments.length(); i++){
        totalPrice += m_instruments[i]->price();
    }
    return totolPrice;
}
```

The `CompositeInstrument` class uses a `QList<Instrument*>` instance in order to keep track of its instruments (a `QList<T>` is one of Qt's generic container classes; see the "Qt Container Classes" section).

Turning our attention to memory management, when a new `Instrument` is added to the composite, the composite takes ownership of the instrument using the `instrument->setParent(this)` method. Similarly, when an instrument is removed from the composite, the composite removes it from its list of children using `instrument->setParent(0)`. In practice, you should always document this kind of behavior so that it is clear to your clients who owns an object at any given time (for example, the Cascades documentation will always explicitly tell you who owns a control after it is added to or removed from another control).

Finally, Listing 3-16 shows you how to use the `CompositeClass` in a small test application.

Listing 3-16. main.cpp

```
int main(){
    Stock* stock = new Stock;
    stock->setSymbol("myStock");
    stock->setSpot(50);
    Option* option = new Option;
    option->setSymbol("myOption");
    option->setSpot(50);
    option->setStrike(55);
    option->setTimeToMaturity(0.5);
    option->setVolatility(.2);
    option->setRiskfreeRate(.05);

    CompositeInstrument* composite = new CompositeInstrument();
    composite->addInstrument(stock);
    composite->addInstrument(option);

    std::cout << "Composite price is: " << composite->price() << std::endl;

    delete composite;

    // more code goes here
}
```

The application's output is given as follows:

```
Composite price is: 51.4532
~CompositeInstrument()
Stock was deleted
Option was deleted
```

As you can see, the `Stock` instance and the `Option` instance are also deleted when the Composite instance is deleted, which illustrates how parent-child relationships work in practice.

Finally, note that parent-child relationships are distinct from the actual class hierarchy. You can set a `QObject`'s parent to any other `QObject` without having the objects sharing a direct inheritance relationship.

QObject Identity

`QObjects` feel strongly about their identity. In other words, you cannot use them as value objects. Having value semantics means for an object that only its value counts and that any copy of the object is equivalent. However, as mentioned previously, when considering pass-by-value semantics, `QObjects` cannot be copied or assigned. Before explaining how this is enforced, let me quickly recap two fundamental concepts that I brushed over when I mentioned pass-by-value semantics. In C++, you can define a copy constructor and an assignment operator. The copy constructor is used, for example, to pass the object by value to a function (or return an object by value from an function). The assignment operator (=) is used to assign one object to another (for example `obj1 = obj2`). I am not going to show you how to implement these operators but instead simply mention their signature:

- *Copy constructor*: The typical form of the copy constructor is `MyClass::MyClass(const MyClass& original)` and is used for creating a new copy of an existing instance. Typically, the copy constructor is called when passing an object by value to a function. Note that the copy constructor takes a constant reference to the original object in order to create the copy. If you do not provide a copy constructor, the compiler will implicitly create one for you doing a member-wise copy of the original object. Also note that you must pass a reference to the original object. The member-wise copy is problematic if your class contains pointers to dynamically allocated resources. In this case, the compiler-generated version of the constructor simply performs a "shallow" copy of the original object—resulting in all sorts of memory ownership problems.

- *Assignment operator*: The typical assignment operator is `const MyClass& MyClass::operator=(const MyClass& rhs)`. The assignment operator is called when you assign one object to another. Here again, if you do not provide one, the compiler will implicitly create an assignment operator for you, which does a member-wise copy of the original object.

Because a `QObject` is not intended to be assigned or copied, it disables the use of the copy constructor and assignment operator using the `Q_DISABLE_COPY(ClassName)` macro (the macro declares ClassName's copy constructor and assignment operator as private, so that you cannot use them).

To summarize, `QObjects` can only be used with reference semantics. In other words, you can pass around references or pointers to `QObjects` in your application without breaking the single identity constraint.

QVariant

A `QVariant` acts like a union of common Qt data types. However, at any time, a `QVariant` can only hold a single value of a given type (however, the value itself can be multivalued such as a list of strings). Also the type stored in a `QVariant` must have value semantics (in other words, it must at least define a public default constructor, a public copy constructor, and a public destructor). A `QVariant` is an essential component of Cascades programming because it is used in many different scenarios, such as parsing `JSON` and `XML` files or retrieving values from a database (you will see how to parse `XML` using the Cascades `XmlDataAccess` class in Chapter 6, and `JSON` using the Cascades `JsonDataAccess` class in Chapter 7). Most importantly, the QML declarative engine uses `QVariants` to pass C++ types to JavaScript and vice-versa (note that this happens transparently behind the scenes). You can store your own C++ type in a `QVariant` by registering it with the Qt type system using the `Q_DECLARE_METATYPE()` macro . Listing 3-17 illustrates typical `QVariant` usage.

Listing 3-17. QVariant

```
QVariant variant = 10;
if(variant.canConvert<int>()){
    std::cout << variant.toInt() << std::endl;
}

variant = "Hello variant";
if(variant.canConvert<QString>()){
    std::cout << variant.toString().toStdString() << std::endl;
}

// program output is
// 10
// Hello variant
```

Finally, you will often encounter the following QVariant-based types in Cascades development:

- `QVariantList`: A typedef for `QList<QVariant>`. Typically when parsing a `JSON` array, the `JsonDataAccess` class will return a `QVariantList`. You can also reference a `QVariantList` in QML as a JavaScript array.
- `QVariantMap`: A typedef for `QMap<QString, QVariant>`. Typically when parsing `JSON` objects, the `JsonDataAccess` class will return a `QVariantMap`. You can then access individual object attributes using the `QVariantMap`'s key.

The next section will give you more information about `QList` and `QMap`.

Qt Container Classes

C++ comes with the standard library, which is a collection of generic containers and algorithms for manipulating them. However, Qt also includes its own set of container classes that can be transparently accessed from QML. Note that the Qt container classes, just like their standard library counterparts, are *class templates* (in other words, you have to pass as a template parameter the type `T` stored in the container; you should be familiar with this if you have already used Java generics).

In Cascades programming, you will mostly use the `QList` and `QMap` containers. A `QList<T>` is a templated class for storing a list of values and provides fast index-based access as well as fast insertions. A `QMap<Key, T>` is a container for storing (key, value) pairs and provides fast lookup of the value associated with a key. You have already seen a `QList` in action in Listing 3-15, and Listing 3-18 gives you a quick overview of how to use a map in practice (you will also have the opportunity to see both containers in action in the code examples given in this book).

Listing 3-18. QMap

```
QMap<QString, int> integers;
integers["one"] = 1;
integers["ten"] = 10;
integers["five"] = 5;

QList<QString> keys = integers.keys();
for(int i=0; i< keys.length(); i++){
    cout << integers[keys.at(i)] << endl;
}
```

Note that you can store any value type in a `QMap`, including `QVariants` and pointers to `QObjects`.

Smart Pointers

I usually prefer to not worry about deleting objects; I would rather delegate the task. Like most difficult problems in programming, you can solve memory management by adding a level of indirection, which in this case is called *smart pointers*. Smart pointers are actually part of the new C++11 standard, but I am going to concentrate on the `QSharedPointer`, which is part of the Qt core framework. `QSharedPointer` is a *reference counting* smart pointer, meaning that it holds a shared reference to a dynamically allocated object. The pointee will be deleted once the last `QSharedPointer` pointing to it is destroyed or goes out of scope. Obviously, `QSharedPointers` must be automatic objects and you cannot allocate them on the heap. (Automatic objects are created on the stack and are destroyed when they get out of scope. To use a QSharedPointer, simply initialize it with a dynamically allocated resource, as shown in Listing 3-19.)

Listing 3-19. QSharedPointer

```
//don't forget to #include <QSharedPointer>

{ // start of scope

    QSharedPointer<MyClass> m_variable(new MyClass);
    m_variable->method1();  // calls MyClass::method1()
    m_variable->method2();  // calls MyClass::method2()

}  // end of scope. MyClass instance gets deleted here
```

As you can see, by automatically assigning a dynamically allocated object to a smart pointer, you don't need to worry anymore about deleting the object when it is no longer required. You can also assign a smart pointer to another one or return a smart pointer from a function (the reference count will be automatically handled for you in both cases). In other words, smart pointers make memory management as hassle free as in garbage-collected languages such as Java. Note that initializing a smart pointer, as illustrated previously, is a special case of the C++ "resource acquisition is initialization" (RAII) programming paradigm. RAII is particularly important in order to avoid memory leaks when exceptions happen during class construction. Listing 3-20 illustrates this by first using raw pointers in a class instantiation.

Listing 3-20. Constructor Exception, Raw Pointers

```
Class MyClass : public QObject{
Q_OBJECT
public:
    MyClass(QObject* parent=0) : QObject(parent){
        m_var1 = new Type1;
        m_var2 = new Type2;
    }

    virtual ~MyClass() {
        delete m_var1;
        delete m_var2;
    }
private:
    Type* m_var1;
    Type2*  m_var2;
};
```

The previous code declares two pointer member variables. Let's now imagine that an exception occurs during `m_var2`'s allocation (at this stage, `m_var1` has already been allocated). When an exception occurs in a constructor, it is as if the class instance never existed, and the destructor will not be called (in other words, the call to delete m_var1 will not happen and you will face a memory leak). If you are thinking of handling the exception in the constructor, don't; your code will become unreasonably convoluted and you would still not handle all possible cases. As you might have guessed, smart pointers are the solution. Listing 3-21 gives you a smart pointer version of the previous code.

Listing 3-21. Constructor Exception, Smart Pointers

```
Class MyClass : public QObject{
Q_OBJECT
public:
    MayClass(QObject* parent=0) : QObject(parent),
      m_var1(new Type), m_var2(new Type2)
    {
    }

    virtual ~MyClass() {
    // empty destructor.
    }
private:
    QSharedPointer<Type> m_var1;
    QSharedPointer<Type2> m_var2;
};
```

As illustrated in Listing 3-21, I am using initialization lists to initialize the smart pointers (initialization lists should be preferred when dealing with non-built-in types). So what happens when an exception occurs? As previously, your destructor does not get called but the C++ standard mandates that the destructor of all successfully constructed sub-objects have to be called. (This will effectively release the memory held by `m_var1` and avoid any leaks. Note that in the case of “dumb” pointers, the pointer is effectively deleted, but not the *pointee*; this is why your class needs a destructor in the first place.)

In practice, if you do not handle an exception, it is propagated up the call stack, and eventually, your program will be terminated by the C++ runtime. This could be the sensible thing to do if your application is in such a “catastrophic state” that it would be pointless to continue running (at the very least, you should create a log trace of the problem). Obviously, smart pointers would not be very helpful in such a situation, and the BlackBerry 10 OS would reclaim the memory anyway. However, if you need to write long-running applications such as headless apps, you need to make sure that your application is resilient; you cannot afford crashing when exceptions occur. Smart pointers will therefore be very useful to avoid memory leaks in exceptional cases by making sure that memory is released.

Exposing C++ Objects to QML

There are essentially four ways of exposing C++ objects to QML:

- You can use a `QDeclarativePropertyMap` to aggregate values in a map, and then set it as a context property of the QML document.
- You can selectively expose properties and methods from a `QObject` derived class, and then set the instance as a context property of the QML document.
- You can “attach” an instance of a `QObject` to a QML `UIObject` object using its `UIObject::attachedObjects` property in QML. Note that you will have to first register the `QObject` derived class with the QML type system.
- You can create a QML custom control in C++ by extending `bb::cascades::CustomControl`. You can then use the control as any other QML element in your document. Once again, you will have to register your control with the QML type system.

> **Note** To make sure that the document context properties are accessible from QML bindings, you need to set them *before* instantiating the scene's root object in the application delegate.

Before getting into the details of exposing C++ objects in practice, let's take a detailed look at the application delegate's constructor and explain the flow of events (see Listing 3-22). (We conveniently skimmed over this in Chapters 1 and 2, but now it is time to get our feet wet).

Listing 3-22. ApplicationUI.h

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
        QObject(app)
{
    // prepare the localization
    // code omitted here

    // Create scene document from main.qml asset, the parent is set
    // to ensure the document gets destroyed properly at shut down.
    QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);

    // Set the qml document context properties before creating root object using:
    // void QMLDocument::setContextProperty(const QString &propertyName, QObject *object)

    // Create root object for the UI
    AbstractPane *root = qml->createRootObject<AbstractPane>();

    // Set created root object as the application scene
    app->setScene(root);
}
```

Here is a step-by-step description of the code shown in Listing 3-22:

- `QmlDocument::create(const QString &qmlAsset, bool autoload= true)` is called and a QML document is loaded from the assets folder of your application. All documents loaded with this method will share the same *QML declarative engine*, which is an instance of a Qt `QDeclarativeEngine`.
- A *context* is also associated to the document. Contexts allow data to be exposed to components instantiated by the QML declarative engine (all documents loaded using the `QmlDocument::create()` method share the same instance of the declarative engine, which is associated with the application).
- Contexts form a hierarchy and the root of the hierarchy is the QML declarative engine's context. The context associated with the loaded document is therefore derived from the root context and shares its properties. Note that these properties are not the ones corresponding to `QObject` but the ones set with `QDeclarativeContext::setContextProperty(const QString &, QObject *)` method. You also have to be aware that you will override a property from the root context if you set it with a different value in the document context.

- A root node is instantiated for the scene graph represented by the QML document by calling the `QmlDocument::createRootObject<T>()` template method (the template T parameter must be pointer to a `UIObject` subclass).
- During the instantiation of the root node, the `UIObject::creationCompleted()` signal will be emitted for all UIObjects in the scene graph.

Now let's look at how the document context is used in practice for exposing C++ objects.

QDeclarativePropertyMap

A `QDeclarativePropertyMap` provides an extremely convenient and easy way to expose domain data or *value types* to the QML UI layer. You basically use an instance of a `QDeclarativePropertyMap` to set key-value pairs that can be used in QML bindings (the bindings are dynamic: whenever a key's value is updated, anything bound in QML to that key will also be updated). The values in the map are stored as `QVariant` instances. Using variants effectively means that you can expose to QML any type that can be "wrapped" as a `QVariant`. As mentioned previously, `QVariantList` and `QvariantMap` are two of the most interesting QVariant-based types because you can build arbitrarily complex data structures using them. Listing 3-23 illustrates this by building a person data structure.

Listing 3-23. ApplicationUI.cpp

```
QmlDocument::create("asset:///main.qml").parent(this);

QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);

QDeclarativePropertyMap* propertyMap = new QDeclarativePropertyMap;

QMap<QString, QVariant> person;
person["firstName"] = "John";
person["lastName"] = "Smith";
person["jobFunction"] = "Software Engineer";
person["age"] = 40;

QVariantList hobbies;
hobbies << "surfing" << "chess" << "cinema";

person["hobbies"] = hobbies;

propertyMap->insert("department", "Software Engineering");
propertyMap->insert("person", person);
qml->setContextProperty("mymap",propertyMap);
```

After having built the `QVariant` data structure, you simply add the QVariant to a `QDeclarativePropertyMap` instance using `QDeclarativePropertyMap::insert(const QString& keyname, const QVariant& value)`. You can then in turn add the map instance as a context property of the QML document using `QmlDocument::setContextProperty(const QString& mapName, QObject* propertyMap)`. In QML, you can finally reference the map by name, as shown in Listing 3-24.

Listing 3-24. main.qml

```
import bb.cascades 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            text: "Department: " + mymap.department;
            }
        }

        Label {
            // Localized text with the dynamic translation and locale updates support
            text: {
                return "last name: "+ mymap.person.lastName;
            }
        }
        Label{
            text:{
                 return "Age: " + mymap.person.age;
            }
        }
        Label{
            text:{
                return "Job function: " + mymap.person.jobFunction;
            }
        }

        Label{
            text: {
                var hobbies = mymap.person.hobbies;
                var s = "Hobbies: ";
                for (var i = 0; i< hobbies.length; i++){
                    s = s + hobbies[i] + " ";
                }
                return s;
            }
        }
    }
}
```

To extract the values stored in the map, you use the `mapname.keyname` “dot notation” syntax (note that in the specific case of the person key, the value returned is also a map and you have to reapply the dot notation in order to retrieve the associated values).

Exposing QObjects

As explained in the previous section, using `QDeclarativePropertyMap` is a great way to expose data structures based on common QML “basic types.” There will be times, however, where you will need to expose your own C++ objects directly so that you can achieve more complex behaviors, such as calling the object's methods or handling its signals in QML (or vice-versa, let the object handle signals emitted from QML). Typically, such objects play the role of application delegates or service façades (I will tell you more about this in the section dedicated to the model-view-controller pattern).

When exposing some functionality to QML, you should always think in terms of services and granularity. For example, if you need to access a large C++ library from QML, it is often preferable to define a set of coarse-grained services that you expose to the QML layer instead of trying to expose every single class of your library. By doing so, you will be able to define clear boundaries between the QML layer and your C++ types. This will also avoid leaking the internals of your class library to the QML layer, thus providing the additional benefit of decoupling your UI logic from the C++ application logic. Once you have decided on your services' granularity, you will be able to design your `QObject` based C++ service classes using the following recipe:

- Identify the class properties that you want to access from QML.
- Identify the class signals that you want to handle in QML.
- Identify any slots and class methods that should be called from QML.
- When implementing your class methods, use types that you can pass as `QVariants`.

In practice, in order to expose a C++ class instance to QML, you need to do the following:

- Add the `Q_OBJECT` macro at the start of the class declaration (and, of course, your class must inherit from QObject).
- Use the `Q_PROPERTY` macro in order to expose class properties to QML.
- Use the `Q_INVOKABLE` macro in order to expose class methods to QML.
- Signals and slots are automatically exposed using the `signals:` and `slots:` annotations, as explained in Chapter 1.

The syntax for declaring object properties with the `Q_PROPERTY` macro is as follows:

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

The only mandatory values are the property type, name, and the getter function for reading the property. In practice, you will be using a much shorter version of the macro:

```
Q_PROPERTY(type name READ getFunction WRITE setFunction NOTIFY notifySignal)
```

> **Note** You must specify the `notifySignal` if you intend on using the property in QML bindings, which I will explain shortly (you must also emit the signal when the property changes).

Using the Document Context

If you carefully study the `Option` class given in Listing 3-6, you will notice that we have already defined the class in such a way that it can be readily used from QML. In fact, just like the `QDeclarativePropertyMap` instance, all you simply need to do is to add an `Option` instance to the QML document context property from C++ (see Listing 3-25 and Listing 3-26).

Listing 3-25. ApplicationUI.hpp

```
class ApplicationUI : public QObject
{
    Q_OBJECT
public:
    ApplicationUI(bb::cascades::Application *app);
    virtual ~ApplicationUI() { }
private:
    Option* m_option;
};
```

Listing 3-26. ApplicationUI.cpp

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
        QObject(app), m_option(new Option(this))
{

    // Create scene document from main.qml asset, the parent is set
    // to ensure the document gets destroyed properly at shut down.
    QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);

    qml->setContextProperty("_option", m_option);

    // Create root object for the UI
    AbstractPane *root = qml->createRootObject<AbstractPane>();

    // Set created root object as the application scene
    app->setScene(root);
}
```

The `main.qml` document referencing the Option instance is given in Listing 3-27.

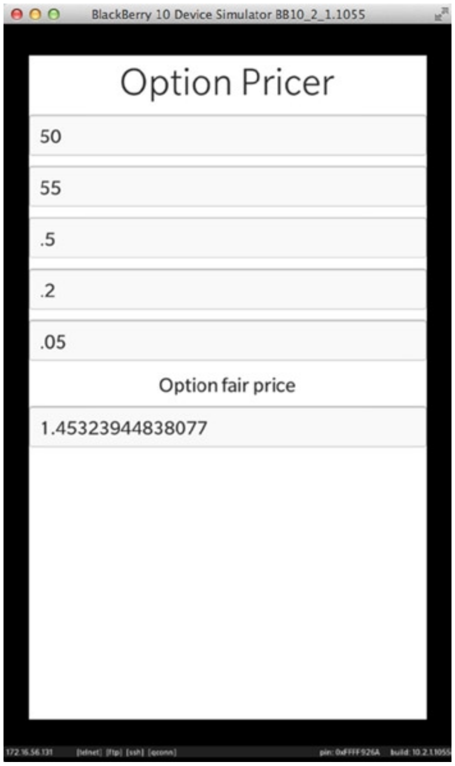
Listing 3-27. main.qml

```
import bb.cascades 1.2
Page {
    Container {
        //Todo: fill me with QML
        Label {
            text: "Option Pricer"
            horizontalAlignment: HorizontalAlignment.Center
            textStyle.base: SystemDefaults.TextStyles.BigText
        }
        TextField {
            id: spotField
            hintText: "Enter spot price"
            onTextChanging: {
                _option.spot = text;
            }
        }
        TextField {
            id: strikeField
            hintText: "Enter strike price"
            onTextChanging: {
                _option.strike = text;
            }
        }
        TextField {
            id: maturityField
            hintText: "Enter time to maturity"
            onTextChanging: {
                _option.maturity = text;
            }
        }
        TextField {
            id: volatilityField
            hintText: "Enter underlying volatility"
            onTextChanging: {
                _option.volatility = text;
            }
        }
        TextField {
            id: riskfreeRateField
            hintText: "Enter risk free rate"
            onTextChanging: {
                _option.riskfreeRate = text;
            }
        }
        Label {
            text: "Option fair price"
            horizontalAlignment: HorizontalAlignment.Center
        }
        TextField {
            id: priceField
            text: _option.price
        }
    }
}
```

Here is a brief description of the code shown in Listing 3-27:

- The `TextFields`' `textChanging` signals are used to update the corresponding `Option` object's properties.
- As mentioned previously, when any of the option's properties is updated, an `Instrument::priceChanged()` signal is also emitted by the `Option`.
- The priceField's text property is bound to the corresponding Instrument::price property (the QML declarative engine will therefore update the QML property when the Instrument`::priceChanged()` signal is emitted).

The resulting application UI is given in Figure 3-1.


BlackBerry 10 Device Simulator BB10_2_1.1055
Option Pricer

50

55

.5

.2

.05

Option fair price

1.45323944838077

172.16.56.131 [telnet] [ftp] [ssh] [qconn] pin: 0xFFFF926A build: 10.2.1.1055

***Figure 3-1.** Option pricer UI*

Using the attachedObjects Property

I am now going to show you how to use the `Option` class as a `UIObject's attachedObjects` property. You first need register the `Option` class with the QML type system (usually, you will do this in `main.cpp`; see Listing 3-28).

Listing 3-28. main.cpp

```
Q_DECL_EXPORT int main(int argc, char **argv)
{

    qmlRegisterType<Option>("ludin.instruments", 1, 0, "OptionType");

    Application app(argc, argv);

    // Create the Application UI object, this is where the main.qml file
    // is loaded and the application scene is set.
    new ApplicationUI(&app);

    // Enter the application main event loop.
    return Application::exec();
}
```

The call to `qmlRegisterType<Option>("ludin.instruments", 1, 0, "OptionType")` effectively registers the `Option` C++ type with the QML type system and the corresponding QML type `OptionType`.

To actually use the type in `main.qml`, you need to import the `ludin.instruments` namespace and declare an `OptionType` object as a `UIObject's` attachedObjects property (see Listing 3-29).

Listing 3-29. OptionType

```
import bb.cascades 1.2
import ludin.instruments 1.0

Page {
    Container {
        //Todo: fill me with QML
        Label {
            text: "Option Pricer"
            horizontalAlignment: HorizontalAlignment.Center
            textStyle.base: SystemDefaults.TextStyles.BigText
        }
        TextField {
            id: spotField
            hintText: "Enter spot price"
        }
        TextField {
            id: strikeField
            hintText: "Enter strike price"
        }
        TextField {
            id: maturityField
            hintText: "Enter time to maturity"
        }
        TextField {
            id: volatilityField
            hintText: "Enter underlying volatility"
        }
        TextField {
            id: riskfreeRateField
            hintText: "Enter risk free rate"
        }
        Label {
            text: "Option fair price"
            horizontalAlignment: HorizontalAlignment.Center
        }
        TextField {
            id: priceField
            text: option.price
        }
        attachedObjects: [
            OptionType {
                id: option
                type: OptionType.CALL
                symbol: "myoption"
                spot: spotField.text
                strike: strikeField.text
                maturity: maturityField.text
                volatility: volatilityField.text
                riskfreeRate: riskfreeRateField.text
            }
        ]
    }
}
```

Using Bindings

You should note that unlike Listing 3-28, you are not using signals and slots to update the controls in the scene graph. In fact, everything is done using bindings and the net result is that the UI code is mostly declarative. As illustrated in the code, the QML `OptionType` object's properties are bound to the corresponding `TextFields`' text properties. Similarly, the priceField's `text` property is bound to the `OptionType` object's `price` property (note that the QML declarative engine automatically transforms the numeric value of the `price` property into a string before setting the `TextField`'s `text` property). Whenever a property changes in C++, the QML declarative engine updates the corresponding bound property in QML. In other words, by using bindings, you have delegated the mundane task of updating your application's controls' to the QML declarative engine (this also results in cleaner QML requiring less maintenance).

Model-View-Controller

An important point to consider when designing Cascades applications is the way your C++ code will interact with the QML UI layer. Typically, graphical user interface frameworks promote the model-view-controller (MVC) pattern, which separates your application's logic in three distinct responsibilities (see Figure 3-2).

- Models are responsible for managing your application's data and provide an abstraction layer for accessing and updating it. Typically, they represent the domain objects in your application. Models don't know how to display themselves. However, they can notify controllers and views when their state changes.
- Views are the visual representation of your application data. The same data can be represented by multiple views in different ways, such as a chart or a list of values. Views are displayed to the user.
- A controller effectively plays the role of a mediator between the model and the view. It handles user input and updates the model and view accordingly. In simple applications, you will usually have a single controller; but in more complex scenarios, nothing stops you from having multiple task-oriented controllers.

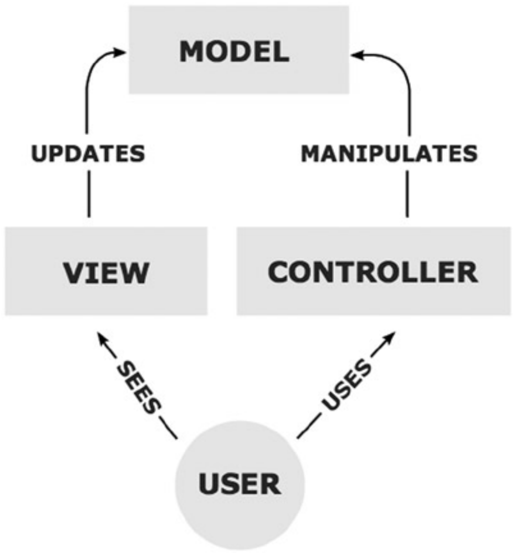


Figure 3-2. MVC interactions

When a model's state changes, it notifies its associated controllers and views so that they can handle the new state. Depending on the degree of separation you may want to achieve, you can also enforce that all model interactions go strictly through the controller. The most fundamental idea is that controllers and views depend on the model, but the opposite is not true. Models are therefore truly independent elements of your applications.

The Cascades framework does not enforce the MVC pattern. For example, there is no controller class to extend. However, Cascades is sufficiently flexible so that you design your application using the MVC pattern, should you choose so. Figure 3-3 illustrates the fundamental elements of a standard Cascades application.

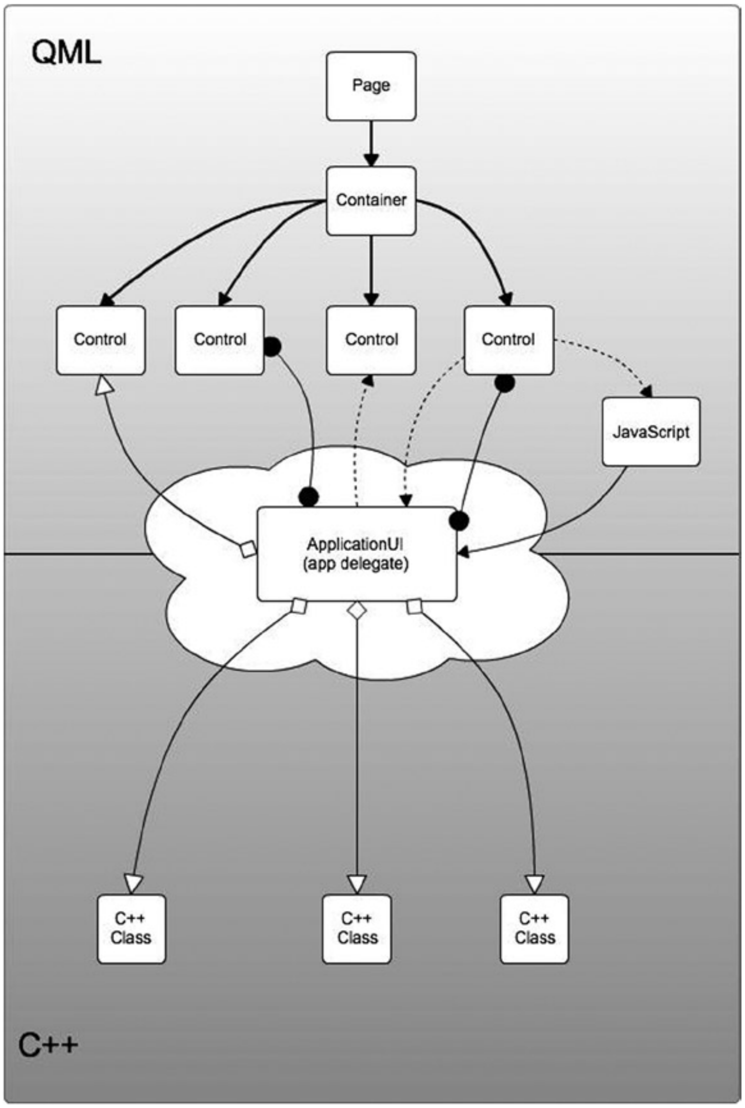


***Figure 3-3.** Cascades application elements*

The QML layer shows a typical scene graph consisting of a root `Page` control and a `Container` with multiple children. Signals and slots are represented using dashed arrows (the signal is the start of the arrow and the corresponding slot is the end). Property bindings are the links shown with a full dot on both sides. Direct references to an element are shown as arrows with an empty diamond at their start. As illustrated in Figure 3-3, you can break up your application in a C++ layer that contains your application business logic and a QML layer that contains your application's views (typically, user interactions (such as a clicked button) is handled in the QML layer using JavaScript).

As mentioned previously, it is always a good idea to expose C++ logic to the QML layer using coarse-grained services. This is the reason why the application delegate is your central entry point to the C++ application logic (the cloud symbol represents the QML document context from which you can access the application delegate). Interactions between the application delegate and QML controls should be essentially done using signals and slots and property bindings (as shown in Figure 3-3, you can also directly access a UI control from C++ in your application delegate, but this is strongly discouraged). Mapping this to the MVC pattern, you can see that you have lots of flexibility in defining where your controllers reside. For example, you could decide that the application delegate is your sole controller that handles all interactions between UI controls and the domain model.

Alternatively, you could also split the controllers between JavaScript and the application delegate. Finally, you could also use property bindings between your application delegate and UI controls exclusively. In this case, the only interactions between the UI layer and the application delegate would happen through property updates (in other words, this is would be a form of reactive programming where the data flow between C++ and QML governs the application's state).

Application Delegate

Until now, I have used the term "application delegate" in a relatively informal way without really explaining what I meant. The application delegate is the ApplicationUI class generated for you by the New Cascades Application Wizard. The class's responsibility is to load the QML scene graph from `main.qml`, wire signals and slots between UI controls and domain objects and add itself to the QML document context if necessary. The application delegate therefore plays a central role in a Cascades application. Here again, Cascades does not enforce the presence of an application delegate and you could simply load `main.qml` in your application's main function. However, centralizing the interactions between UI controls and C++ domain objects in a dedicated object will greatly simplify your application's design in the long run. The role of the application delegate is therefore to

- Define signals reflecting the state of model objects used for updating Cascades controls.
- Define slots used by the QML layer in order to update the domain model according to user interactions.
- Define properties used in QML bindings. The properties can be used to selectively expose `QObject` subclasses to the QML layer of your application. There are really no limitations in what the properties can represent. For example, a property could be a `DataModel` used by a `ListView` in order to display a list of items (see Chapter 6) and another property could represent a list of contacts from the contacts database (see Chapter 8), and so on.
- Centralize all interactions between QML and C++ objects (in other words, use the application delegate as your main app controller).

To illustrate the previous points, Listing 3-30 shows you a hypothetical application delegate definition for our financial instruments. (I will not provide the member function definitions. The most important point to keep in mind is how the application delegate is used as an interface to the C++ data model. Also note that the properties' accessors are defined inline.)

Listing 3-30. Application Delegate

```
#include "Stock.h"
#include "Option.h"
#include "CompositeInstrument.h"

class ApplicationUI : public QObject
{
    // used for displaying  instruments in ListView
    Q_PROPERTY(bb::cascades::ArrayDataModel* READ instrumentsModel CONSTANT)

    Q_PROPERTY(QList<CompositeInstrument*> READ composites NOTIFY compositesChanged)
    Q_PROPERTY(QList<Option*> options READ options NOTIFY optionsChanged)
    Q_PROPERTY(QList<Stocks*> stocks READ stocks NOTIFY stocksChanged)

public:
    ApplicationUI(bb::cascades::Application *app);
    virtual ~ApplicationUI() { }

    // load financial instruments in ArrayDataModel
    Q_INVOKABLE void loadInstruments() { // code not shown};
signals:
    void compositesChanged();
    void stocksChanged();
    void optionsChanged();

private:
    bb::cascades::ArrayDataModel* dataModel() {return m_instrumentsModel};

    QList<CompositeInstrument*> composites() {return m_composites};
    QList<Option*> options() {return m_options};
    QList<Stocks*> stocks() {return m_stocks};

    QList<Stock*> m_stocks;
    QList<Option*> m_options;
    QList<CompositeInstrument*> m_composites;

    bb::cascades::ArrayDataModel* m_instrumentsModel;

};
```

The properties defined in the application delegate are accessible from QML and represent the domain model. A `Q_INVOKABLE` function is also provided in order to load the instruments from a database, for example (here again the function is callable from QML). Finally, the model property can be used by a ListView in order display the current list of instruments (`ListViews` and `DataModels` are covered in Chapter 6). As mentioned previously, you need to register the `Stock`, `Option`, and `CompositeInstrument` classes with the QML type system before being able to use them in QML. The application delegate's constructor is one possible place where you perform this. You also need to add the application delegate to the QML document context (see Listing 3-31).

Listing 3-31. Application Delegate Constructor

```
ApplicationUI::ApplicationUI(bb::cascades::Application *app) :
        QObject(app){

    // Create scene document from main.qml asset, the parent is set
    // to ensure the document gets destroyed properly at shut down.

    qmlRegisterType<Stock>("ludin.instruments", 1, 0, "Stock");
    qmlRegisterType<Option>("ludin.instruments", 1, 0, "OptionType");
    qmlRegisterType<CompositeInstrument>("ludin.instruments", 1, 0, "Composite");

    QmlDocument *qml = QmlDocument::create("asset:///main.qml").parent(this);
    qml->setContextProperty("_app", this);
}
```

And finally, Listing 3-32 shows you how to access the application delegate in your QML document.

Listing 3-32. main.qml

```
Page {
    id: page
    function optionsTotalPrice() {
        var total = 0;
        var options = _app.options;
        for (var i = 0; i < options.length(); i ++) {
            total += options[i].price();
        }
        return total;
    }

    Container {
        Label {
            text: "Options total price: " + page.optionsTotalPrice()
        }
        ListView {
            dataModel: _app.instrumentsModel
        }
    }
    onCreationCompleted: {
        _app.loadInstruments(); // loads intruments from db and popultates data model
    }
}
```

QThread

It is very important not to block the main UI thread when developing Cascades applications.
You should therefore always execute long-running operations in a secondary thread so that the main UI thread stays as responsive as possible. A thread is simply an independent execution flow within your application. In other words, threads can share your application's data but simply run independently

(a thread is also often called a *lightweight process*). In Qt, a thread is managed by an instance of the `QThread` class. This section shows you how to effectively execute a long-running operation using a `QThread` object. As with many things in Qt, it is mostly achieved using signals and slots.

Before starting a new thread, you need to package your workload as a worker object (see Listing 3-33).

Listing 3-33. Worker.h

```
class Worker : public QObject{
Q_OBJECT
public:
    Worker();
    virtual ~Worker();
public slots:
    void doWork();    // do the processing here
signals:
    void finished(double result);
    void error(QString error);

};
```

The worker declares a `Worker::doWork()` that will be called to start the processing and a `finished()` signal that will be emitted once the workload has been completed (in other words, the `finished()` signal will be emitted at the end of `Worker::doWork()`; see Listing 3-34).

Listing 3-34. Worker.cpp

```
Worker::doWork(){
    // do the long processing here
    emit finished(result);
}
```

Assuming that the application delegate is responsible for launching the new thread, it needs to move the `Worker` object to the `QThread` object and start the new thread to perform the workload (see Listing 3-35).

Listing 3-35. ApplicationUI.cpp

```
void ApplicationUI::doWorkAsynch() {
    QThread* thread = new QThread;
    Worker* worker = new Worker;

    worker->moveToThread(thread);
    connect(worker, SIGNAL(error(QString)), this, SLOT(errorString(QString)));
    connect(thread, SIGNAL(started()), worker, SLOT(doWork()));
    connect(worker, SIGNAL(finished(double)), this, SLOT(finished(double)));
    connect(worker, SIGNAL(finished(double)), worker, SLOT(deleteLater()));
    connect(worker, SIGNAL(finished(double)), thread, SLOT(quit()));
    connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
    thread->start();
}
```

As illustrated in the Listing 3-35, the `Worker::doWork()` method is called when the thread's `started()` signal is emitted (the signal is emitted when `QThread::start()` is called). When the worker object has completed the long-running task, it emits the `finished()` signal, which could be used to pass a result back to the application delegate, for example. Note also that the `Worker::finished()` and `QThread::finished()` signals are also used to handle cleanup and make sure dynamically allocated memory is reclaimed (in both cases `QObject::deleteLater()` is used to schedule the objects for deletion).

Summary

Congratulations! By now you know enough to start designing complex applications using QML, JavaScript, Qt, and C++. This chapter has been quite dense, so let's do a quick recap.

C++ is a complex language, but we got to the essentials for building object-oriented programs. In C++, you can override a function in a child class if it has been declared as virtual in the parent class. Having a pure virtual function in a class will effectively make that class abstract. Polymorphism is achieved in C++ through references or pointers to objects. C++ also makes the distinction between value types and references types, which you don't find in languages such as Java, where everything is a reference (except primitives types such int, double, float, boolean, etc.).

By using the MVC pattern, you discovered how to organize your application objects with clearly defined boundaries and responsibilities. This will help you cope with complexity and accommodate change as your application design evolves. The following chapters will build on the foundations presented here and show you how to design beautiful UIs using the Cascades framework. You will master the Cascades core controls, as well as the more advanced ones, integrate with platform services, use the device sensors—and there are many more exciting things to come. From now on, the truly fun topics begin…