APPENDIX A

Credit Derivatives

A credit derivative is a financial contract that aims at reducing credit risk—that is, the risk of default posed by contracts established with a business counterparty. These kinds of derivatives have become increasingly popular in the last decade, because they allow the hedging of complex financial positions even in industries that are not covered by mainstream markets.

As a financial software engineer, you are interested in modeling and analyzing such credit derivative contracts using programming code. Employing some of the methods developed in the previous chapters, it becomes possible to write applications that simplify the pricing and evaluation of such derivative instruments. In particular, credit derivatives can be modeled using some of the same tools that have already been discussed for the analysis of options.

In this chapter, you will learn how to create the C++ code that can be used in the quantitative analysis of credit derivative contracts. Here are some of the topics discussed:

- *General concepts of credit derivatives*: A general exposition of what credit derivatives are and the main types of derivatives commonly used in the marketplace.
- *Modeling the problem*: How to model the problems occurring in the area of credit derivatives. I'll present examples of how such derivatives can be modeled using tools that have been previously used for standard options.
- *Barrier options*: You will learn about barrier options and how they can be used to compute prices for large classes of credit derivatives. You will also see coding examples of how to handle barrier options in C++.
- *Using QuantLib for credit derivatives*: You will find a complete example of how to use the financial classes contained in QuantLib to implement derivatives-related C++ code. I will present the `CDSSolver` class, which implements a pricing strategy for derivatives based on barrier options.

C. Oliveira, *Options and Derivatives Programming in C++20*, https://doi.org/10.1007/978-1-4842-6315-0

Introduction to Credit Derivatives

A credit derivative is a type of financial contract that protects participants from credit risk. Credit risk, in the large majority of the cases, refers to the risk of default (or lack of payment by other means) from a counterpart. For example, consider a company that creates a financial operation that is backed by an insurance contract. If this insurance contract is provided by a third party, this presents a risk of bankruptcy. The participants of this contract want to protect against the possibility of default, so companies can create a credit derivative that will pay a considerable amount of money if that the counterpart goes bankrupt. Such contracts are signed with another third party, which makes the payment if the bankruptcy occurs.

Credit derivatives can be classified according to several categories, which consider how the contract is structured and the participants in such a contract. Here are some of the most common types of credit derivatives that are traded in the market:

- *CDO (collateralized debt obligations)*: A CDO is a type of credit derivative where the obligations paid are collateralized based on some underlying asset. This process of collateralization creates a tiered system, where the several payers are pooled and graded according to their credit risk. Thus, financial companies can sell different tiers, ranging from the highest credit (AAA) to lower level that represent higher default risk (B+ for example).
- *CDS (credit default swap)*: A CDS allows companies to protect themselves against the default of a major market player. The buyer of a CDS makes one or more payments for a predefined period of time. If a default occurs on the covered asset, the CDS buyer is entitled to receive compensation for this credit event.
- *Credit default option*: A credit default option resembles an option contract, but the underlying corresponds to the credit default against which you are seeking protection.
- *CDN (credit linked note)*: A CDN is a financial instrument that allows a particular type of credit risk to be transferred to other investors. Usually these notes are structured as bonds on lower risk assets, which are used to pay creditors if the target institution defaults.

- *CMCDS (constant maturity CDS)*: A CMCDS works just like a CDS, but it has different rules for the amount of the payoff received in the case of a default. With the CMCDS, payoffs change based on considerations that are determined between the participants of the contract. For example, the payoff may be determined according to a particular interest rate index.
- *Total return swap*: This category of derivative is used to transfer financial results between two institutions according to a predefined contract. The buyer makes one or more payments, while it expects to receive the total return of a particular investment as a payoff. This allows some institutions, such as hedge funds, to receive the return of complex financial investment with the help of a second entity that transfers the financial return at the end of the covered period.

Modeling Credit Derivatives

As you saw in the previous section, credit derivatives encompass a large number of financial products that have in common the mitigation of credit risk from one entity to another. This makes it difficult to come up with general models for such a wide class of financial instruments. In this section, you will see a few examples of C++ models applied to a few common classes of credit derivatives.

The first step in creating effective code for credit derivatives is to have a computer model for this type of security. Given the diversity of CD contracts, having a proper model becomes even more important so that other algorithms can be applied to this type of security without the need to understand the internal complexities of each type of credit derivative.

As a first step, you can define a simple class that can be used to store and manipulate the data corresponding to a credit default swap. The fields in this class represent the characteristic values that define a CDS contract. These values are the following:

- *Notional*: This represents the total value of the position encompassed by the contract. The notional is usually larger than the payments due to leverage that is allowed on derivatives contracts.
- *Spread*: The value paid by the buyer of the CDS. It may be paid in a particular schedule, or in a single payment.

- *Time period*: Defines the time period in which the CDS is valid.
- *Pay at default*: A Boolean value that determines if the payoff should be made at the time of credit default.
- *Is long*: A Boolean value that is true if the contract is being bought, and false if the contract is being sold.

In the next few sections, you will see how this information can be used to model CDS contracts with standard techniques employed in quantitative finance. In particular, I will discuss how to analyze such derivatives using the concept of barrier options. You will also see how to calculate the price for such barrier options.

Using Barrier Options

This section I discusses how to use a technique that is frequently employed for the pricing of derivatives in general, including credit derivatives. To simplify the discussion, I use the most basic structure for a financial derivative so that you don't need to worry about complex contractual issues. However, the barrier technique described in this section can be expanded to solve a large class of commonly traded derivatives.

The first step in understanding the solution method is to define a barrier option. A *barrier option* is a special type of derivative where payoff occurs when a particular price level, or *barrier*, is crossed. This makes it different from a normal option, because common options have a payoff that depends on how much the underlying is above or below some threshold. With a barrier option, however, the payoff is paid only as the barrier is crossed.

Barrier options work well as a simple model for credit derivatives, because the credit event is frequently defined as a particular barrier. For example, if the credit event is the bankruptcy of a company, the barrier to be crossed is given by the difference between assets and liabilities in the corporation. When that barrier is breached, the company becomes insolvent and the payoff needs to be made.

There are two main types of barrier options, depending on how the barrier is considered as part of the contract:

- *Knock-in*: This is a barrier option where the payoff is given only when the barrier is touched before expiration.
- *Knock-out*: This is a barrier option where the payoff is given only when the barrier is *not* touched before expiration.

Thus, for example, a barrier option that pays when a company claims bankruptcy is a knock-in option, because the payment happens when the default barrier is reached. You can also classify barrier options according to the current value in relation to the barrier:

- *Down-option*: This is a barrier option where the barrier is below the current value of the underlying asset.
- *Up-option*: This is a barrier option where the barrier is above the current value of the underlying asset.

These two classifications can also be combined, so that you can have down-in options or up-out options. Finally, these options can be calls or puts, depending on whether you are buying the right to sell (put) or the right to buy (call) the underlying instrument.

A Solver Class for Barrier Options

To solve the problem, a new class called `CDSSolver` is defined in this section. This class contains all the elements necessary to define a barrier option, along with the code that solves the pricing problem using functions and classes from the QuantLib repository. The definition of the class contains the member variables needed by the pricing algorithm:

```
class CDSSolver : boost::noncopyable {
public:

   // constructor
   CDSSolver(double val, double sigma, double divYield,
             double forwardIR, double strike, double barrier, double rebate);

   // solve the model
   std::pair<QuantLib::BarrierOption, QuantLib::BlackScholesMertonProcess>
   solve(QuantLib::Date maturity_date);

   // generate a grid
   void generateGrid(QuantLib::BarrierOption &option,
                     QuantLib::BlackScholesMertonProcess &process,
                     const std::vector<QuantLib::Size> &grid);
private:

   double currentValue;
   double sigma;
   double divYield;
   double forwardIR;
   double strike;
   double barrier;
   double rebate;
};
```

The first thing to consider when reviewing this class is that the QuantLib code also uses boost libraries for basic functionality, such as smart pointers. In this case, the `CDSSolver` uses `boost::noncopyable` as a base class, which indicates that the class cannot be copied. Therefore, no copy constructor or assignment operators are declared in `CDSSolver`.

Note Observe that the `CDSSolver` class uses shared pointers declared in boost. This is necessary because QuantLib has boost as a direct dependency, and many of the internal smart pointers are declared in this way. Remember, however, that C++11 also has its own version of `shared_ptr`, which is part of the standard namespace. It is important to avoid confusion between `shared_ptr` declared in boost and in the standard library.

There are two main member functions in the `CDSSolver` class. The solve function is responsible for performing the main tasks associated with the pricing of barrier options. The `generateGrid` evaluates the value of the barrier option at particular time points, as defined by the vector of times points passed as a parameter.

The member variables used by the `CDSSolver` class are the following:

- *currentValue*: Represents the current value of the underlying instrument.
- *sigma*: Represents the variance of the financial instrument.
- *divYield*: The dividend yield paid annually by the underlying.
- *forwardIR*: The forward interest rate, which is used to determine the return of cash that is not invested in the barrier option.

- *strike*: The strike of the barrier option, that is, the price that determines the payoff value.
- *barrier*: The price barrier that needs to be crossed to trigger the payout of the option contract.
- *rebate*: Contractual rebate defined when the barrier option is created.

These variables are later used to solve the pricing problem, as you can see in the following description of the associated code. But first, I will provide a short introduction to the classes included in QuantLib that are used solve this kind of pricing problem.

Barrier Option Classes in QuantLib

QuantLib offers support for pricing credit derivatives and related instruments. In particular, the library contains a set of classes that can be used to price barrier options as defined in the previous section. First, I will review some of these classes, which will later be used in a complete example of how to compute prices for barrier options.

The first class of importance is the `Quote` class. A quote is defined as one or more values that determine the current price of an instrument. The `Quote` class is just the base for several classes that represent quotes for different financial instruments. In this example, I will use a `SimpleQuote` to initialize the quote for the barrier option. The following code shows how this is done:

```
Handle<Quote> quote(boost::shared_ptr<Quote>(new SimpleQuote(currentValue)));
```

This line of code uses a second class that is frequently used in QuantLib: the `Handle` class. A handle is a simple container that allows objects to be referenced and changed when necessary.

The next class used in the implementation of barrier options is `YieldTermStructure`. This class allows you to specify the yield curve currently used by the markets. The yield curve is a representation of the effective interest rates in a particular market, such as, for example, United States Treasury bonds. The curve is formed as you consider the different interest rates for each maturity period, usually measured in years. Figure 15-1 shows an example of the yield term structure for Treasury bonds.

Using the `YieldTermStructure` class in QuantLib, it is possible to store and use this information to compute barrier options. Depending on how the financial instrument is defined, such a yield term structure may be represented by several interest rates, one for each desired time horizon. The `YieldTermStructure` class is abstract and should be instantiated using one of their subclasses, which include:

- `FlatForward`: The simplest cases in which the curve is flat and no variation in interest rate is forecasted.
- `ForwardCurve`: A type of yield curve that can use different rates for each time period. This class can be used for the most common case where the interest rates for different time periods are known.
- `PiecewiseYieldCurve`: A yield curve in which the different segments of the curve are linearized.
- `FittedBondDiscontCurve`: A yield curve where interest rates are given indirectly and the yield can be fitted to represent a set of bonds.

In the example code of the next section, I use the `FlatForward` class as a way to represent a simple short-term yield structure, with no variation in interest rates. More complex yield term structures can be easily accommodated by using one of the previous classes.

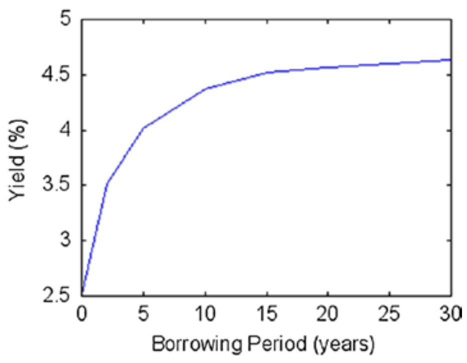

Figure 15-1. *Example of yield structure for a U.S. Treasury bond*

A similar class provided by QuantLib is the `BlackVolTermStructure`. This class represents the volatility term structure, and allows you to determine a particular curve that represents the implied volatility (also known as Black volatility, which is used in the Black-Scholes equation) for the underlying instrument. Similarly to the yield term structure, there are several options for the type of volatility term structure. They differ in the shape of the curve, as well as in the functions that can be used to represent each part of the curve. QuantLib also provides a number of classes that can be used to represent the different types of volatility term structure. Here are some of them:

- `BlackConstantVol`: Used to represent a volatility type that is constant over the whole period.
- `BlackVarianceCurve`: A type of volatility curve where different values of variance are used to determine the volatility.
- `ImpliedVolTermStructure`: A volatility term structure that is defined by the implied volatility associated with a particular instrument.
- `BlackVarianceSurface`: Defines a volatility curve based on a set of data points that define a variance surface. These values are interpolated to generate the desired variance surface.

Using the information stored in these classes, it is possible to describe the Black-Scholes model using the class `BlackScholesMertonProcess`, which is also part of QuantLib. This class receives as parameters the quote, a risk-free yield term structure, and a yield term representing the asset dividend. The class constructor also receives a volatility term structure as a parameter that describes the process.

The class `StrikedTypePayoff` is used to build complex payoffs. It also has a few useful derived classes, including the following:

- `PlainVanillaPayoff`: Represents the most common type of payoff, described by a single value and a strike.
- `PercentageStrikePayoff`: A type of payoff where the strike is given as a percentage of the underlying price, instead of as a fixed value.
- `AssetOrNothingPayoff`: A payoff that is structured as a binary decision. The results are either an asset or nothing.
- `CashOrNothingPayoff`: A payoff that is structured as a binary decision. The results are either cash or nothing.

The example code in the next section uses the `PlainVanillaPayoff` class. The constructor to this class uses as parameters the option type (put or call) and a strike.

BarrierOption is the central class used by QuantLib to model barrier options. This class can be used to calculate the value of a particular barrier option, given a set of parameters that represent that option.

The first parameter to the constructor of BarrierOption is the type of barrier option. As previously described, barrier options can be of four types—UpIn, UpOut, DownIn, and DownOut—depending on the underlying price and the type of barrier used. The next parameters are values that correspond to the barrier, the rebate, the payoff, and the exercise.

Finally, this example also uses a barrier options engine called FdBlackScholesBarrierEngine. This class is used as an implementation for the pricing strategy.

An Example Using QuantLib

Using the classes presented in the previous section, it is possible to explain the implementation of the class CDSSolver. First, consider the first member function, called CDSSolver::solve. This function receives as a parameter a Date object that represents the maturity date of the desired barrier option.

The first step is to create a quote for the option, instantiating the SimpleQuote class and using the current value of the underling as its single argument. Today's date is also computed with the help of the Date::todaysDate member function.

Next, the code tries to instantiate the two term structure objects, one for the dividend yield and another for free cash interest rates. A volatility term structure object is also instantiated using the given volatility, which is estimated using the parameter sigma.

```
// solve the valuation problem using the barrier technique, from today to
the maturity date pair<BarrierOption, BlackScholesMertonProcess>
CDSSolver::solve(Date maturity_date)
{
   Handle<Quote> quote(boost::shared_ptr<Quote>(new
   SimpleQuote(currentValue)));
   Date today = Date::todaysDate();

   shared_ptr<YieldTermStructure> ts1(new FlatForward(today, divYield,
   Thirty360()));
   shared_ptr<YieldTermStructure> ts2(new FlatForward(today, forwardIR,
   Thirty360()));

   shared_ptr<BlackVolTermStructure> vs(new BlackConstantVol(today,
   NullCalendar(),sigma, Thirty360()));
```

The next part of the solve function is responsible for instantiating a process object, which uses QuantLib::BlackScholesMertonProcess. Such a process requires a quote object, yield term structures for interest rates and cash, and a volatility term structure that was previously created.

The function also creates two new objects: a payoff object of type PlainVanillaPayoff that represents the desired call option and a given strike. The exercise is established as a EuropeanExercise type, at the given maturity date.

```
auto process = BlackScholesMertonProcess(quote,
         Handle<YieldTermStructure>(ts1),
         Handle<YieldTermStructure>(ts2),
         Handle<BlackVolTermStructure>(vs));

   shared_ptr<StrikedTypePayoff> payoff(new PlainVanillaPayoff
   (Option::Type::Call, strike));
   shared_ptr<Exercise> exercise(new EuropeanExercise(maturity_date));
```

Finally, you're ready to create a barrier option object, which is an instance of QuantLib::BarrierOption. It takes as parameters the type of barrier, the barrier value, a rebate (if it is available), and the two objects previously created: payoff and exercise.

The next two steps are to create a generalized Black-Scholes object using the existing process and to set the price engine of the barrier option. The price engine algorithm is responsible for price calculation, and this example uses AnalyticBarrierEngine, which is a common algorithm available from QuantLib. The member function CDSSolver::solve will finally return a pair that contains the option and process objects.

```
auto option = BarrierOption(Barrier::Type::UpIn, barrier, rebate, payoff,
exercise);

   auto pproc = shared_ptr<GeneralizedBlackScholesProcess>(&process);
```

```
    option.setPricingEngine(shared_ptr<PricingEngine>(new AnalyticBarrier
    Engine(pproc)));

    return std::make_pair(option, process);
}
```

The next member function implemented in the CDSSolver class is generateGrid. This function is conceptually simple, and it just prints a grid of prices calculated from the given barrier option, using the given BlackScholesMertonProcess and a set of points that determines the option price at a particular date.

Essentially, the function assumes that the grid points are sorted and selects the maximum value. Then, for each element of the grid, a new barrier engine is instantiated and used with the existing barrier option. The price is computed using the resulting combination of option and pricing engines. The code then prints the ratio of increase for that particular point. A backward computation is also performed for comparison purposes.

```
void CDSSolver::generateGrid(BarrierOption &option,
BlackScholesMertonProcess &process, const vector<Size> &grid)
{
    double value = option.NPV();
    Size maxG = grid[grid.size()-1];   // find maximum grid value

    for (auto g : grid)
    {
        FdBlackScholesBarrierEngine be(shared_ptr<GeneralizedBlackScholes
        Process>(&process), maxG, g);
        option.setPricingEngine(shared_ptr<PricingEngine>(&be));

        cout << std::abs(option.NPV()/value -1);

        FdBlackScholesBarrierEngine be1(shared_ptr<GeneralizedBlackScholes
        Process>(&process), g, maxG);
        option.setPricingEngine(shared_ptr<PricingEngine>(&be1));

        cout << std::abs(option.NPV()/value -1);
    }
}
```

Complete Code

This section contains the complete listing of the CDSSolver class. It contains a header file (Listing 15-1), which defines the interface for the class, and an implementation file (Listing 15-2), where the methods solve and generateGrid are implemented.

Listing 15-1. Header File for the CDSSolver Class

```
//
//  CDS.hpp
//  CppOptions

#ifndef CDS_hpp
#define CDS_hpp

#include <stdio.h>

#include <utility>

#include <ql/instruments/barrieroption.hpp>
#include <ql/processes/blackscholesprocess.hpp>

//
// CDSSolver class, incorporates the solution to Credit Default
class CDSSolver : boost::noncopyable {
public:

   // constructor
   CDSSolver(double val, double sigma, double divYield,
             double forwardIR, double strike, double barrier, double rebate);

   // solve the model
   std::pair<QuantLib::BarrierOption, QuantLib::BlackScholesMertonProcess>
   solve(QuantLib::Date maturity_date);

   // generate a grid
   void generateGrid(QuantLib::BarrierOption &option,
                     QuantLib::BlackScholesMertonProcess &process,
                     const std::vector<QuantLib::Size> &grid);
private:

   double currentValue;
   double sigma;
   double divYield;
   double forwardIR;
   double strike;
   double barrier;
   double rebate;
};

#endif /* CDS_hpp */
```

Listing 15-2 shows the implementation file for class `CDSSolver`. It also contains a simple test stub called `test_CDSSolver`, which creates a new instance of `CDSSolver` using a few test parameters.

Listing 15-2. Implementation File for Class CDSSolver

```
//
//  CDS.cpp

#include "CDS.h"

#include <iostream>

//include classes from QuantLib
#include <ql/instruments/creditdefaultswap.hpp>
#include <ql/instruments/barrieroption.hpp>
#include <ql/quotes/SimpleQuote.hpp>
#include <ql/time/daycounters/thirty360.hpp>
#include <ql/exercise.hpp>
#include <ql/termstructures/yield/flatforward.hpp>
#include <ql/termstructures/volatility/equityfx/blackconstantvol.hpp>
#include <ql/processes/blackscholesprocess.hpp>
#include <ql/pricingengines/barrier/analyticbarrierengine.hpp>
#include <ql/pricingengines/barrier/fdblackscholesbarrierengine.hpp>
```

```
using namespace QuantLib;

using std::cout;
using std::vector;
using std::pair;

using boost::shared_ptr;

CDSSolver::CDSSolver(double val, double sigma, double divYield, double
forwardIR, double strike, double barrier, double rebate)
:
   currentValue(val),
   sigma(sigma),
   divYield(divYield),
   forwardIR(forwardIR),
   strike(strike),
   barrier(barrier),
   rebate(rebate)
{
}

// solve the valuation problem using the barrier technique, from today to
the maturity date
pair<BarrierOption, BlackScholesMertonProcess>
CDSSolver::solve(Date maturity_date)
{
   Handle<Quote> quote(boost::shared_ptr<Quote>(new SimpleQuote
   (currentValue)));
   Date today = Date::todaysDate();

   shared_ptr<YieldTermStructure> ts1(new FlatForward(today, divYield,
   Thirty360()));
   shared_ptr<YieldTermStructure> ts2(new FlatForward(today, forwardIR,
   Thirty360()));
   shared_ptr<BlackVolTermStructure> vs(new BlackConstantVol(today,
   NullCalendar(),sigma, Thirty360()));
   auto process = BlackScholesMertonProcess(quote,
         Handle<YieldTermStructure>(ts1),
         Handle<YieldTermStructure>(ts2),
         Handle<BlackVolTermStructure>(vs));

   shared_ptr<StrikedTypePayoff> payoff(new PlainVanillaPayoff(Option::Type
   ::Call, strike));
   shared_ptr<Exercise> exercise(new EuropeanExercise(maturity_date));

   auto option = BarrierOption(Barrier::Type::UpIn, barrier, rebate,
   payoff, exercise);

   auto pproc = shared_ptr<GeneralizedBlackScholesProcess>(&process);

   option.setPricingEngine(shared_ptr<PricingEngine>(new AnalyticBarrier
   Engine(pproc)));

   return std::make_pair(option, process);
}

void CDSSolver::generateGrid(BarrierOption &option,
BlackScholesMertonProcess &process, const vector<Size> &grid)
{
   double value = option.NPV();
   Size maxG = grid[grid.size()-1];   // find maximum grid value

   for (auto g : grid)
   {
      FdBlackScholesBarrierEngine be(shared_ptr<GeneralizedBlackScholes
      Process>(&process), maxG, g);
      option.setPricingEngine(shared_ptr<PricingEngine>(&be));

      cout << std::abs(option.NPV()/value -1);

      FdBlackScholesBarrierEngine be1(shared_ptr<GeneralizedBlackScholes
      Process>(&process), g, maxG);
      option.setPricingEngine(shared_ptr<PricingEngine>(&be1));

      cout << std::abs(option.NPV()/value -1);
   }
}
```

```
void test_CDSSolver()
{
   // use a few test values

   double currentValue = 50.0;
   double sigma = 0.2;
   double divYield = 0.01;
   double forwardIR = 0.05;
   double strike = 104.0;
   double barrier = 85.0;
   double rebate = 0.0;

   CDSSolver solver(currentValue, sigma, divYield, forwardIR, strike,
   barrier, rebate);

   Date date(10, Month::August, 2016);

   auto result = solver.solve(date);

   std::vector<Size> grid = { 5, 10, 25, 50, 100, 1000, 2000 };
   solver.generateGrid(result.first, result.second, grid);
}

int main()
{
   test_CDSSolver();
   return 0;
}
```

Conclusion

Credit derivatives are one of the most common types of derivatives traded in world markets. In this chapter, you learned a little more about such types of derivatives and how they can be modeled using C++.

I initially discussed the concept of credit derivatives and the different types of financial instruments that take part in this category of derivatives. You saw that such derivatives can be used to mitigate credit risks, such as the bankruptcy of a counterparty or the default of a loan, for example.

You also learned about techniques to model such derivatives. In particular, you saw barrier options as a simplified model that can be used to analyze the behavior of such financial instruments.

This chapter presented a complete example of credit derivatives through the use of barrier options using QuantLib classes. The QuantLib repository contains a number of algorithms that are readily available to analyze credit derivatives. In particular, these classes can be used to determine the fair price of certain types of derivatives.

Another task that is frequently necessary when dealing with options and derivatives is the processing of input and output data in common formats. The most popular format for this type of application is based on XML. However, some other formats offer advantages as well. In the next chapter, you will learn about different strategies to process financial data and the common formats used to transfer such information across applications.