Practical .NET for Financial Markets

Yogesh Shetty
Samir Jayaswal

Apress®

Practical .NET for Financial Markets

ISBN-13: 978-1-59059-564-0

ISBN-10: 1-59059-564-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Ewan Buckingham
Technical Reviewer: Ravi Anand
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jason Gilmore, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager and Production Director: Grace Wong
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Lori Bring
Compositor and Production Artist: Kinetic Publishing Services, LLC
Proofreader: April Eddy
Indexer: Valerie Perry
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 2

The Order-Matching Engine

The beauty of time is that it controls the construction and destruction of each instance of the human type.

This chapter discusses an important wing of the trading system—the order-matching decision process. All trading systems must be able to process orders placed by multiple customers from multiple locations, with several orders arriving at the same time and desperately trying to grab the best price in the market at that particular moment. Therefore, it is not surprising to note that trading systems face an influx of data during peak periods. Thus, the systems must easily withstand the heavy traffic of orders and still be able to judiciously find the best price in the market for a given order at the given time.

In this chapter, we explain how an order is matched as soon as it is received and explain the variants that an order exhibits, which in turn affects the matching process. The first part of this chapter discusses the business know-how; then the chapter covers the .NET Framework and exposes the tools that will enable the fulfillment of the discussed business case. Specifically, we explain the different types of collection classes available in the .NET Framework, and we then continue the technical exploration with in-depth coverage of the threading features and the types of synchronization methods that are essential for building a thread-safe system. This is followed by the merging stage where every aspect of business is directly mapped to its low-level technical implementation. Finally, we provide illustrative code samples for the prototype of an order-matching engine written in C#.

Understanding the Business Context of Order Matching

The following sections cover the business context of the order-matching engine.

The Need for Efficient Order Matching

The two primary objectives in the financial marketplace are to keep transaction costs at a minimum and to avoid credit defaults. Although several market practices have been devised to fulfill these objectives, efficient order matching is an important factor for achieving these goals.

A market's liquidity is measured by how easily a trader can acquire (or dispose of) a financial asset and by the cost associated with each transaction. For example, if you wanted to sell a house, you could place an advertisement or go through a real estate agent. Both of these options have costs associated with them. It may also take a month to locate a buyer who is willing to match the price you desire. In this case, the house is considered to be relatively illiquid. But imagine a marketplace where all sellers and buyers of houses in the city came together in one area and tried to find a match—the search would be easier, the chances for finding a buyer would be greater, and the convergence of all buyers and sellers would result in price discovery and hence better prices. In this case, the house is

considered highly liquid. If you extend this example to a marketplace where company instruments (shares and debt instruments) are traded, you get a *stock exchange*, as introduced in Chapter 1. To avoid search costs (and of course to enforce other legal statutes), buyers and sellers come together in a stock exchange on a common platform to transact. Since many buyers and sellers are present at any point in time, searching for a counterpart for an order is relatively easy.

An *order* is an intention to enter into a transaction. Each order has certain characteristics such as type of security, quantity, price, and so on. Initiators of orders notionally announce their willingness to transact with the specified parameters.

Each player in the market wants to get the best possible price. There is a huge scramble to get one's order executed at the right time and at the best available market price. Efficient order matching is thus a highly desirable tenet of an advanced market.

Also, anonymity is considered good for a financial market. This means traders do not know any information about the people with whom they are trading. This is desirable when the participants in the market do not have the same financial strength and when it becomes important for the market to protect the interests of the small players. Anonymity also prevents large players from exerting undue influence on the trade conditions. In such a situation, to protect the integrity of the market, precautions must be taken to ensure that no credit defaults take place. As mentioned in Chapter 1, this is the job of a clearing corporation, which takes away the credit risk concerns of large players through novation. This process also takes care of matching large orders with several potential small players.

Actors: Exchanges and Brokers

Let's examine who the actors are in this matching process. Two counterparties, at a minimum, are required with opposing views to trade with each other—after all, one must be willing to buy when another is willing to sell. And their orders must converge on a common platform, which is the exchange. Generally, exchanges support two forms of trading:

- Oral auctions
- Electronic trading

In an oral auction, traders meet each other face to face on the exchange's trading floor. They shout their bids and offer prices for other traders to hear; the other traders constantly write down these quotes. When two traders agree on a price and an associated quantity, a transaction takes place. Some traders may provide a two-way quotation (a bid price and an asking price) and enter into a transaction only with another trader willing to take the offer or accept the bid.

Oral auctions are the conventional form of trading used in absence of automation, but with the advent of electronic trading, they are on verge of decline. Electronic trading offers the same function through a computer and a trading screen. Traders log their orders through the trading system, and their orders are recorded in the exchange's order book. These orders are then considered for potential matches as designated per the order-matching rules and algorithm defined by exchange.

The most common matching logic uses the concept of priority based on price and time:

- All buy orders received by the exchange are first sorted in descending order of bid price and in ascending order of time for the same prices. This means orders where traders are willing to pay the highest price are kept on the top, reflecting the highest priority. If two orders have the same bid price, the one entered earlier gains a higher priority over the one entered later.
- All sell orders are sorted in ascending order by offer price and in ascending order of time for the same offer prices. This means orders where traders are willing to accept the lowest rate are kept on the top, giving them the highest priority. Two orders asking the same price would be prioritized such that the one entered earlier gets a higher priority.

For example, consider traders A, B, and C who want to buy shares of Microsoft and traders D, E, and F who want to sell shares of Microsoft. Assuming the last traded price of Microsoft (MSFT) shares was $40, consider the scenario shown in Table 2-1.

Table 2-1. *Traders Who Want to Buy/Sell MSFT Orders*

Traders Who Want to Buy	Traders Who Want to Sell
A wants to buy 1,000 @ $40.	D wants to sell 250 @ $40.10.
B wants to buy 500 @ $40.10.	E wants to sell 500 @ $41.20.
C wants to buy 10,000 @ $39.50.	F wants to sell 250 @ $41.50.

A separate *bucket* is assigned for each company's stock, and all orders for the company are grouped into the specific bucket. In business lingua, this bucket is called the *order book*. Thus, the order book for the example in Table 2-1 will look like Table 2-2.

Table 2-2. *MSFT Order Book*

Buy		Sell	
Quantity	**Rate**	**Rate**	**Quantity**
500	$40.10	$40.10	250
1,000	$40	$41.20	500
10,000	$39.50	$41.50	250

All transactions happen on the rate reflected in the topmost row of the order book. This price is popularly called the *touchline price*. The touchline price represents the best ask (lowest sell) price and best bid (highest buy) price of a stock.

In the previous example, because there is a consensus on the rates from both the buyer and the seller, at the touchline price the order will get matched to the extent of 250 shares at $40.10. Thus, the order book will look like Table 2-3.

Table 2-3. *Updated MSFT Order Book After Buy and Sell Order Matched at Touchline Price*

Buy		Sell	
Quantity	**Rate**	**Rate**	**Quantity**
250	$40.10	$41.20	500
1,000	$40	$41.50	250
10,000	$39.50		

Types of Orders

The following are the most common types of orders:

Good till cancelled (GTC) order: A GTC order is an order that remains in the system until the trading member cancels it. It will therefore be able to span several trading days until the time it gets matched. The exchange specifies the maximum number of days a GTC order can remain in the system from time to time.

Good till date (GTD) order: A GTD order allows the trading member to specify the days or a date up to which the order should stay in the system. At the end of this period, the order will automatically get flushed from the system. All calendar days, including the starting day in which the order is placed and holidays, are counted. Once again, the exchange specifies the maximum number of days a GTD order can remain in the system from time to time.

Immediate or cancel (IOC) order: An IOC order allows a trading member to buy or sell a security as soon as the order is released into the market; failing that, the order will be removed from the market. If a partial match is found for the order, the unmatched portion of the order is cancelled immediately.

Price conditions/limit price order: This type of order allows the price to be specified when the order is entered into the system.

Market price order: This type of order allows buying or selling securities at the best price, obtainable at the time of entering the order. The price for such orders is left blank and is filled at the time of the trade with the latest running price in the exchange.

Stop loss (SL) price/order: SL orders allow the trading member to place an order that gets activated when the market price of the relevant security reaches or crosses a threshold price. Until then, the order does not enter the market. A sell order in the SL book gets triggered when the last traded price in the normal market reaches or falls below the trigger price of the order.

Note that for all of these order types, the behavior of an order is determined by a set of special attributes. Every order entered by a buyer or seller follows the same basic principle of trading, but this special attribute further augments the nature of an order by having a direct (or indirect) effect on the profitability of a business. For example, if an order's last traded price was $15 and a limit buy order was placed with a limit price of $15.45 with a stop loss at $15.50, this order would be sent to the market only after the last traded price is $15.50, and it would be placed as a limit price order with the limit price of $15.45.

Order Precedence Rules

The order precedence rules of an oral auction determine who can bid (or offer) and whose bids and offers traders can accept. To arrange trades, markets with order-matching systems use their order precedence rules to separately rank all buy and sell orders in the order of increasing precedence. In other words, they match orders with the highest precedence first.

The order precedence rules are hierarchical. Markets first rank orders using their primary order precedence rules. If two or more orders have the same primary precedence, markets then apply their secondary precedence rules to rank them. They apply these rules one at a time until they rank all orders by precedence.

All order-matching markets use *price priority* as their primary order precedence rule. Under price priority, buy orders that bid the highest prices and sell orders that offer the lowest prices rank the highest on their respective sides. Markets use various secondary precedence rules to rank orders that have the same price. The most commonly used secondary precedence rules rank orders based on their times of submission.

Most exchanges give an option to traders to hide the total quantity of shares they want to transact. This is to discourage other traders from changing their bids/offers in case a large order hits the market. In this case, displayed orders are given higher precedence over undisclosed orders at the same price. Markets give precedence to the displayed orders in order to encourage traders to expose their orders. Though disclosure is encouraged, traders also have the option of not displaying the price in order to protect their interests.

Size (quantity) precedence varies by market. In some markets, small orders have precedence over large ones, and in some markets the opposite is true. Most exchanges allow traders to issue orders with size restrictions. Traders can specify that their entire order must be filled at once, or they can specify a minimum size for partial execution. Orders with quantity restriction usually have lower precedence than unrestricted orders because they are harder to fill.

Order Precedence Ranking Example

Assume that some traders enter the orders shown in Table 2-4 for a particular security.

Table 2-4. *Buy and Sell Orders Sorted Based on Order Arrival Time*

Time (a.m.)	Trader	Buy/Sell	Quantity	Price
10:01	Anthony	Buy	300	$20
10:05	Anu	Sell	300	$20.10
10:08	Nicola	Buy	200	$20
10:09	Jason	Sell	500	$19.80
10:10	Jeff	Sell	400	$20.20
10:15	Nicholas	Buy	500	Market price order
10:18	Kumar	Buy	300	$20.10
10:20	Doe	Sell	600	$20
10:29	Sally	Buy	700	$19.80

The exchange will send an order acknowledgment to the traders' trading terminals and fill the order book as shown in Table 2-5.

Table 2-5. *Order Book (Pre-Match)*

Buy Order Time Stamp	Buy Order Quantity	Buy Price	Buyer	Sell Price	Sell Order Quantity	Seller	Sell Order Time Stamp
10:15	500	Market	Nicholas	$19.80	500	Jason	10:09
10:18	300	$20.10	Kumar	$20	600	Doe	10:20
10:01	300	$20	Anthony	$20.10	300	Anu	10:05
10:08	200	$20	Nicola	$20.20	400	Jeff	10:10
10:29	700	$19.80	Sally				

Note the following in the order book:

- Jason's sell order has the highest precedence on the sell side because it offers the lowest price.
- Nicholas' buy order has the highest precedence on the buy side because it is a market price order.
- Anthony's order and Nicola's order have the same price priority, but Anthony's order has time precedence over Nicola's order because it arrived first.
- In the actual order book, names are not stored and not displayed to traders because the trading system preserves anonymity.

The Matching Procedure

The first step is to rank the orders. Ranking happens on a continuous basis when new orders arrive.

Then the market matches the highest-ranking buy and sell orders to each other. If the buyer is willing to pay as much as the seller demands, the order will be matched, resulting in a trade.

A trade essentially binds the two counterparties to a particular price and quantity of a specific security for which the trade is conducted.

If one order is smaller than the other, the smaller order will fill completely. The market will then match the remainder of the larger order with the next highest-ranking order on the opposite side of the market. If the first two orders are of the same size, both will fill completely. The market will then match the next highest-ranking buy and sell orders. This continues until the market arranges all possible trades.

Order-Matching Example

If the traders in the previous example (see Table 2-5) submit their orders, the market will match the orders as follows:

1. Nicholas's buy order at market price will match Jason's sell order. This will result in the first trade, and both the orders will be removed from the order book.
2. Kumar's order of 300 buy will get matched to Doe's order of 600 sell. Interestingly, this order will get matched at $20.10 even though Doe wanted to sell at $20. Exchange systems are designed to protect the interests of both buyers and sellers. Since there was a passive order from Kumar willing to buy at $20.10 and Doe's order comes in later asking only for $20, she will still get $20.10. Since Kumar's order is completely filled, it will be removed completely from the order book. However, Doe's order of 600 is only half-filled. So, 300 shares of Doe will remain in the order book.
3. In the next step, Anthony's buy order of 300 shares will get fully filled by Doe's balance of 300 at $20, and both orders will be removed from the order book.
4. Now Nicola wants to buy 200 at $20, but Jeff will sell only at $20.20. There is no agreement in price; hence, there will be no further matching, and the matching system will wait either for one of the parties to adjust the price or for a new order at a price where either a buy or a sell can be matched.

Table 2-6 shows the trade book for trades resulting from these orders.

Table 2-6. *Trade Book*

Trade	Buyer	Seller	Quantity	Price
1	Nicholas	Jason	500	$19.80
2	Kumar	Doe	300	$20.10
3	Anthony	Doe	300	$20

Table 2-7 shows the order book after matching.

Table 2-7. *Order Book (Post-Match)*

Buy Order Time Stamp	Buy Order Quantity	Buy Price	Buyer	Sell Price	Sell Order Quantity	Seller	Sell Order Time Stamp
10:08	200	$20	Nicola	$20.20	400	Jeff	10:10
10:29	700	$19.80	Sally				

When the buy side matches with the sell side at an agreed price, the order finds a match, thereby converting it to a trade. Each order can get converted into a single trade or multiple trades, in case the order size is large. Traders whose orders get executed have to make payments when they buy or have to deliver the securities when they sell.

Containment of Credit Risk and the Concept of Novation

Millions of orders get executed everyday, with each trader transacting hundreds and sometimes thousands of trades. In such a process, traders potentially commit to pay others (from whom they bought) and anticipate the receipt of money from others (to whom they sold). Imagine if one of the traders exhausted his payment capacity and defaulted. His default would actually give rise to a chain of defaults, and the integrity of the market as a whole would be in danger. In such a scenario, it would be difficult for large traders to transact with small traders. This, in turn, would raise transaction costs, because traders would start selectively trading with each other. To circumvent this credit risk and bring about confidence in the minds of traders, clearing corporations implement novation.

■ **Note** *Novation* is a Latin word that means *splitting*.

Novation essentially splits every transaction into two parts and replaces one party in the trade with the clearing corporation. So, each party in the transaction feels they have transacted with the clearing corporation.

For example, assume that the orders of buyer A and seller B match for 10,000 shares of Microsoft. In the absence of novation, the trade will look like Figure 2-1.

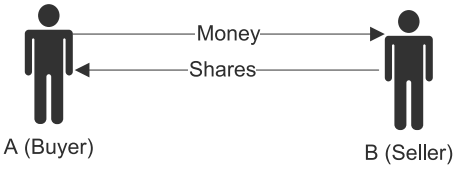

Figure 2-1. *Example of trade without novation*

With novation in place, the trade gets split in two and looks like Figure 2-2.

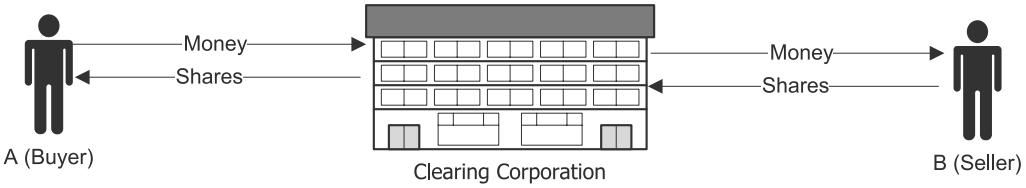

Figure 2-2. *Example of trade with novation*

Buyer A pays money and receives the shares from the clearing corporation. The clearing corporation, in turn, collects the shares and pays the money to seller B.

This brings us to the end of the discussion about the business know-how of order matching. The next section explains the relevant features of .NET that you can use to automate this business case.

Introducing .NET Collections

Data structures provide a container for storing "arbitrary" data and provide a uniform mechanism to operate on this data. Data structures and algorithms share a bloodline relationship—each algorithm is specifically designed and tuned to work with a specific data structure. Therefore, when a specific algorithm is applied on the appropriate data structure, it yields the best possible result. For instance, numerous algorithms can iterate over data stored in a data structure, but often one will work faster than the other when applied in the appropriate environment.

The real litmus test for the performance of an algorithm is to apply it on a huge collection of data elements and then compare the results with that of other similar algorithms. The reason for such an evaluation is that algorithms tend to predict satisfactory results—or at worst only marginal differences when applied on a small amount of data with poor data density. It is only when the number of elements in the data structure increases that the algorithm loses its strength and eventually deteriorates in performance.

The venture of a right algorithm and data structure is the Holy Grail of any good data operation exercise. The scope of a data operation is not only limited to inserting or deleting a data element but, more important, is also limited to seeking data elements. The key to the success of any "data-seeking" activity is directly attributed to the efficiency of the algorithm, denoted by the *number of iterations* it takes to locate an item. This number is derived from the worst-case scenario list; for example, in a linear list of 50 elements where items are inserted in sequential order, it can take at most 50 iterations to locate an item. So, the efficiency of the algorithm is determined by the number of elements stored inside the data structure and is measured based on the following two factors:

- Time (the amount of computation required by the algorithm)
- Space (the amount of memory required by the algorithm)

The efficiency of an algorithm is represented in *Big-O notation*, which acts as a barometer for measuring the efficiency of algorithms. Big-O notation allows a direct comparison of one algorithm over another. The value denoted by Big-O form is sufficient enough to draw a rational comparison between two algorithms without looking at the real code and understanding the real mechanics.

This concludes the brief introductory journey into algorithms; it is time to step back into the .NET world and understand the various types of data structures defined under the `System.Collections` namespace.

Arrays

Arrays have been in existence since the genesis of the computing world. They are a basic necessity of every developer; hence, you will find their implementations molded into all programming languages. Arrays are tightly coupled types, and therefore they are known as *homogenous data structures* (see Figure 2-3). This means once an array of a particular data type is declared, it ensures that the data elements stored must be of the same type.

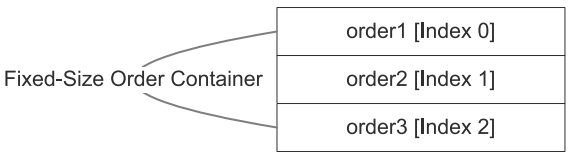

Figure 2-3. *Linear arrangement of a homogeneous order using an array*

The following code example demonstrates how to use an array data structure in .NET:

```
using System;

class ArrayContainer
{
  class Order
  {}
  static void Main(string[] args)
  {
    //Create orders
    Order order1 = new Order();
    Order order2 = new Order();
    Order order3 = new Order();

    //Declare array of order type
    //and add the above three order instance
    Order[] orderList = { order1,order2,order3};

    //Access the order
    Order curOrder = orderList[1] as Order;
  }
}
```

In the previous code, an array of the `Order` type is declared by allocating space for three elements. Then the code assigns a value to an individual element of an array. The primary benefit of using an array is the simplicity it provides in manipulating data elements. An individual data element is accessed by its ordinal position, using an index. An array is extremely efficient when it comes to searching for a data element, even if the number of elements stored in the array is large. Another benefit of using an array is it provides good locality of reference because data elements are arranged in a contiguous block of memory. An array is one of the basic foundations for building sophisticated data structures. These data structures are queues, stacks, and hash tables, and their underlying implementations in .NET are based on arrays.

Array Lists

Array lists inherit the same characteristics of arrays but are specifically designed to address the shortcomings of arrays (see Figure 2-4).

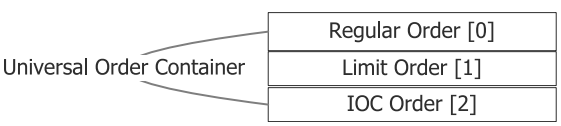

Figure 2-4. *Linear arrangement of a heterogeneous order using an array list*

The foremost problem faced by an array is it is a fixed size—once allocated, an array cannot be resized in a straightforward manner. The array size is defined either during runtime or during compile time. After that, the size remains fixed for the entire duration of the program. The only way to redimension an array is to apply a crude approach—by allocating a separate temporary array (which acts as a temporary storage container for the old data elements), moving the elements from the source array to the temporary array, and then reallocating a different size to the source array, as illustrated in the following code:

```
using System;

class ArrayCopy
{
  class Order{}
  [STAThread]
  static void Main(string[] args)
  {
    //Create an order array
    Order[] orderList = { new Order(),new Order(),
                          new Order(),new Order()};

    //Create a temp array of exactly the same size
    //as original order container
    Order[] tempList = new Order[4];

    //copy the actual items stored in the order array
    //to temp order array
    Array.Copy(orderList,0,tempList,0,4);

    //resize the order array
    orderList = new Order[5];

    //copy the order items from the temp order array
    //to the original order array
    Array.Copy(tempList,0,orderList,0,4);
  }
}
```

Array lists alleviate the fixed-size problem faced by arrays. Behind the scenes, array lists follow the same crude mechanism demonstrated in the previous code, but the mechanism is transparent to developers. Developers, without worrying about dimension issues, can add data element at runtime.

The other major problem faced by an array is the "type coupleness" behavior. An array list solves this problem by acting as a universal container and allows you to insert a data element of any data type. This means an instance of both the value types and the reference type is allowed. However, be careful when dealing with value types because of the implicit boxing and unboxing penalty cost incurred at runtime. The internal storage implementation of an array list is of the reference type, and an instance of the value type that is allocated on the stack cannot be straightforwardly assigned to a reference type. To achieve this task, the instance of the value type needs to be converted to a reference type through a process known as *boxing*. Similarly, in the reverse process known as *unboxing*, the boxed value type that is a reference type is converted to its original value type.

The following code example demonstrates various operations performed on array lists and how different types of orders are added, retrieved, and finally removed:

```
using System;
using System.Collections;

class ArrayListContainer
{
  class Order
  {}
  class LimitOrder
  {}
  class IOCOrder
  {}
  [STAThread]
  static void Main(string[] args)
```

```
  {
    ArrayList orderContainer;
    orderContainer  = new ArrayList();
    //Add regular order
    Order order = new Order();
    orderContainer.Add(order);

    //Add limit order
    LimitOrder limOrder = new LimitOrder();
    orderContainer.Add(limOrder);

    //Add IOC order
    IOCOrder iocOrder =new IOCOrder();
    orderContainer.Add(iocOrder );

    //Access limit order
    limOrder = orderContainer[0] as LimitOrder;

    //Remove limit order
    orderContainer.RemoveAt(0);

    //Display total elements
    Console.WriteLine("Total Elements : " +orderContainer.Count);
  }
}
```

Quick Sort and Binary Search

Both arrays and array lists provide a simple way to insert and delete an item. However, you must also take into account the cost involved in locating a specific data element. The simple technique is to conduct a sequential search where the entire array is enumerated element by element. This approach sounds sensible if you have only a few elements but proves to be inefficient for large numbers of items. Additionally, often you have requirements to sort arrays in either ascending order or descending order. This requirement for both searching and sorting elements illustrates the need for efficient searching and sorting algorithms. Although several well-known, robust sorting and searching algorithms exist, .NET provides out-of-the-box quick sort and binary search algorithms to cater to the sort and search requirements. The quick sort is considered to be one of the most highly efficient sorting algorithms.

The following code demonstrates how elements of arrays are sorted using the quick sort algorithm:

```
using System;

class QuickSort
{
  static void Main(string[] args)
  {
    //elements arranged in unsorted order
    int[] elements = {12,98,95,1,6,4,101};

     //sort element using QuickSort
     Array.Sort(elements,0,elements.Length);

     //display output of sorted elements
     for(int ctr=0;ctr<elements.Length;ctr++)
     {
```

```
      Console.WriteLine(elements[ctr]);
    }
  }
}
```

Similarly, to locate items in an array, the .NET Framework provides a binary search algorithm. The only prerequisite required for this search algorithm is that the array must be sorted. So, the first step is to apply a quick sort to ensure that the arrays are sorted either in ascending order or in descending order. With the incredible increase in the processing power of computers, a search conducted on 1,000 elements using either sequential search or binary search would make no difference to the overall performance of an application. However, the binary search technique would easily outperform the sequential search when the underlying array contains an extremely large number of items.

The following code demonstrates how to use the built-in binary search algorithm to locate a specific item in an array:

```
using System;

class BinarySearch
{
  static void Main(string[] args)
  {
    //elements arranged in unsorted order
    int[] elements = {12,98,95,1,6,4,101};

    //sort element using quick sort
    Array.Sort(elements,0,elements.Length);

    //find element using binary search
    //i.e find 95
    int elementPos = Array.BinarySearch(elements,0,elements.Length,95);

    //if exact match found
    if ( elementPos >= 0 )
    {
      Console.WriteLine("Exact Match Found : " +elementPos);
    }
    else
    //nearest match found
    {
      //bitwise complement operator
      elementPos = ~elementPos;
      Console.WriteLine("Nearest Match : " +elementPos);
    }
  }
}
```

The search is initiated by calling the `Array.BinarySearch` static method. If this method returns a positive value, then it is a success indicator and represents the index of the searched item. However, if the method fails to find the specified value, then it returns a negative integer, and to interpret it correctly, you need to apply a bitwise complement operator. By applying this operator, you get a positive index, which is the index of the first element that is larger than the search value. If the search value is greater than any of the elements in the array, then the index of the last element plus 1 is returned.

The code for the binary search and quick sort demonstrated is based on a single-dimensional fixed array. But in the real world, you will be using an array list to store custom objects such as instruments and order information. Moreover, the binary search and sorting will be based on some specific attributes of custom objects. So, the interesting question is, how do you apply sorting on

specific user-defined attributes of the data element? This is possible with the help of the `IComparer` interface. The role of this interface is to provide a custom hookup that influences the decision made by the quick sort and binary search algorithms.

The following code example shows how orders stored in an order container of the `ArrayList` type are sorted by order price in ascending order and by quantity in descending order:

```
using System;
using System.Collections;

class OrderComparer
{
  public class Order
  {
    public string Instrument;
    public int Qty;
    public int Price;
    public Order(string inst, int price,int qty)
    {
      Instrument= inst;
      Qty= qty;
      Price= price;
    }
  }
  [STAThread]
  static void Main(string[] args)
  {
    //order collection
    ArrayList orderCol = new ArrayList();

    //add five orders
    orderCol.Add(new Order("MSFT",25,100));
    orderCol.Add(new Order("MSFT",25,110));
    orderCol.Add(new Order("MSFT",23,95));
    orderCol.Add(new Order("MSFT",25,105));

    //Invoke the sort function of the ArrayList, and pass the custom
    //order comparer
    orderCol.Sort(new OrderSort());

    //Print the result of the sort
    for ( int ctr = 0;ctr<orderCol.Count;ctr++)
    {
      Order curOrder = (Order)orderCol[ctr];
      Console.WriteLine(curOrder.Instrument+ ":"
                        +curOrder.Price +"-" +curOrder.Qty);
    }
  }

  public class OrderSort : IComparer
  {
    public int Compare(object x, object y)
    {
      Order ox = (Order)x;
      Order oy = (Order)y;
      //Compare the price
      int priceCompare = ox.Price.CompareTo(oy.Price);
```

```
      //Compare the quantity
      int qtyCompare = ox.Qty.CompareTo(oy.Qty);
      if ( priceCompare == 0 )
      {
        //return value multiplied with -1
        //will sort quantity in descending order
        return qtyCompare * -1;
      }

      //returns indication of price comparison value
      return priceCompare;
    }
  }
}
```

In this code, a new instance of `OrderSort` is created that implements `IComparer` and is passed as an argument to the `Sort` method of `ArrayList`. `OrderSort` implements the `Compare` method of `IComparer`. This method compares two values and returns 0 if the first argument is equal to the second argument. Similarly, if the first argument is less than the second argument, then it returns -1; and in case the first argument is greater than the second argument, then it returns 1. The value 0, -1, or 1 determines the sort order position of an element in an array. To sort an array in descending order, you simply multiply this value with -1, which basically reverses the original logical operator.

Queues

In real life, thousands of orders are submitted to the trading system for final processing. These orders originate from different sources, and it is important to process each order based on its arrival time. It is also important to acknowledge these individual orders first and then process them asynchronously. Processing each order synchronously would lead to a higher turn-around time to traders/system users, which is totally unacceptable during peak trading hours. This scenario demands a data structure that can do both of these tasks—storing and retrieving data based on its arrival time. A *queue* is a data structure that meets this condition. It places data at one end called the *entry point* and removes it from the other end called the *exit point*. Because of this characteristic, a queue is called a *first-in, first-out* (FIFO) data structure (see Figure 2-5).

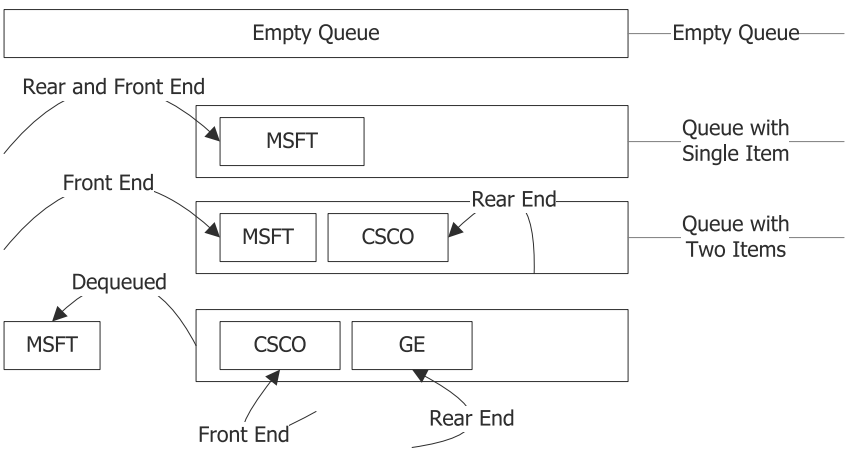

Figure 2-5. *Order processed in a FIFO manner*

The following code demonstrates how orders are processed in a FIFO manner using a queue data structure:

```
using System;
using System.Collections;

class OrderQueue
{
  class Order
  {
    public string Instrument;
    public Order(string inst)
    {
      Instrument = inst;
    }
  }
  static void Main(string[] args)
  {
    //Create Queue collection
    Queue orderQueue = new Queue();

    //Add MSFT order
    orderQueue.Enqueue(new Order("MSFT"));
    //Add CSCO order
    orderQueue.Enqueue(new Order("CSCO"));
    //Add GE order
    orderQueue.Enqueue(new Order("GE"));

    //retrieves MSFT order
    Order dequedOrder = orderQueue.Dequeue() as Order;

    //peek at CSCO order but do not remove from the queue
    Order peekedOrder = orderQueue.Peek() as Order;
  }
}
```

In this code, a queue data structure is constructed by creating an instance of `Queue`. This class provides `Enqueue` and `Dequeue` methods. `Enqueue` adds an order at the rear end of the queue, and `Dequeue` removes the order from the front end of the queue. Oftentimes you may want to peek at the front end of the queue and not remove it; in such cases you can use the `Peek` method, which does not modify the queue and returns the item without removing it. Also, you can use a `Count` property to return the total number of items in `Queue`.

Stacks

Stacks are popularly known as *last-in, first-out* (LIFO) data structures (see Figure 2-6); from a functionality point of view, they do the reverse of queues. In a queue, items are served based on a FIFO basis, whereas in a stack, items are served on a LIFO basis. Stacks push the new item on top of all the other items, and when requesting data, they pop up the topmost item. Modern compilers use a stack data structure extensively during the parsing and compilation process.

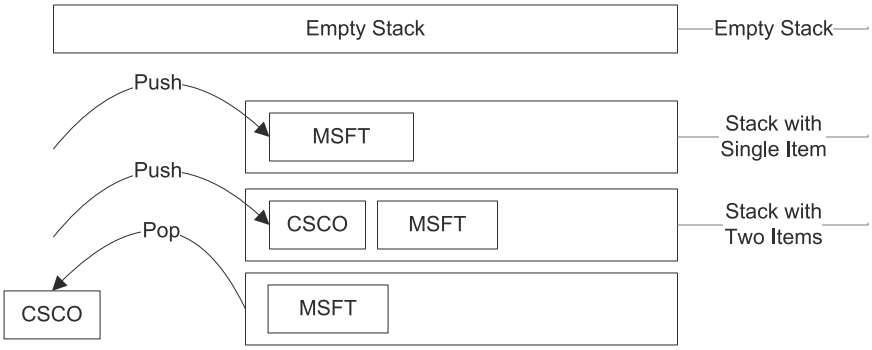

Figure 2-6. *Order processed in a LIFO manner*

The following code demonstrates how orders are processed in a LIFO manner using a stack data structure:

```
using System;
using System.Collections;

class OrderStack
{
  class Order
  {
    public string Instrument;
    public Order(string inst)
    {
      Instrument = inst;
    }
  }

  static void Main(string[] args)
  {
    //create empty stack
    Stack orderStack = new Stack();

    //push MSFT order
    orderStack.Push(new Order("MSFT"));
    //push CSCO order
    orderStack.Push(new Order("CSCO"));
    //pop CSCO order
    Order poppedOrder = orderStack.Pop() as Order;
  }
}
```

In this code, a stack data structure is constructed by creating an instance of `Stack`. This class provides `Push` and `Pop` methods. `Push` places the new order on top of all orders, and `Pop` removes and returns the topmost order. Also, a `Count` property returns the total number of orders stored in `Stack`.

Hash Tables

Consider a scenario where a humongous list of orders is stored in an array list. Rarely would you access an order by its array index; instead, you will be interested in accessing an individual order by its unique order ID. To accomplish this, you will build your own custom search implementation

using IComparer, apply a quick sort on the array list, and finally search the order using the binary search technique. But when the size of the list starts growing at a rapid rate, then this approach sounds inefficient because on every order insert or delete activity the entire array list needs to be re-sorted. This is clearly unacceptable from a performance point of view, and therefore you need a different data structure that conducts efficient searching even during stress conditions. This is where you can use a hash table. A hash table is one of the most commonly used data structures, and its primary goal is to increase search efficiency. The search cost incurred by a hash table to locate an item easily outperforms an array list. Furthermore, this data structure allows you to associate a unique key identifier to an individual data element to form the base for all kinds of activity. So, any subsequent operation (such as the search, update, or delete) of a data element on a hash table is conducted using this unique key identifier.

The following code shows how orders stored in a hash table are tagged by order ID. This key value then forms the basis for all other operations such as searching or deleting a specific order.

```
using System;
using System.Collections;

class HashTbl
{
  //Order Domain class
  public class Order
  {}
  static void Main(string[] args)
  {
    //create empty hash table
    Hashtable orderHash = new Hashtable();

    //add multiple order, order ID is the key
    //and the actual instance of Order is the value
    orderHash.Add("1",new Order());
    orderHash.Add("2",new Order());
    orderHash.Add("3",new Order());

    //locate a specific order using order ID
    Order order = orderHash["1"] as Order;

    //Remove a particular order
    orderHash.Remove("1");

    //check whether order exists with a particular ID
    if ( orderHash.ContainsKey("2") == true )
    {
      Console.WriteLine("This order already exist");
    }
  }
}
```

In this code, the Hashtable class represents a hash table data structure, and orders are added using the Add method. This method has two arguments; the first argument is a unique key identifier, which in this case is the order ID, and the second argument is the actual data element, which is an instance of Order. After inserting a new order, you can retrieve it using its unique order ID. You can also delete orders from Hashtable using the Remove method. The previous code also demonstrates the ContainsKey method, which is used to validate duplicate orders.

Introducing Specialized Collections

The .NET Framework provides other types of collections specifically tuned for performance and storage. These collections are grouped under the `System.Collections.Specialized` namespace.

ListDictionary

`ListDictionary` is primarily used when the total number of elements to be stored is relatively small. The internal storage implementation of any data structures is realized in two ways: a linked list or a vector (array). But in .NET the majority of important data structures such as `Hashtable`, `Queue`, and `Stack` are based on vectors. `ListDictionary` is the only collection in which the underlying storage implementation is based on a linked list. The benefit of using a linked list is that you conserve storage cost by allocating space only when needed. `ListDictionary` is recommended only when the total number of data elements is ten or fewer.

The following code example demonstrates how to use the `ListDictionary` data structure:

```
using System;
using System.Collections.Specialized;

class ListDict
{
  class Order
  {
    public string Instrument;
    public Order(string inst)
    {
      Instrument = inst;
    }
  }
  static void Main(string[] args)
  {
    //create empty list dictionary
    ListDictionary listDict = new ListDictionary();

    //add MSFT order
    listDict.Add("MSFT",new Order("MSFT"));
    //add CSCO order
    listDict.Add("CSCO",new Order("CSCO"));

    //retrieve MSFT order
    Order order = listDict["MSFT"] as Order;
    Console.WriteLine(order.Instrument);
  }
}
```

HybridDictionary

`HybridDictionary`, as the name suggests, provides the characteristics of both `ListDictionary` and `Hashtable`. The internal storage implementation of this collection initially uses `ListDictionary`; however, as soon as the collection starts growing, it switches to `Hashtable`, and subsequent operations are performed on the `Hashtable`. This decision-making process is completely transparent to developers. So, in a nutshell, this collection offers the best of both worlds.