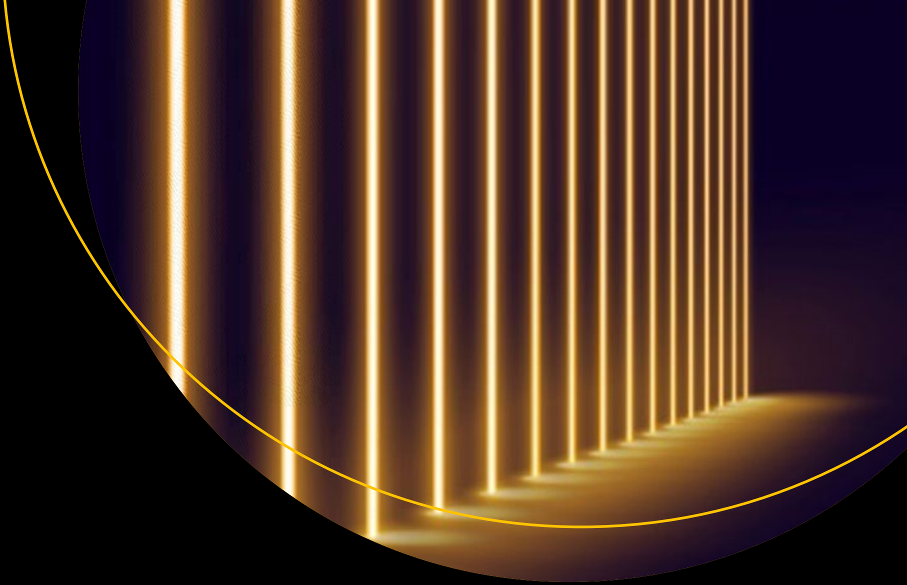

# Essential ASP.NET Web Forms Development

Full Stack Programming with C#, SQL, Ajax, and JavaScript

Robert E. Beasley

Apress®

# Essential ASP.NET Web Forms Development

## Full Stack Programming with C#, SQL, Ajax, and JavaScript

Robert E. Beasley

Apress®

***Essential ASP.NET Web Forms Development: Full Stack Programming with C#, SQL, Ajax, and JavaScript***

Robert E. Beasley
Franklin, IN, USA

ISBN-13 (pbk): 978-1-4842-5783-8 ISBN-13 (electronic): 978-1-4842-5784-5
https://doi.org/10.1007/978-1-4842-5784-5

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 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 www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484257838. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

*To Elizabeth, Zachariah, Isaac, Nathanael, and Elijah*
*I Love You*

# Table of Contents

# About the Author

**Robert E. Beasley** is Professor of Computing at Franklin College in Franklin, Indiana, USA, where he teaches a variety of software engineering courses. He received both his BS and MS degrees from Illinois State University and his PhD from the University of Illinois at Urbana-Champaign. He has been developing software since 1981, has been an active software consultant in both the public and private sectors since 1987, and has been teaching software engineering since 1995. He has authored three books on software engineering, contributed chapters to two books, published over 50 articles in refereed journals and conference proceedings, and delivered numerous speeches and keynote addresses at international conferences.

# Acknowledgments

For any project like this to be successful, input is required from a number of people. I would like to thank David G. Barnette for providing a significant amount of technical feedback on the entire book, Elijah M. Beasley for providing a number of suggestions for improving the flow and continuity of the book, and my other software engineering students for reporting misspellings, typos, and other defects as they were encountered.

# Preface

## Audience

This book was written for anyone interested in learning the ASP.NET Web Forms, C#.NET, SQL, Ajax, and JavaScript Web application development stack, including novice software developers, professional software developers, and college or university students enrolled in a one-semester course or two-semester sequence of courses in Web application development.

## Organization

This book helps you become a pro in one of the most effective and widely used technology stacks for developing highly interactive, professional-grade, database-driven Web applications—ASP.NET Web Forms, C#.NET, SQL, Ajax, and JavaScript. It takes you from beginner to pro in no time. In Part 1, you become familiar with some of the major concepts, methodologies, and technologies associated with .NET Web application development. In this part, you learn about the client-server model, the .NET Framework, the ASP.NET and C# programming languages, and the Visual Studio integrated development environment. In Part 2, you learn how to develop a single-page .NET Web application. In this part, you learn how to create a page and add server and data validation controls to it. The concepts in this part of the book lay the foundation required for learning the C# programming language *in the context of an ASP.NET Web application*. In Part 3, you learn how to program in the C# programming language. In this part, you learn how to perform assignment operations, conversion operations, control operations, string operations, arithmetic operations, date and time operations, array operations, collection operations, and file system operations, as well as create custom C# classes—*in the context of a .NET Web application*. In Part 4, you learn how to develop a multiple-page .NET Web application. In this part, you learn how to maintain state between pages and create master pages, themes, and navigation controls. In Part 5, you learn how to connect a .NET Web application to a SQL Server database. In this part, you learn to read a database schema, program in the SQL programming language, utilize data binding, perform single- and multiple-row database table maintenance, and write code behind database operations. And in Part 6, you learn how to enhance the interactivity of a .NET Web application. In this part, you learn to generate email messages, make use of basic Ajax controls and the Ajax Control Toolkit, and program in the JavaScript programming language.

## Features

### Class Focus

A class diagram is included for every class discussed in the text. Each class diagram articulates some of the most important properties, methods, and events of the class. For those properties, methods, and events that are not included in the class diagram, a link to the official class reference is provided.

### Real-Life Examples

A significant proportion of the examples in the text are drawn from the real-life experiences of the author's own software development practice that began in 1987.

### Clear-Minded, Consistent, and Concise Prose

Every effort has been made to present concepts clearly and logically, utilize consistent language and terminology across all chapters and topics, and articulate concepts fully yet concisely.

### Accessible Language

Although the subject matter of this book is highly technical and specialized, trendy and/or arcane language that is inaccessible to the average learner is either clearly defined or replaced in favor of clear and generalizable terminology.

# PART I

# Overview

CHAPTER 1

# Web Application Development

## 1.1 Introduction

The concept of *hypermedia* (i.e., the combination of hypertext and media) was first envisioned in 1945 by American engineer, inventor, and science administrator Vannevar Bush. However, it wasn't until much later that the technology required to support such a concept was mature enough to make hypermedia something most of us take for granted today.

In 1969, the *Advanced Research Projects Agency Network* (ARPANET) became the first computer network to implement *packet switching* using the *Transmission Control Protocol/Internet Protocol* (TCP/IP) suite—the protocol suite that forms the technical foundation of the Internet today. Packet switching is a method of data transmission that requires three basic steps to get data (e.g., remote computer screens, files, email messages, Web pages) from one computer on a network to another. First, at its origin, the data to be transmitted is separated into a sequenced set of relatively small parts called *packets*. Second, the packets are transmitted independently from their origin to their final destination over routes that have been determined to be optimal for each packet. And third, after all the packets have made their way to their final destination, the data is reassembled from its packets. Early TCP/IP *Application Layer* protocols included *Telnet* for logging in to remote computers, *File Transfer Protocol* (FTP) for transmitting files from one computer to another, and *Simple Mail Transfer Protocol* (SMTP) for sending email messages. These protocols are still in heavy use today.

Although the Internet was alive, well, and growing from the late 1960s through the late 1980s, there was no World Wide Web (a.k.a., Web). However, this was about to change. In 1989, development of the *Hypertext Transfer Protocol* (HTTP) was initiated by English scientist Tim Berners-Lee at the European Organization for Nuclear Research (a.k.a., CERN) in Meyrin, Switzerland—a suburb of Geneva. This protocol was to become the standard for

© Robert E. Beasley 2020
R. E. Beasley, *Essential ASP.NET Web Forms Development*, https://doi.org/10.1007/978-1-4842-5784-5_1

governing the communication between distributed hypermedia systems. With the definition of the first official version of HTTP in 1991, the Web, the hypermedia part of the Internet, was born, and HTTP became another TCP/IP Application Layer protocol like its predecessors Telnet, FTP, and SMTP. Shortly thereafter, Berners-Lee created the very first Web browser. This browser became available to other researchers in January 1991 and was released to the public in August 1991.

Early on, the Web was simply a large collection of *static Web pages*. These pages did little more than display formatted text and visual media (i.e., images, graphics, animations, videos) and permit us to download files and play audio recordings. Today, however, the Web is a massive collection of both static and *dynamic Web pages*. And thanks to programming languages like ASP.NET, dynamic Web pages can do much more than static Web pages can. In addition to the things static Web pages allow us to do, dynamic Web pages allow us to *interact* with the items displayed on a Web page. They also permit us to do things like edit the data on a page, check the data for errors, and save the data to a database.

In this chapter, we will begin by looking at the client-server model, which is a computing approach that distributes processing between servers and clients. Next, we will introduce the .NET Framework. The .NET Framework is Microsoft's Windows-based software development and execution framework. Then, we will discuss ASP.NET and C# programming. ASP.NET is a software development framework that includes all of the classes necessary for building modern, sophisticated Web applications, and C# is a general-purpose programming language for building a variety of application types, including Web applications and Windows applications. After that, we will look at Visual Studio, which is Microsoft's flagship integrated development environment (IDE). This development environment permits us to code and test in several different programming languages via a consistent user interface. And finally, we will learn how to start a new ASP.NET Web Application project.

## 1.2 Client-Server Model

The client-server model is a computing approach that distributes processing between a *server* (i.e., the provider of a resource, service, or application) and its *clients* (i.e., the users of a resource, service, or application). A server is composed of a *server host*, which is a physical computing device connected to a network, and a *server application*, which is a software program that manages multiple, simultaneous client access to the server. Likewise, a client is composed of a *client host*, which is a physical computing device connected to a network, and a *client application*, which is a software program that initiates a *session* with a server so that it can access the server's resources, services, and/or applications. Examples of client-server systems include Web servers and Web clients, email servers and email clients, and FTP servers and FTP clients. Examples of Web server applications include *Internet Information Services* (IIS), *Apache HTTP Server*, and *Oracle iPlanet Web Server*. Examples of Web client applications include *Microsoft Internet Explorer*, *Google Chrome*, and *Mozilla Firefox*. Web client applications are usually called *Web browsers*.

Figure 1-1 shows an example of the client-server model as it applies to a Web application. In the middle of the figure, we see a Web server. As mentioned previously, this server is composed of a server host and a server application that manages client access to the host. Connected to this server via a network (e.g., the Internet) are a number of different clients, including a tablet client, a laptop client, a Mac client, a PC client, and a phone client. The dotted line in the figure indicates that the phone client is connected to the Internet wirelessly. Of course, any server or client can be connected to the Internet wirelessly. Again, each of these clients is composed of a client host and a client application that initiates a session with the server and then accesses the server's resources, services, and/or applications.

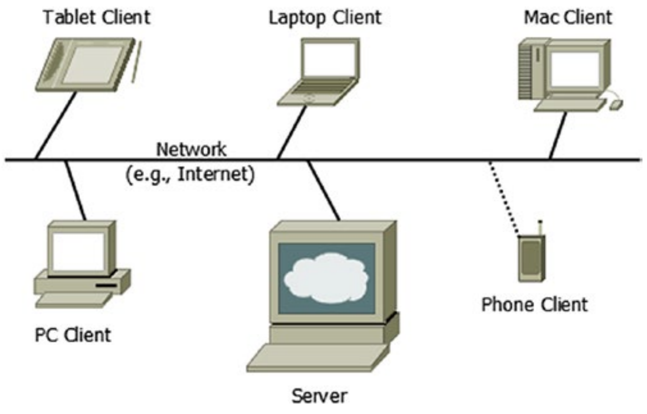

***Figure 1-1.*** *Example of the client-server model as it applies to a Web application*

Recall that Web pages are either static or dynamic. The content and appearance of a static Web page doesn't change each time it is requested. Instead, it always looks the same no matter how many times it is requested or who requests it. It is easy to tell if a Web page is static because it has a file extension of .htm or .html. As we will see in the next figure, this type of Web page only requires the attention of a Web server.

Figure 1-2 shows the processing cycle of a *static* Web page. As can be seen, a Web client (e.g., a laptop computer running Internet Explorer) requests a Web page from a Web server (e.g., a tower computer running IIS) via an *HTTP request*. One important part of this request is the name of the requested Web page (e.g., Display_Products.html). Two other important parts of the request are the *IP addresses* (i.e., the unique Internet addresses) of the server and client. These are necessary so that the HTTP request can make its way to the Web server and so that the requested Web page can make its way back to the requesting Web client. When the Web server receives the HTTP request, it locates the desired Web page file on its hard drive, attaches the file's *Hypertext Markup Language* (HTML) code to an *HTTP response*, and then sends the response to the requesting Web client. When the Web client receives the HTTP response, it uses the attached HTML code to format and display the requested Web page for the end user. If the requested Web page does not exist on the server, the infamous 404 (i.e., Page Not Found) error is passed back to the Web client where it is displayed for the end user.

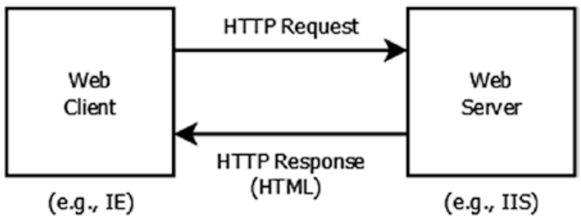

***Figure 1-2.*** *Processing cycle of a static Web page*

Unlike the content and appearance of a static Web page, a dynamic Web page can (and usually does) change each time it is requested. In fact, depending on when it is requested and by whom, it usually contains different information (e.g., different customer information) and can look completely different (e.g., different fields, different images). It is easy to tell if a Web page is dynamic because it has a file extension that is associated with dynamic Web pages. Examples of such file extensions are .aspx (active server page), .php (hypertext preprocessor), and .jsp (java server page). As we will see in the next figure, this type of Web page is processed by both a Web server and an *application server*. When a Web application requires database functionality, a *database server* is required as well.

Figure 1-3 shows the processing cycle of a *dynamic* Web page. As before, a Web client requests a Web page from a Web server via an HTTP request. In this case, however, the request contains the name of a dynamic Web page (e.g., Display_Products.aspx) and the state of any Web page controls (e.g., a name entered into a text box, a check mark placed into a checkbox, a date selected from a calendar). When the Web server receives the HTTP request and sees that the Web page has a file extension of .aspx, it passes processing control to the application server where the *business logic* (e.g., ASP.NET and C# code) of the Web page is executed. If the business logic of the Web page requires the services of a database server (i.e., reading, inserting, updating, or deleting data), the application server passes processing control to the database server (along with any pertinent input parameters) where the database call (usually a Structured Query Language [SQL] call) of the Web page is executed. Once the database call is executed, the response from the database server (e.g., the retrieved data and/or the status of the call) is passed back to the application server where it is processed (e.g., the retrieved data is formatted and/or the status of the call is handled). After this, the application server passes its work back to the Web server, where it locates the desired Web page file on its hard drive, formats the Web page's HTML based on the results of the application server's work, attaches the resulting HTML code to an HTTP response, and then sends the response to the requesting Web client. When the Web client receives the HTTP response, it uses the attached HTML code to format and display the requested Web page for the end user. Again, if the requested Web page does not exist on the server, the infamous 404 (i.e., Page Not Found) error is passed back to the Web client where it is displayed for the end user.

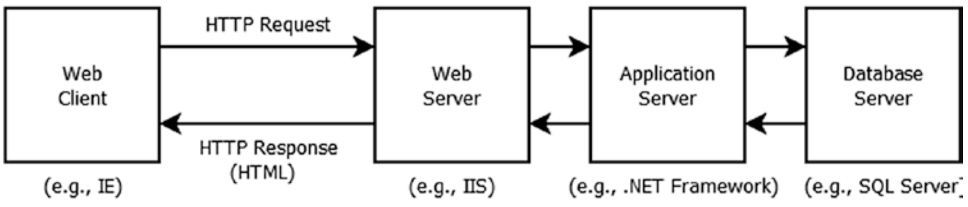

***Figure 1-3.*** *Processing cycle of a dynamic Web page*

Keep in mind that although servers and clients usually run on separate computing devices, they can run on the same device. As an example of the latter, we often use a Web server (e.g., IIS Express), an application server (e.g., .NET Framework), a database server (e.g., SQL Server), and a Web client (e.g., Internet Explorer) all installed on the *same* machine when developing ASP.NET Web applications.

## 1.3 .NET Framework

The .NET Framework is a Windows-based software development and execution framework from Microsoft. This framework consists of two main parts—the *Framework Class Library* (FCL) and the *Common Language Runtime* (CLR).

The Framework Class Library is a large library of *classes*. These classes perform many of the functions needed to develop modern, state-of-the-art software applications, such as Windows applications and Web applications. The classes in the FCL can be utilized by any of the programming languages associated with the .NET Framework (e.g., Visual Basic, Visual C++, Visual C#, Visual F#) and include user interface classes, file access classes, database access classes, and network communication classes. By combining our own custom programming code with the classes in the FCL, we can develop sophisticated software applications relatively efficiently.

The Common Language Runtime is an environment in which all .NET applications execute. These applications do not interact with the operating system directly like some software applications do. Instead, regardless of the programming language used to develop them, .NET applications are compiled into a *Microsoft Intermediate Language* (MSIL) *assembly* and then executed by the CLR. Thus, it is the CLR that interacts with the operating system, which then interacts with the computer's hardware via device drivers. An important aspect of the CLR is the *Common Type System*. The Common Type System defines how all of the value types, reference types, and other types are declared, used, and managed across all of the programming languages of the .NET Framework. Since the CLR provides for its own security, memory management, and exception handling, code running in the CLR is referred to as *managed code*. Figure 1-4 summarizes the organization of the .NET Framework.

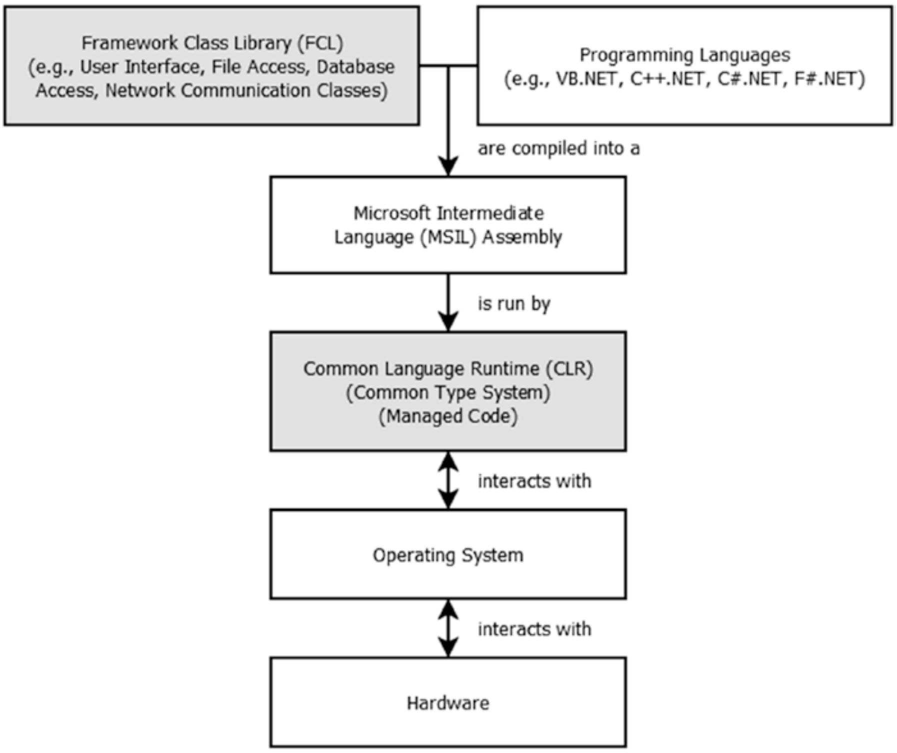
Framework Class Library (FCL) (e.g., User Interface, File Access, Database Access, Network Communication Classes)
Programming Languages (e.g., VB.NET, C++.NET, C#.NET, F#.NET)
are compiled into a
Microsoft Intermediate Language (MSIL) Assembly
is run by
Common Language Runtime (CLR) (Common Type System) (Managed Code)
interacts with
Operating System
interacts with
Hardware

***Figure 1-4.*** *Organization of the .NET Framework*

## 1.4 Object-Orientation Concepts

Object Orientation is a software development paradigm where virtually everything is viewed in terms of *classes* (e.g., customers, Web pages, buttons on a Web page) and *objects* (e.g., a specific customer, a specific Web page, a specific button on a Web page). A class can contain *properties* (i.e., the data of the class) and *methods* (i.e., the functionality of the class) and can handle *events* (i.e., end-user actions or other things that occur in time). The properties, methods, and events of a class are referred to as its *members*. A class *encapsulates* its properties, methods, and events by bundling them together into a single unit and by hiding the details of those internals from other classes. And finally, a class can *inherit* (i.e., take on and utilize) the properties, methods, and events of other classes. We will learn more about these concepts next.

## 1.4.1 Classes and Objects

Classes are like "templates" that represent the characteristics and behaviors of things we encounter in the real world. In our professional lives, we would likely encounter things like customers, employees, products, and orders. On a Web page, we would normally interact with things like buttons, checkboxes, calendars, and text boxes. When developing software applications that involve such things, we typically design and/or utilize classes that model their attributes and actions.

In the .NET Framework, there are two types of classes—*non-static classes* and *static classes*. As a general rule, a non-static class contains non-static properties, non-static methods, and non-static events that we can utilize, but only *after* an object has been instantiated from the class.[1] A static class, on the other hand, contains static properties, static methods, and static events that we can utilize *immediately*, without having to instantiate an object from the class.

When describing a class in this book, we will include a *class diagram*. Table 1-1 shows the general format of a class diagram. Such a diagram will always contain the name of the class and the *namespace* in which it resides. A namespace contains *classes* that provide specific functionality (e.g., page functionality, email functionality, database access functionality) or specialized *types* (e.g., interface types, array types, value types, reference types, enumeration types). A class diagram will also list some *selected* properties, methods, and events of the class. The descriptions of these items will be taken directly from Microsoft's official documentation so that they can be trusted as authoritative. And finally, a class diagram will provide a reference to Microsoft's official documentation of the class. To see all of a class's properties, methods, and events, as well as see code samples of how the class can be used, the interested reader can refer to this documentation.

[1]A *non-static* class can also contain *static* properties, *static* methods, and *static* events that we can utilize *immediately*, without having to instantiate an object from the class.

***Table 1-1.** General format of a class diagram*

| |
|---|
| **Class** |
| **Namespace** |
| **Properties** |
| **Methods** |
| **Events** |
| **Reference** |

There is one more *very* important thing to remember about the class diagrams used in this book. The *event handler methods* used to handle the events of a class will be omitted to conserve space. Event handler methods are those methods that begin with the word "On" and end with an event name. For example, OnInit is an event handler method that is raised by the Init event. If the Init event is already displayed in the Events section of the class diagram, then the OnInit event handler method will be omitted from the Methods section of the class diagram to conserve space.

An object is a single *instance* of a class. For example, say we have an Employee class that serves as the "template" for all employees. In this case, we might have an Employee object that represents Jim J. Jones who has an email address of jjones@mail.com and a password of abc123. We might also have an Employee object that represents Mary M. Morris who has an email address of mmorris@work.com and a password of xyz789. These two distinct objects, both of which are viewed as independent items, were *instantiated* from the Employee class by *constructing* each one and then *setting* their respective Name, EmailAddress, and Password properties. The ability to instantiate multiple objects from a single class is why, for example, we can have several text box and button objects on a single Web page, and each one of them can look and behave similarly yet differently.

## 1.4.2 Properties

Properties represent the data of a class. Properties are *read from* (via a get method) and *written to* (via a set method). For example, the .NET TextBox class has a Text property. If we wish to retrieve the value entered into a text box object, we would need to *get* this property. As another example, the .NET Button class has a BackgroundColor property, a ForegroundColor property, and a Text property. If we wish to display a gray button with red lettering that says "Submit," we would need to *set* these three properties appropriately. Properties can be non-static or static.

## 1.4.3 Methods

Methods perform a function (i.e., a task that returns a value) or a procedure (i.e., a task that *does not* return a value) and are *invoked* or *called*. There are two types of methods—non-static methods and static methods.

A non-static method is a method that can be invoked, but only *after* an object has been instantiated from its associated *non-static* class. For example, if we have an Employee object that has been created from a non-static Employee class, and this class includes a non-static ModifyPassword method, then we can invoke the Employee object's ModifyPassword method to update the employee's password, something like this

```
booSuccess = Employee.ModifyPassword("abc123");
```

where Employee is an object of the non-static Employee class and ModifyPassword is a non-static method of the Employee object.

A static method, on the other hand, is a method that can be invoked *immediately*, without having to instantiate an object from a class. For example, if we have a static Math class that includes a static Sqrt method, and we want to take the square root of 100, then we can invoke the Math class's Sqrt method *directly* (i.e., *without* having to instantiate a Math object from the Math class) to get the square root of 100, something like this

```
bytResult = Math.Sqrt(100);
```

where Math is a static class and Sqrt is a static method of the Math class.

## 1.4.4 Events

Events are things that happen. Events are *raised* by an end-user action or by something else that occurs in time. When an event is raised, and we wish to *handle* that event, we invoke a corresponding method. For example, the .NET Page class raises a Load event every time a Web page loads. If we want to display something for the end user every time the page loads, we would need to handle that event by adding the necessary code to the corresponding OnLoad method. Keep in mind that we need not handle every event that is raised. Events can be non-static or static.

## 1.4.5 Encapsulation

Encapsulation has two meanings in the context of object orientation. First, it refers to the notion that a class's properties (i.e., data) and methods (i.e., the processing that operates on that data) are bundled together and treated as a single unit. Second, it refers to the notion that a class's properties and methods cannot be directly accessed by code that resides outside of the class itself. Thus, in order to get or set a class's properties or execute a class's methods, a class that requires such operations must *request* them from the class that contains the desired properties or methods. This idea is referred to as *information hiding*. Although the concept of information hiding is an important guideline of object orientation, the .NET programming languages permit us to explicitly relax or enforce such access restrictions by declaring properties and methods as private (i.e., they can only be accessed by code within the same class), protected (i.e., they can be accessed by code within the same class and by any related subclasses), or public (i.e., they can be accessed by code in any other class).

One of the benefits of encapsulation is that it shields the internals of a class from other classes so that they can utilize the class's functionality without concern for how the class actually performs its duties. The only thing the other classes need to know about the class is what inputs it requires and what outputs it produces—that is, knowledge of the class's *interface*. In addition, encapsulation facilitates code refactoring (e.g., making a method more efficient or easier to maintain). This is because we can modify the methods of a class without disrupting the class's use by other classes—as long as the modifications do not affect the class's interface. Another benefit of encapsulation is that it encourages us to think through all of a class's properties and methods and to keep them together in one place. This makes coding, testing, and maintenance much easier.

## 1.4.6 Inheritance

Inheritance permits a *child class* (a.k.a., subclass, derived class) to take on and utilize the properties, methods, and events of its *parent class* (a.k.a., superclass)—as well as its parent's parent class and so on. A child class inherits all of the properties, methods, and events of its parent class (with the exception of its *constructor methods* and *destructor methods*), but it also contains properties, methods, and/or events of its own. Thus, a child class always extends the attributes and functionality of its parent class. A parent class that does not inherit any of its properties, methods, or events from another class is referred to as a *base class*. In a class inheritance hierarchy, the relationship that exists between a parent class and its child class is an *is-a-type-of* relationship.

As will become apparent, the main benefits of class inheritance are that *code redundancy* is minimized and *code reuse* is maximized. This is because a child class can use all of the properties, methods, and events of its parent class as if they were its own—we need not write that code again. Keep in mind that inherited properties, methods, and events can be *overridden* by a child class when necessary.

Figure 1-5 shows an example of a class inheritance hierarchy for an employee. In the figure, we can see that the base class in the hierarchy is the Employee class. This class contains the most fundamental properties and methods of the class.[2] The Employee class's child classes (i.e., the Salaried class and the Hourly class) not only inherit *all* of the properties and methods of the Employee class, but they each include additional properties and methods that extend the characteristics and functionality of the Employee class. Looking farther down the hierarchy, we can see that the Salaried class's child classes (i.e., the Administrator class and the Faculty class) not only inherit *all* of the properties and methods of the Salaried class and the Employee class, but they each include additional properties and methods that extend the characteristics and functionality of those classes. Thus, we can see, for example, that a faculty member is a type of salaried employee, who is a type of employee, who works in a department, who has a list of degrees, has a title, gets paid a salary, has a name, has an email address, and has a password. The Faculty class also inherits *all* of the methods of the classes above it, so in addition to a ModifyDepartment method, the Faculty class has a ModifyTitle method, a ModifyName method, and so on.

[2]No events are shown in this example.