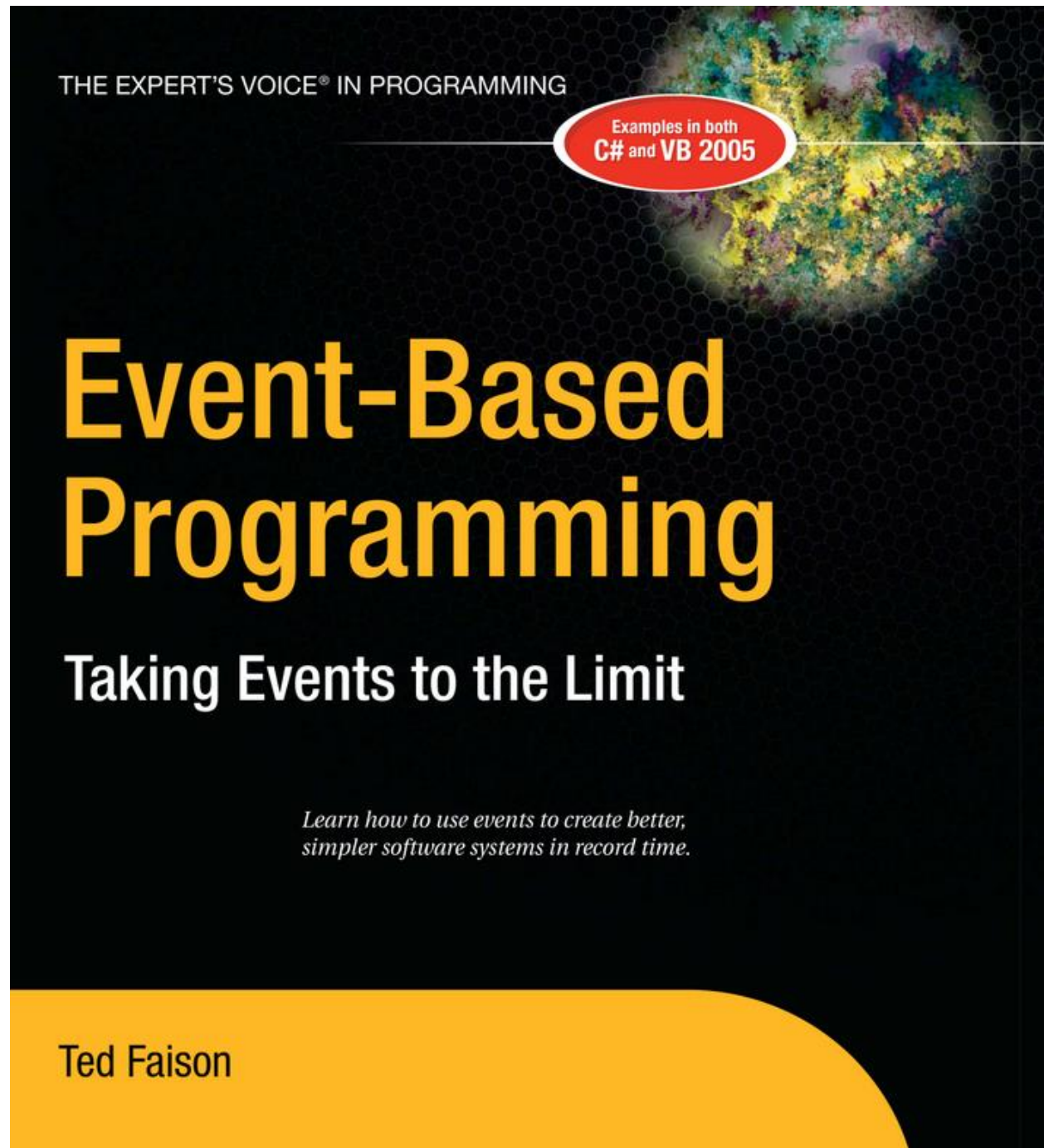


[https://www.google.com/books/edition/Event Based Programming/9CL446IzhuAC?hl=en&gbpv=1&dq=event+notification+definition&pg=PA71&printsec=frontcover](https://www.google.com/books/edition/Event+Based+Programming/9CL446IzhuAC?hl=en&gbpv=1&dq=event+notification+definition&pg=PA71&printsec=frontcover)

Event-Based Programming, Taking Events to the Limit. By [Ted Faison](#) · 2006



CHAPTER 2



Events and Notifications

Talking about events can sometimes be confusing, because people use the word *event* in different ways in different software contexts. For example, graphical user interface (GUI) operating systems use *events* to interact with applications. Applications must be written to conform to an *event-driven* model, doing nothing until an event occurs. When the OS determines that an event of interest to an application has occurred, the OS notifies the program. In this context, the information is sent using procedure calls, with additional parameters telling the application what type of event occurred, such as movement of the mouse or the pressing of a key on the keyboard. The application is free to take any action it wants in response to the event *notification*—including nothing.

Another use of the word *event* is in the context of concurrent programming. Programs set and reset *events*, which are process synchronization primitives such as semaphores and mutual exclusions (mutexes). Events are basically flags that one process can set and another can read.

A popular OO design pattern, called the *subject-observer* pattern, is based on events. The pattern is used when an object, called the *observer*, is interested in knowing about events that occur in another object, called the *subject*. The observer tells the subject which events it is interested in. When the subject detects one of these events, it sends an event notification to the observer, often by calling one of the observer's methods. A subject can theoretically have any number of observers.

Programming languages also use the word *event* to denote actions or entities of various kinds. In .NET components, events are associated with *delegates*, *event variables*, and *handlers*. In Java, events are associated with *event sources*, *event objects*, and *event listeners*. When used alone in a Java context, the word *event* usually indicates the object passed as an argument from an event source to an event listener. You can organize these events into a hierarchy with specialized event types derived from general ones, forming an *event type system*.

Defining Events and Notifications

In ordinary language, an event is an occurrence of some kind, while a notification is a message informing the recipient that something (presumably important) has happened. In the software context, the definitions of events and notifications are inextricably bound, with one being defined in terms of the other. Events are a cause; notifications are an effect.

Definition An event is a detectable condition that can trigger a notification.

Definition A notification is an event-triggered signal sent to a run-time-defined recipient.

Let's look at notifications in more detail. A **notification** is essentially a signal sent when an **event** occurs. The signal can be sent in two basic ways: by transferring data or by transferring execution control.

When the signal transfers data, the signal is a message written to a resource that is shared by the sender and receiver. The shared resource might be a network connection, shared memory, an OS service, a pipe, a computer, or something else. The message can be anything from a single value to a formatted message with fields describing the **event**. The fields might include a sender identifier, a message identifier, a time stamp, a priority value, or other information related to the **event**. In order for the **notification** to have an effect, the receiver must monitor the shared resource and be able to detect any new data written to it.

When the signal transfers execution control, the signal is sent as a procedure call. Depending on the relative locations of the sender and receiver, the procedure call might be local or remote. The signal is the call itself, with optional data passed as parameters or returned values. Figure 2-1 shows the steps involved in the production of a **notification**:

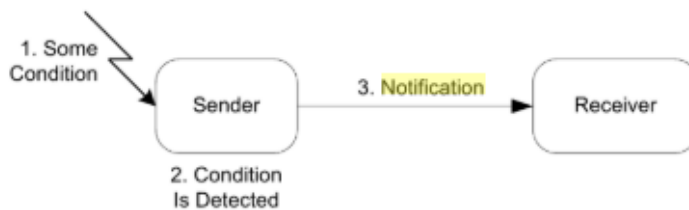
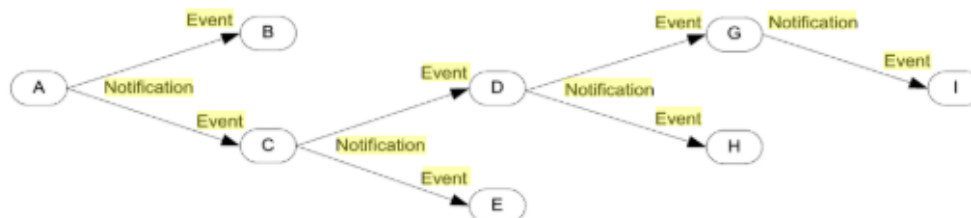


Figure 2-1. Notifications as signals sent from one place to another

The receiver doesn't necessarily have to do anything when it receives a **notification**. What it does is not the concern of the sender. In Figure 2-1, the arrow shows the direction of travel of the signal. Although the arrow *usually* shows the direction of data flow, this isn't always the case. For example, if a **notification** is delivered with a procedure call, the called method might have no input parameters and return a value. In this case, the direction of data flow would be in the opposite direction of the **notification**.

Notifications can be sent pretty much anywhere, limited only by their implementation strategy. Notifications can cross process boundaries, machine boundaries, and even network boundaries. Notifications sent by one component sometimes trigger events in others, producing a sequence of notifications and events that propagate through a system in a chain reaction. One component's notifications are another component's events, as shown in Figure 2-2.



supporting visualization of a system at various levels of abstraction—from the contents of a single component all the way up to the layout of an entire system.

Messaging services, also known as message queues or store-and-forward systems, are used in distributed systems for the reliable delivery of messages, which can be considered in many ways as notifications. Messaging services started receiving widespread interest in the mid-1990s. Messaging services use a central message queuing server²⁰ that acts as a notification server for messages in a distributed system. Several commercial queues are available, including IBM WebSphere MQ (formerly MQSeries), TIBCO Rendezvous, Microsoft Message Queuing (MSMQ), and others. In message queuing systems, client programs send messages to the queuing server, with information regarding the intended recipients and (optionally) the quality of service (QoS) desired. The QoS might specify things such as support for transactions and delivery of return receipts. Messages are persisted locally on the message queue server, typically in a database or in disk files. Delivery is attempted according to a message priority scheme. If the addressee is offline or not available, delivery is attempted repeatedly until a limit is reached, such as a timeout.

The history of events and event-based systems (EBSs) is very rich and includes a large number of people, papers, and products. Due to the limited space available in this book, I have described only the most significant ones. My apologies to the people and projects that I was forced to omit. For another look at the history of event-based systems and current trends, I recommend reading the paper by Eugster.²¹

Nomenclature and Semantics

Before getting too wrapped up in details regarding events, I need to introduce some of the terminology used in EBSs. Distributed systems tend to use different terms and expressions than local (nondistributed) systems, and all the major event-oriented component infrastructures use their own expressions.

Let's start with the terminology used in this book. As mentioned earlier, an event is the detection of a condition, triggering a signal that is sent to interested parties. The entity that is able to detect events is called the *event publisher*, the *event source*, or simply the *sender*. The entity that receives notifications is called the *event subscriber*, the *event handler*, the *notification target*, the *notification receiver*, or simply the *receiver*. The act of sending the notification signal is also called *sending a notification* or *firing the event*. Any data passed with the notification is called the *notification payload*. Before an event can be fired, there must be receivers. The act of establishing a receiver is called *subscription* or *registration*.

Distributed systems use different terms and expressions. As mentioned earlier, EBSs are also called *publish-subscribe* systems. Publishers are components or objects that can detect events and send notifications. Subscribers are components or objects that receive notifications. In order for a subscriber to receive notifications, the subscriber must register by *subscribing* to the publisher. When dealing with publish-subscribe systems that include middleware systems, I'll sometimes use the term *event* to refer to incoming notifications from upstream, because many middleware systems consider incoming signals as events.

An orthogonal nomenclature is also in widespread use in distributed systems that use message servers. These systems often go by the names *message-oriented middleware* (MOM), *store-and-forward systems*, *message-based systems*, and others. Notifications are called *messages*, because they often consist of pure data packets sent over a network connection. Instead of *firing events*, these systems *send messages*. Entities that register to interact with the server are called *clients*. Clients can

C# 3.0 in a Nutshell, A Desktop Quick Reference, By [Joseph Albahari](#), [Ben Albahari](#) · 2009

https://www.google.com/books/edition/C_3_0_in_a_Nutshell/Y0rWd-Q2xkC?hl=en&gbpv=1



3rd Edition
Covers LINQ and
.NET 3.5 CLR and Core Classes

C# 3.0

IN A NUTSHELL

A Desktop Quick Reference



A trailing slash is significant in a URI and makes a difference as to how a server processes a request if a path component is present.

For instance, given the URI `http://www.albahari.com/nutshell/`, you can expect an HTTP web server to look in the *nutshell* subdirectory in the site's web folder and return the default document (usually *index.html*).

Without the trailing slash, the web server will instead look for a file called *nutshell* (without an extension) directly in the site's root folder—which is usually not what you want. If no such file exists, most web servers will assume the user mistyped and will return a 301 *Permanent Redirect* error, suggesting the client retries with the trailing slash. A .NET HTTP client, by default, will respond transparently to a 301 in the same way as a web browser—by retrying with the suggested URI. This means that if you omit a trailing slash when it should have been included, your request will still work—but will suffer an unnecessary extra round trip.

The `Uri` class also provides static helper methods such as `EscapeUriString()`, which converts a string to a valid URL by converting all characters with an ASCII value greater than 127 to hexadecimal representation. The `CheckHostName()` and `CheckSchemeName()` methods accept a string and check whether it is syntactically valid for the given property (although they do not attempt to determine whether a host or URI exists).

Request/Response Architecture

`WebRequest` and `WebResponse` are the common base classes for managing both HTTP and FTP client-side activity, as well as the “file:” protocol. They encapsulate the “request/response” model that these protocols all share: the client makes a request, and then awaits a response from a server.

`WebClient` is simply a façade class that does the work of calling `WebRequest` and `WebResponse`, saving you some coding. `WebClient` gives you a choice of dealing in strings, byte arrays, files, or streams; `WebRequest` and `WebResponse` support just streams. Unfortunately, you cannot rely entirely on `WebClient`; some features (such as cookies) are available only through `WebRequest` and `WebResponse`.

WebClient

Here are the steps in using `WebClient`:

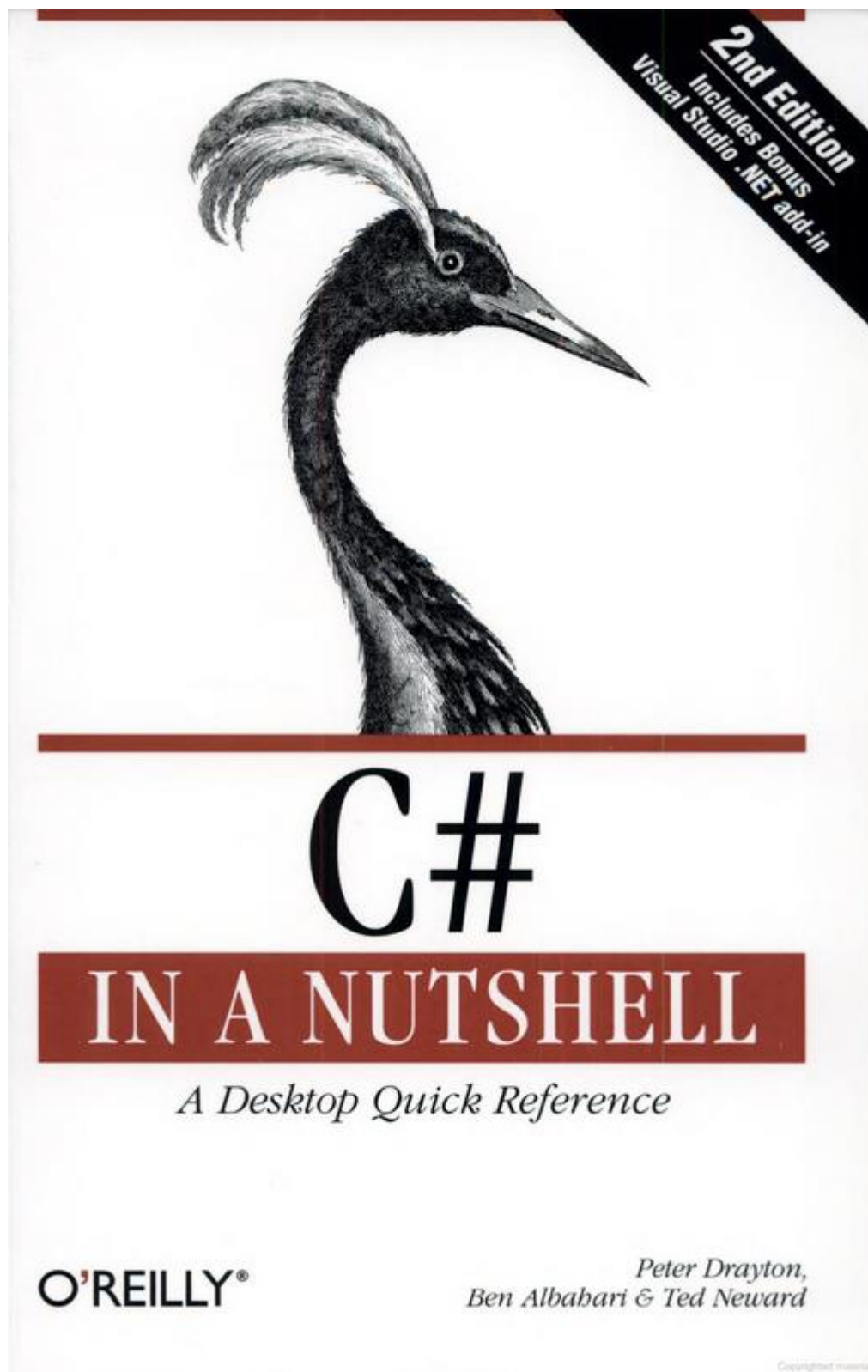
1. Instantiate a `WebClient` object.
2. Assign the `Proxy` property.
3. Assign the `Credentials` property if authentication is required.
4. Call a `DownloadXXX` or `UploadXXX` method with the desired URI.

Its download methods are as follows:

```
public void DownloadFile (string address, string fileName);
public string DownloadString (string address);
public byte[] DownloadData (string address);
public Stream OpenRead (string address);
```

C# in a Nutshell ; By Peter Drayton, Ben Albahari, Ted Neward · 2003

[https://www.google.com/books/edition/C in a Nutshell/bG Aqb6iOUYC?hl=en&gbpv=0](https://www.google.com/books/edition/C%20in%20a%20Nutshell/bGAqb6iOUYC?hl=en&gbpv=0)





9

Networking

The FCL includes a number of types that make accessing networked resources easy. Offering different levels of abstraction, these types allow an application to ignore much of the detail normally required to access networked resources while retaining a high degree of control.

This chapter describes the core networking support in the FCL, and provides numerous examples leveraging the predefined classes. The types mentioned in this section all exist in the `System.Net` and `System.Net.Sockets` namespaces.

Network Programming Models

High-level access is performed using a set of types that implement a generic request/response architecture, which is extensible, to support new protocols. The implementation of this architecture in the FCL also includes HTTP-specific extensions to make interacting with web servers easy.

Should the application require lower-level access to the network, types exist to support the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Finally, in situations in which direct transport-level access is required, there are types that provide raw socket access.

Generic Request/Response Architecture

The request/response architecture is based on Uniform Resource Indicator (URI) and stream I/O, follows the factory design pattern, and makes good use of abstract types and interfaces. A factory method, `WebRequest.Create()`, parses the URI and creates the appropriate protocol handler to fulfill the request.

Protocol handlers share a common abstract base type (`WebRequest`), which exposes properties that configure the request and methods used to retrieve the response.

Responses are also represented as types and share a common abstract base type (`WebResponse`), which exposes a `NetworkStream`, providing a simple streams-based I/O and easy integration into the rest of the FCL.

This example is a simple implementation of the popular Unix `snarf` utility. It demonstrates the use of the `WebRequest` and `WebResponse` classes to retrieve the contents of a URI and print them to the console:

```
// Snarf.cs
// Run Snarf.exe <http-uri> to retrieve a web page
// e.g. snarf.exe http://www.oreilly.com/catalog/csharpnut/
using System;
using System.IO;
using System.Net;
using System.Text;
class Snarf {
    static void Main(string[] args) {

        // Retrieve the data at the URL with a WebRequest instance
        WebRequest req = WebRequest.Create(args[0]);
        WebResponse resp = req.GetResponse();

        // Read in the data, performing ASCII->Unicode encoding
        Stream s = resp.GetResponseStream();
        StreamReader sr = new StreamReader(s, Encoding.ASCII);
        string doc = sr.ReadToEnd();

        Console.WriteLine(doc); // Print result to console
    }
}
```

HTTP-Specific Support

The request/response architecture inherently supports protocol-specific extensions via the use of subtyping. Since `WebRequest.Create()` creates and returns the appropriate handler type based on the URI, accessing protocol-specific features is as easy as downcasting the returned `WebRequest` object to the appropriate protocol-specific handler and accessing the extended functionality.

The FCL includes specific support for the HTTP protocol, including the ability to easily access and control elements of an interactive web session, such as the HTTP headers, user-agent strings, proxy support, user credentials, authentication, keep-alives, pipelining, and more.

This example demonstrates the use of the HTTP-specific request/response classes to control the user-agent string for the request and retrieve the server type:

```
// ProbeSvr.cs
// Run ProbeSvr.exe <servername> to retrieve the server type
using System;
using System.Net;
class ProbeSvr {
    static void Main(string[] args) {
```