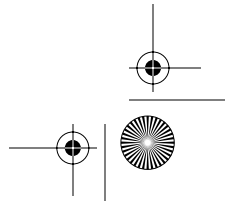
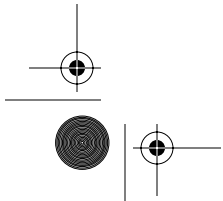


Why SOAP? Web Services and .NET

IN THIS CHAPTER

- A Peek at SOAP
- Web Services and Internet 3.0
- .NET: How Evil Is It?

It appears the Internet is evolving again, and SOAP is playing a major part in that evolution. Before we dive into what SOAP is and why it's useful, we need to step back and look at the larger picture of what's happening on the Internet, and how it's related to Microsoft's latest attempt at world domination (.NET). We need a larger picture in order to put SOAP in the proper context. SOAP itself is simple enough so that its actual structure isn't the most interesting part of the story.





A Peek at SOAP

Briefly, Simple Object Access Protocol (SOAP) is a way to structure data so that any computer program in any language can read SOAP and send messages in SOAP.

If you're like me, you're impatient when it comes to new technology—you want to know all the gritty details and the big picture all at once. To sate your appetite, I'll show you two example SOAP packets here, a request for data and a response to that request, but I won't go into any detail yet. That will have to wait until Chapter 2.

The classic SOAP example involves an application needing to know the latest stock price for a certain company. Your application sends a request for information to a remote computer that has the stock price information. That remote computer hears your request via SOAP, and returns the stock price. This type of interaction is known as request-response, and it's how the Web currently works (you ask a server for a Web page, and the server gives it to you).

Here's all the code for the two messages. Example 1-1 shows the request.

EXAMPLE 1-1 Example SOAP request

```
<env:Envelope
  xmlns:env="http://www.w3.org/2001/06/soap-
  envelope">
  <env:Body>
    <m:getStockPrice
      env:encodingStyle="http://www.w3.org/2001/06/
      soap-encoding" xmlns:m="http://www.wire-
      man.com/services/stockquotes">
      <symbol>PSFT</symbol>
    </m:getStockPrice>
  </env:Body>
</env:Envelope>
```



Example 1–2 shows what the response might look like:

EXAMPLE 1–2 Example of SOAP response

```
<env:Envelope xmlns:env="http://www.w3.org/2001/06/soap-envelope">
  <env:Body>
    <m:getStockPriceResponse
      env:encodingStyle="http://www.w3.org/2001/06/soap-encoding"
      xmlns:m="http://www.wireman.com/services/stockquotes">
      <price>45.89</price>
    </m:getStockPriceResponse>
  </env:Body>
</env:Envelope>
```

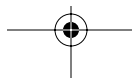
These examples will have to hold you until the next chapter, when we dive headfirst into the details of SOAP. The rest of this chapter covers something more important: SOAP's place in the universe—why it exists and why you should care. Here's a preview: Many people think SOAP is the killer app of XML.

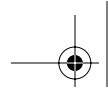
Web Services and Internet 3.0

SOAP exists because of an idea called Web services. Web services are new, and there aren't a whole lot of them out there right now. However, they're growing in number daily and have the same potential for explosion that Web pages had back in 1996 (maybe even greater). To get a strong idea of what Web services are, and why they're part of the next phase of the Internet, it's useful to cover a bit of Internet history to get some context.

A Brief History of the Internet

The Internet began as an American military experiment called ARPANET, whose goal was to build an interconnected computer network that could continue to function even if some nodes (that is, some computers) were destroyed or rendered inoperable by, say, a nuclear strike. Clearly, this experiment worked (without having to test exactly what effect a nuclear





4 Chapter 1 • Why SOAP? Web Services and .NET

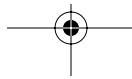
explosion would have), and ARPANET grew until it became known as the Internet. Most serious geeks knew about it and started playing with something called email, even though everyone at the time knew that phones were more convenient. Some folks have called this Internet 1.0.

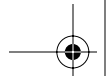
Then in 1991, a guy called Tim Berners-Lee came up with a way for Swiss physicists to post their scientific papers on this Internet so they could read each other's papers easily. This idea caught on quickly and bloomed into the World Wide Web. Initially, the Web was used for people to communicate with each other, sharing such things as scientific papers, personal Web sites, marketing sites for companies, and pornography. In the beginning, the Web was mainly a static environment used mostly for person-to-person communication.

As more serious developers started learning about the Web and a new technology called Common Gateway Interface (CGI), they were able to create programs that allowed people to actually accomplish some things over the Web, such as buying books, reserving airline tickets, and transferring money across bank accounts. In this fashion, a large portion of the Web became devoted to people-machine communication and interaction. People interacted with software programs in order to accomplish things. This phase has been called Internet 2.0.

Many people think the next step in using the Internet will involve software applications communicating with each other over the Internet. These programs will use each other's components, modules, and methods in order to accomplish their tasks. For example, you could write a program in Java that does something, but your program needs to get some information that lives on another machine, a remote computer. Your Java app must be able to communicate with this remote machine in order to get its work done. It must be able to ask that computer for the data it needs, and be able to receive and understand that data. The remote machine must be able to listen for your Java app's cry for help, process the request, and return the information in such a way that your Java program can actually read it.

This next use of the Internet that focuses on machine-machine communication has been called Internet 3.0.





So What's a Web Service?

A Web service is the name for a method or function that is available for other applications to access over the Internet. To use a classic example, say that you have a program that needs to find out the latest stock price for a certain company, and that information lives on another computer. This remote computer has been configured in such a way that if you send it a request for a stock price, it will respond with that company's current price. In this case, the remote computer is hosting a *Web service*. It's just a method or function that can be accessed by other programs over the Web.

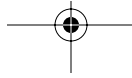
Another way to look at it: When people want some information, they look for certain Web pages and get that information in HTML form. When applications want information, they look for certain Web services and get that information by requesting it from another computer.

People receive Web pages in the form of Hypertext Markup Language (HTML). Computers receive information from Web services in the form of SOAP. SOAP is HTML for computer-computer interaction over the Web.

The goal of Web services is a seamless integration of services across the hardware and software barriers that exist on the Internet. In this system, much of the Internet could become repackaging of other bits and pieces out there.

MONEY AND WEB SERVICES

Why would you bother to write a Web service? The story to get developers such as yourself onto the Web services wagon is that you could write a Web service, expose it to the world, and then charge people who used your Web service, either on a per-use basis or subscription basis. It's certainly an appealing idea: Write something brilliant, even a small thing, and people will flock to you to use your program and drive truckloads of cash to your front door. This has not actually happened yet, but honestly, the potential is there. The next few years will be interesting.





EVOLUTION FROM SOFTWARE TO SERVICES

Right now, the idea of software is that of a discrete process. A consumer buys a shrink-wrapped package with a manual and a CD. They install the CD's program on their machine, and it runs without anyone else on the planet knowing about it. Then the person usually "loans" the CD to someone who asks nicely.

In the Web services view of the world, this process disappears. Instead, you buy access to a Web service for a certain amount of time. For example, instead of buying Word at the store, you'll buy the access rights from Microsoft for, say, a year. You download some components from the Microsoft Web site, but otherwise, your Word program mostly lives on some server at Microsoft. If Word needs to be updated, Microsoft simply upgrades the program on their server, and your program is updated without your even being aware of it. This upgrade is free for you—it's part of the subscription price. After a year, you must renew your subscription.

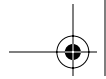
Notice you don't have a CD to loan to anyone else. Part of the push for Web services is that they will cut down substantially on software piracy (at least for a while, anyway). With less piracy, more people will actually have to buy software, and software companies will make more money.

Though this probably rankles some of you, this has aspects of fairness to it—you pay for what you use. Most of us more or less adhere to the philosophy of using something for free if you're using it to learn, but paying for it once you're using the software in a way that makes money for you.

And this is one big reason why Microsoft is putting so much of their weight behind Web services and making it easy for developers (and themselves) to create Web services: In the long run, they think they will make more money.

REQUIREMENTS FOR WEB SERVICES

Initially, a lot of people will be experimenting with creating their own Web services, and many of those services will break from time to time. However, if you want your Web service to be used by anyone who will actually pay money for it, there are some things you must remember. Your Web service:



- Must be 100% reliable. It must always work, and always work in the same way.
- Can also use other Web services, which means that everything must be working or a whole chain of services could go down.
- Must be extremely available—that is, highly scalable and always on.
- Must be able to handle unexpected inputs (usually returning a fault is adequate).
- Must not affect the availability of any other Web service, even if those Web services share common components.
- Must be secured so that only authorized systems and users can use it.

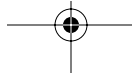
That's a high-level look at what will be required of Web services in order to become used and (hopefully) profitable.

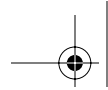
From a more granular point of view, what do Web services need in order to be able to communicate with each other and announce themselves to the world? Here's a list of some other requirements. We'll go into more detail about these later (actually, the whole book is devoted to understanding how the following requirements are implemented, so don't worry if the list below isn't 100% clear; it will be).

- A standard way to represent data, so the services know what to look for.
- A common, extensible message format.
- A common, extensible service description language.
- A way to discover services located on a particular Web site.
- A way to discover service providers.

SOME CONCERNS ABOUT WEB SERVICES

Famed software interface guy Bruce "Tog" Tognazinni recently exposed a potential abuse of Web services. He signed up with a digital TV recording service (ReplayTV), and one of the reasons he signed up was because he could record TV without the commercials. He paid the subscription price and received the service. Partway into the year he signed up for, the company changed its service, and he could no longer record without commercials. Essentially, he paid for a service that was now





significantly less valuable to him, and there was nothing he could do about it.

ReplayTV has since changed their policy and reverted back to their original functionality, but this raises a legitimate concern: How well do you trust the company that is providing a Web service? In an exaggerated example, suppose you're using a Microsoft Web service to use Word, and one day they decide they don't like Word anymore and you find the only thing the program does is run Tetris.

As with many other aspects of Web services, we'll have to see how this issue plays itself out. I can already picture attorneys wringing their hands in happy anticipation of conflict.

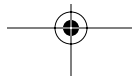
FINDING WEB SERVICES

We'll cover this in more detail later, but much work has been done to define exactly how someone can locate a Web service they need, how they can find out what sort of input parameters that Web service needs, and how to structure those parameters. The current plan is for all Web services to register in a big list called the Universal Discovery, Description and Integration Registry (UDDI Registry). Inside this registry you'll find all the information you need to connect to and interact with a particular Web service. We'll cover this in more detail in Chapter 6, when we cover UDDI.

Once you find the Web service you want to use, your application will also have to know what sort of information to pass to this Web service. A whole new language called, appropriately, Web Services Description Language (WSDL) is being developed that spells out exactly what sort of information a Web service needs in order to work.

WEB SERVICES AND RPC

The request-response form of communication between a local computer and a remote one is often called a Remote Procedure Call (RPC), because it involves a program invoking a method, or function, that lives on a remote computer. RPC is not new—people have been doing it for about 20 years in one form or another. Sun Microsystems is usually given credit for being the





first to create a generic way for applications to call methods on remote machines.

This method of software applications talking to each other over the Internet, machine-to-machine communication, will likely be the next big use of the Web, and may surpass all others in significance. Will it actually happen? Probably—many developers are excited by the idea, and incidentally, Microsoft has bet the company's future that it will.

This kind of system is also known as “distributed.” That is, the system is distributed over several computers. It could be said that part of your application lives in a separate place (even though you can also say that your application is simply calling another application). In any case, this is known as a distributed architecture.

A view of how this works is illustrated in Figure 1–1.

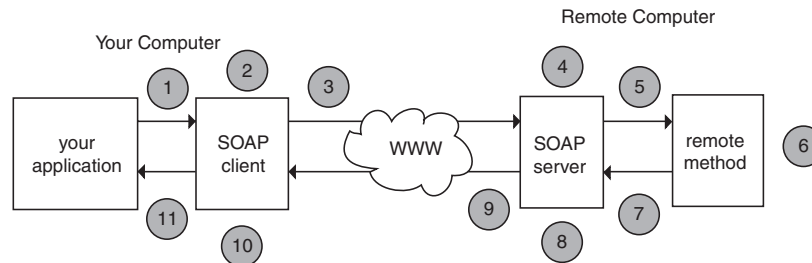
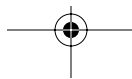
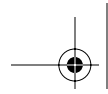


FIGURE 1–1 A Web service using SOAP

In this scenario, your application wants some information that lives on a remote computer somewhere.

1. Your application sends out a request for data, and the first stop on its journey is the SOAP client (which might also live on your computer).
2. This SOAP client takes in the request for data and translates the parameters into a SOAP message, like we saw at the beginning of the chapter.
3. The client then sends this SOAP message on its way to the remote computer, which has a SOAP server running at all times, listening for incoming SOAP messages.

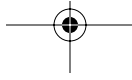
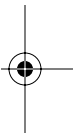


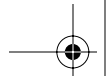
**10 Chapter 1 • Why SOAP? Web Services and .NET**

4. The job of this SOAP server is to listen for SOAP messages, take the input data in those SOAP messages, and translate it into something the remote method can understand.
5. It then passes the input data to the remote method.
6. The method does its thing and comes up with some data.
7. This data is given back to the SOAP server.
8. The SOAP server then acts as a SOAP client, in that it takes the returned data and turns it back into SOAP.
9. It then sends this response back to the original SOAP client on your computer.
10. Your SOAP client then translates the SOAP data into something your application can understand.
11. And finally, your application gets the data it needs in a form it can understand.

A vital facet of Web services is that it doesn't matter what platform the requesting or responding computer is. It could be a Mac calling a method on a Linux box, or a Palm requesting something from a Windows machine. Not only do Web services not care about platforms, they don't even know. This is like the Web: When you download a Web page, do you know what kind of server that it came from? Not without doing a little research. And really, you don't care—you just want your Web page. Web services have a similar goal and so must be platform-independent.

Web services take this concept one step further—they don't even care what language the remote methods are in. C# can talk to Perl, which can talk to Python, which can talk to PHP, which can talk to a really fast guy with an abacus. It doesn't matter. The only thing these programs need to have in common is their ability to read and write SOAP messages. Some of you may remember when people were trying to make a new language called Esperanto, a language that everyone on the planet would learn to speak and read so everyone would be able to talk to each other. It failed miserably. Fortunately, computers are more malleable than people, and we can force computers to all speak the same language, and that language can be SOAP.





TRANSPARENCY AND PLUMBING

A Web service using SOAP is an example of a loosely connected application. Connections are said to be tight or loose. The looser the connection is, the less one application knows about the other. Web services are very loose—methods only know what other methods do, not what platform they're on or even what language they're written in. Loose connections generally go through an intermediate step. In the case of Web services, SOAP is this intermediate step.

The details of the connections between applications are known as plumbing. The tighter a connection, the more you, the programmer, must know about plumbing. Plumbing is usually a pain, and most developers avoid it if possible. That may be one reason why connections have become looser and looser over the years: No one wants to program plumbing.

Looser connections generally provide more flexibility for developers.

SUPPORT FOR WEB SERVICES

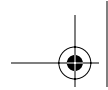
Companies like Microsoft, Oracle, IBM, Hewlett-Packard and Sun are running to create their own Web services. In fact, that's what Microsoft's HailStorm project is: a bunch of Web services.

Part of the Web services plan is for people never to install software on their computers, but instead just access the programs over the Internet. For now, "Web services" just means invoking methods over the Internet: My computer makes your computer run a little program for me and return the results.

You can create Web services right now. You don't need to wait for anyone, including Microsoft. This is important: .NET is not necessary to create a fully functional Web service. Some .NET tools (like Visual Studio.NET) are designed to make it easy to create Web services, but that's it—no one, not you or anyone, has to go through Microsoft to create or use Web services.

In general, it's expected that a number of individual developers will slowly start experimenting and creating their own Web services until some sort of critical mass is reached. At that point, an explosion of Web services is expected to arise as businesses eventually decide that Web services are a solid technology to invest IT capital in.





Why SOAP Is Needed

Now, as developers became more sophisticated in both their skills and needs, they became frustrated with some parts of the Web. It's not because their dot-coms sank beneath the waves along with their dreams of IPOs, but because of the inherent limitations of HTML. Right now, the easiest way to get information from another computer is to request a Web page, and this Web page usually appears in HTML form. It's almost impossible for an application to figure out what the important pieces of data are in an HTML file. After all, HTML is mostly a bunch of formatting tags: bold, italic, table cells, and so on. Here's one way to look at it: HTML isn't information, it's just a picture of information. That's one reason why HTML works so well for people and so poorly for machines. By and large, we see in pictures, but computer programs need consistent, predictable structure in order to make sense of the world. Thus, computers need something other than HTML to talk to each other.

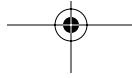
You may remember from earlier in the chapter that we discussed some of the requirements for a Web service. Two of them were:

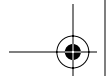
- A standard way to represent data, so the services know what to look for.
- A common, extensible message format.

SOAP provides the answer to these two requirements. More precisely, XML provides a standard way to represent data, and SOAP provides an extensible message format ("extensible" essentially means you can make up your own tags; for example, HTML is not extensible).

.NET: How Evil Is It?

.NET may indeed be evil. But it's also pretty cool. .NET is Microsoft's way of saying, "We think Web services are cool, and we want to make it easy for developers to create Web services." Of course, there's more to .NET than spreading developer-love. Microsoft wouldn't pour this much effort and money into something unless they planned to make a ton of money on it later. Remember earlier in the chapter when we





discussed how software companies could stand to make more money from a subscription model of selling software? I'm going to hazard a guess that's a major reason .NET exists—Microsoft thinks it will make money. I'm not casting judgment, since businesses must make money, but it's useful to understand the motivations behind such things.

So what is .NET, exactly? .NET can be seen as a new runtime environment and a common base class library. If that doesn't make much sense to you, keep reading—there isn't a quick answer. For those of you who already understand compilers and runtimes, you can skim the next section.

Here's what we'll be examining in .NET:

- Microsoft Intermediate Language (IL)
- Common Language Runtime (CLR)
- Common Type System (CTS)
- Common Language Specification (CLS)

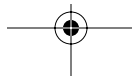
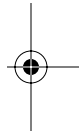
Background: Compilers and Runtimes

When you write a program in, say, Visual Basic, it exists as source code. Source code is what you write in a text editor. No computer in the world can run a program that exists only as source code. You have to translate the source code, which you understand, into a language that the computer can understand. That's what a compiler does: It takes your source code, and squeezes it into something that's almost impossible for people to understand, but computers have no problem with.

Well, almost no problem. Compiling a program from source code is a necessary step, but a computer needs something else that will actually run the compiled program. This something knows how to take the compiled code and force the computer to actually follow the instructions in that code. This piece of software is called a runtime environment.

So, in order to get your source code actually executed, you need both a compiler and a runtime environment.

Usually, each language has its own runtime. The runtime for Java (also known as the Java Virtual Machine) can only force a computer to run a chunk of compiled Java code. If you give a Java compiler some compiled C++ code, it won't know what to do with it.



14 Chapter 1 • Why SOAP? Web Services and .NET

For clarification, it's important to note that Java and C# compile code into bytecode that's understood as native language by a virtual (imaginary) machine. Visual Basic, similarly, compiles into pseudocode that an interpreter executes. C++ compiles into native machine language. In this sense, C++ does not have a runtime environment other than the operating system itself.

Intermediate Language

A huge part of the .NET program is the creation of a common intermediate language. What does this mean? First, an intermediate language is the computer-friendly language your source code is compiled into. When you compile a program using a .NET compiler (like the upcoming Visual Studio.NET), your source code is compiled into a language called Microsoft Intermediate Language (IL). .NET compilers compile all languages into this Intermediate Language, and it tacks on a little metadata as well. Figure 1-2 illustrates this:

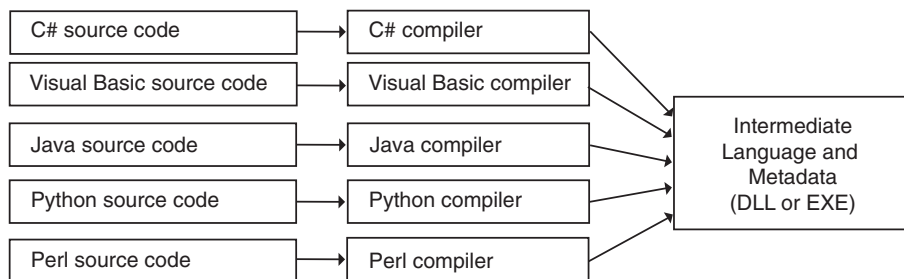


FIGURE 1-2 Compiling source code into a common intermediate language

New languages are being added to this list all the time. At last count, there are 18 languages that can be compiled into IL, and by the time you read this, hopefully a few dozen more.

How do you get a hold of one of these compilers for your language of choice? A number of vendors are creating .NET-aware compilers, so by now, the most common languages have several .NET-aware compilers available.



As you can see from Figure 1-2, the IL files are given the extension DLL or EXE. This is just like a COM (Component Object Model) binary, but internally, the two are nothing alike.

METADATA AND ASSEMBLIES

When some source code is compiled into IL, a .NET-aware compiler does a little bit more than just a translation into IL. It also creates some metadata, or data about the compiled program. This metadata describes in exhaustive detail all of the types of data that are in your code, including base classes, methods, properties, and events. You don't have to worry about creating this metadata yourself—a .NET-aware compiler will do it for you.

When a .NET compiler translates your source code into IL and creates some metadata, this creates something called an *assembly*. We won't be going into more detail here about assemblies.

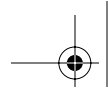
For those of you with experience in Windows programming, you know about a metadata language called IDL, which sometimes was present and sometimes wasn't. This ambiguity isn't part of .NET—you must have metadata. In fact, you can't help it, since the compiler always creates it.

Common Language Runtime

Since all .NET source code is compiled into a single intermediate language, IL, it should only require a single runtime to execute anything in that intermediate language, right? That is, since a runtime can only run a certain kind of compiled code, if you compile different languages into that same compiled code, then a runtime could execute all of them. You would have a single runtime that could execute compiled code from multiple languages. In fact, the runtime wouldn't even care if the compiled code started out as C#, Eiffel, or COBOL. The runtime just sees the IL and runs that. This runtime is called the Common Language Runtime (CLR), and it's a hugely important part of .NET.

The CLR basically has two components: an engine and a base class library.





CLR ENGINE

One component of the CLR is the actual execution engine (also known as `mSCOREE.dll`). This engine examines the metadata of the assembly, makes sure everything is where it should be, and then compiles the IL instructions into something that's platform-specific (for example, into something that's Windows-based or UNIX-based), which is then finally executed.

BASE CLASS LIBRARY

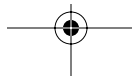
The CLR's base class library essentially contains all the rules for all allowable kinds of data types. In other words, every data type that is in any compiled IL file has to have some equivalent in the base class library. A language can't have any old data type it wants and then expect the CLR to be able to run it. .NET languages have to follow certain rules, and the base class library contains most of those rules. That means that some languages will have to be tweaked slightly in order for them to compile into IL properly. However, this isn't expected to be much of a problem—the base class library has many, many types in it, so whatever language you're working in, chances are you're fine.

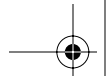
Common Type System (CTS)

The CTS describes all of the data types that the CLR can understand. That is, since the Common Language Runtime isn't all-knowing and can only understand a limited number of data types, the CTS is needed to define exactly what types the CLR understands. In addition, it also defines how those types interact with each other and how they should appear in the metadata.

To get a more thorough understanding of what this actually means, let's look at some broad categories of data types that are part of the CTS. These include:

- **Class types:** The notion of a "class" is necessary for any object-oriented programming, so it's logical that classes are supported by the CTS. A class is composed of any number of methods, properties, and events.
- **Structure types:** Structures are also supported by the CTS.





- **Interface types:** Interfaces are collections of methods, properties, and event descriptions—they're more abstract than classes.
- **Type members:** A member is either a method, a property, a field, or an event.
- **Enumeration types:** An enumeration allows you to create a list of variables that can exist under a single name.
- **Delegate types:** For those of you from the C world, this is the equivalent of a type-safe C style function pointer.
- **Intrinsic data types:** These data types are different forms of integers, strings, Booleans, objects, decimal numbers, and so on.

Common Language Specification

Different computer languages can have different levels of functionality. Some will let you overload operators or will support unsigned data types, while others will not. However, in order for the IL and Common Language Runtime to work, all languages must share at least a certain subset of functionality. The purpose of the Common Language Specification is to define a set of explicit rules that provides a baseline for all languages that wish to be .NET-aware.

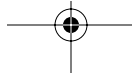
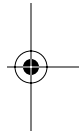
Rules of the CLS include how to represent text strings, how to use static types, how enumerations are represented, and so on.

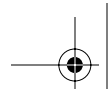
.NET and SOAP

So .NET involves a common intermediate language and a common language runtime (among some other things). What does this have to do with SOAP?

Well, one thing we didn't cover is how to actually create Web services in your .NET-aware language. As it turns out, this is amazingly easy in most languages. In some cases, all you have to do is add a single line of code:

```
[WebMethod]  
your method code
```



**18** Chapter 1 • Why SOAP? Web Services and .NET

When you add this code, you automatically create a method that is capable of acting as a Web service. Well, the compiler and the runtime actually make your method a Web service. Now this Web service can read and write SOAP messages. Microsoft decided to use SOAP as their way of communicating between Web services. For example, if you want your .NET web services to communicate, they must communicate via SOAP messages. Microsoft has decided that SOAP is the glue that will hold Web services together.

NOTE .NET and HailStorm

You may have heard about HailStorm but not completely understood it. HailStorm is a bunch of Web services that Microsoft is developing. These Web services have different functions, like keeping track of calendar events, schedules, contacts, document storage, email and voicemail inboxes, and so on. Microsoft would like you to keep all of your important information with them, and that you would communicate with this information via Web services. The fact you're using a Web service shouldn't be part of the experience, though.

The upshot of the above is that if you only plan to use Microsoft's tools, you don't really need to know anything about SOAP—Microsoft will take care of it for you. While this will save you time, you'll be stuck in case anything goes wrong, because you won't know enough to debug anything. Besides, how much do you trust any software vendor to know what's best for you? Learning SOAP gives you another tool in your programming tool chest, and as most of us know, the more tools we have, the better off we usually are.

Recap

.NET is an aggregation of several things, but its most interesting feature is the ability to write programs in any language. Since they're all compiled into the same intermediate language, these programs can now talk to each other as Web services, and it doesn't matter what language they were written in, where the applications are, or what platform they're on.

