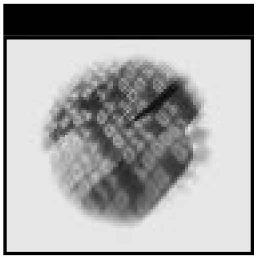
WILEY

NQL

Network Query Language

David Pallmann

Foreword by Harry Forsdick
Vice President, Genuity

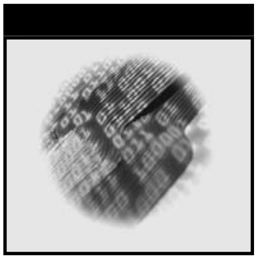


Network Query Language (NQL)

David Pallmann

Wiley Computer Publishing

John Wiley & Sons, Inc.



Network Query Language (NQL)

David Pallmann

Wiley Computer Publishing

John Wiley & Sons, Inc.

To my dear wife, Rebekah

Publisher: Robert Ipsen
Editor: Ben Ryan
Development Editor: Kathryn A. Malm
Managing Editor: Angela Smith
New Media Editor: Brian Snapp
Text Design & Composition: D&G Limited, LLC

This book is printed on acid-free paper. ♾

Published by John Wiley & Sons, Inc., New York
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Pallmann, David.
Network query language (NQL) / David Pallmann.
p. cm.
ISBN: 0-471-20766-7 (acid-free paper)
1. Internet programming. 2. Network Query Language (Computer program language) I. Title.
QA76.625 .P35 2002
005.2'762--dc21
2001006061

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Contents

Foreword

Information is a company's greatest asset. Within the maze of corporate networks and back office systems that asset is typically locked in place, a prisoner of the system that produced it, network that carried it, the client that displayed it. Reusing information presented by an existing system in a new application is tantalizingly easy to talk about but difficult in practice to do. Certainly newer open system architectures make this easier, but there is always some legacy system which is either not open or inaccessible through programmable interfaces. These are the villains of the system integrator's world.

The situation that is so frustrating is that you can see the data that you want to use on the screen—it's right there in front of you, perhaps in three different systems—and you could describe to a six-year-old what steps to take to come up with the desired result. Why can't computers connected to networks do the same?

I first met David Pallmann in 1998 when he came to pitch his ideas about standard ways to access data on network-connected computers. David was spellbinding with his rapid-fire presentation on how the Network Query Language (NQL) could harvest in situ data and repurpose it in new applications. Because combining information from multiple separate systems together to produce larger results has been my passion and profession for many years, David and I hit it off.

David's idea with NQL was to produce a common approach for accessing networked data, much like SQL made access to relational databases uniform. Initially David and his company NQLi focused on building *agents* to access data from World Wide Web pages using NQL. They built a variety of systems for their clients using NQL, including StockVue, a product that uses NQL to harvest and reassemble information about investments.

As they built these applications, David and his team added new capabilities to the language—features that were motivated by the needs of real-world application developers.

When I first encountered NQL, it had primitives supporting Web access, as well as normal control structures, pattern scanning, and output commands for taking harvested information and presenting it in HTML pages. Over the next two years, NQL blossomed into a very complete programming system, adding support for such essential topics as XML, filling out support for networked data facilities such as news, mail, ftp, telnet, and Web site crawling, adding access to non-networked data in local databases and file systems, and finishing off the system with lots of niche, but important, capabilities such as support for genetic algorithms, fuzzy logic, and parallel processing.

Ho, hum, you say. YAPS: Yet Another Programming System. We've all seen similar capabilities in a variety of different systems—perhaps not as well integrated together, but they exist. Wait, the best is yet to come . . . .The NQL Programming Environment.

I don't know about you, but I never get things right the first time I try something. I am constantly trying things out, refining them, trying other approaches. I hate the delay and brittleness of the edit, compile, link, debug cycle. I much prefer the edit, run cycle that an interpreted system allows. Furthermore, although a programmer at heart, I'd much rather tell a computer to watch carefully what I am doing and then make a procedure based on those actions. (For the hard core: I love EMACS keyboard macros).

One of the strongest features of NQL is that it is interpreted. As such, it earns all of the honors that go to such systems. As a developer, you interact with NQL through the NQL Client, a full featured programming environment for creating, modifying, and debugging NQL programs. You can use the editor to write your programs, but also there are a number of wizards which will build NQL program fragments for you.

One such wizard, the browser recorder, allows you to visit a Web site, drill down to the page that contains the information you are interested in and then identify the specific area of that page that contains the information, putting that information in an NQL variable for subsequent processing. With this program-by-example wizard, you identify the place to visit for subsequent information harvesting of that moment's version of the information—the lead story in the living/arts section of a newspaper, the current price of soybeans, or the front page picture on your favorite sports Web site. Further programming in NQL would allow you to build and export an HTML page that combined these items into one consolidated presentation—or deliver an XML representation of these items, plus XML representations of other information retrieved from local SQL databases to another application.

One last observation: By staying at a high level in the types of objects and operations on those objects, NQL allows you to stand on the shoulders of other successful system developers, take information components from those systems, and deliver your audience the value of selective combinations, decisions, and derivations based on those components.

Let me stop now and let you read the book.

Harry Forsdick
VP Advanced Technology
Genuity, Inc.

Introduction

This book describes a new programming language, Network Query Language. Network Query Language is a kind of shorthand for Internet and network programming that lets you easily combine and leverage technologies such as XML and the Web. If you like Stuctured Query Language (SQL) for database programming, you'll love NQL for network programming. If you have a need to create commercial-grade connected applications such as bots, intelligent agents, middleware, and Web applications, you'll find NQL well worth looking at. For investing a small amount of time in getting to know the language, you'll receive a startling savings in development time.

Network Query Language runs on all major computing platforms, and you'll find a copy of it on the CD at the back of this book. You can also download the language from www.nqli.com.

Overview of the Book and Technology

Welcome to a language that breaks some of the rules and isn't afraid to venture into new territory. Where else can you crawl a Web site and download files with a single statement? Or create, train, and run a neural network with a five-line script? Network Query Language equips you with extensive capabilities in four functional areas that are usually lacking in other programming languages: Communications, conversion, distributed processing, and artificial intelligence.

Network Query Language is intended for both experienced developers and beginners. If you're a seasoned developer, you'll find NQL saving you oodles of time. Tasks that used to take you a month can be accomplished in a week, and what used to take a week can be accomplished in a day. For the junior programmer, you'll find NQL very easy to pick up. The language avoids cryptic punctuation, does not assume a computer science background, and uses simple, easy-to-remember keywords. The three hallmarks of NQL are unparalleled ease of use, lightning-fast development, and sheer power. It's not uncommon for developers to tell us they've created solutions and put them into production within 24 hours of their first exposure to NQL.

Network Query Language has interesting roots. My colleagues and I didn't set out to create a new programming language, but we ended up creating one out of necessity. After five years of developing custom solutions—such as intelligent agents, corporate portals, content servers, shopping bots, search engines, and middleware—we had used many different languages and found all of them unsatisfying. A pattern was emerging: No matter what language we used for a project, the building blocks we needed to use on a daily basis were missing. It offended us that programming languages don't contain built-in statements for tasks such as interacting with Web sites, sending and receiving e-mail, working with XML, and applying artificial intelligence techniques. We decided to do something about it. Network Query Language was born when we developed over 500 powerful building blocks and wrapped an SQL-like language around them.

If NQL sounds similar to SQL, that's no accident: SQL is our hero, one of the great success stories of the computer industry. It turned long, proprietary database programs into short, generic queries. There were numerous benefits. Database programs could be written in significantly less time. Short queries were less likely to contain bugs than large database programs, and they could be developed far more rapidly. A more junior level person could perform database programming. Structured Query Language was such an elegant idea that although 25 years have gone by, it is still with us and is used pervasively. We believe today's Internet and network programming situation is remarkably similar to the database problems of the past and cries out for an SQL-like remedy. Our intention with NQL is to appeal to the same paradigm. The things that are true of SQL are also true of NQL: Large programs become short, simple scripts. Extremely rapid development becomes the norm. Junior people can do powerful things.

Best of all, you don't have to choose between NQL and your favorite programming language. Although NQL can be used as a stand-alone solution, it can also be called as a component from another language. If you prefer, you can continue using languages such as Visual Basic, C++, or Java and call NQL when needed to be that missing puzzle piece in a project. Be forewarned, however, that NQL is addictive and you may find yourself using it more and more.

This book's aim is to serve as a handbook: Your field guide to NQL, if you will. It introduces the language, takes you on a guided tour, teaches through the use of examples, and provides reference information.

How This Book Is Organized

The organization of this book is such that it can be read straight through or consulted topically. Readers who are new to the language will find introductory material at the beginning and a smooth progression from chapter to chapter. Each chapter stands on its own, however; readers already familiar with NQL can go directly to an individual chapter for assistance in a particular subject.

Chapter 1, *Presenting NQL*, gives you a first look at Network Query Language without getting into too much of the nuts and bolts. The chapter covers the appropriate uses for NQL, its design philosophies, and the unique features that distinguish it from other languages.

Chapter 2, *Developing in NQL*, guides you through the use of the NQL development environment. The chapter describes how to enter and edit code, run scripts, use the debugger, and deploy solutions.

Chapter 3, *Language Fundamentals*, covers the syntactical rules of NQL, including statements, functions, comments, code blocks, literals, and variables. The chapter also describes the stack and the success/failure condition, two constructs which allow statements to work together efficiently.

Chapter 4, *Flow of Control*, describes how program flow is governed in NQL. The chapter discusses loops, functions, parallel processing, and timing.

Chapter 5, *Expressions, Operators, and Functions*, explains the rules for combining constants, variables, operators, and functions together in expressions. It also documents the built-in functions of NQL and how user-defined functions are specified.

Chapter 6, *Working with Files*, visits the subject of file input/output in NQL. The ability to create, open, append, read, and write files is discussed. File and directory operations are also covered.

Chapter 7, *Transferring Files*, describes how files can be transferred to and from remote systems using File Transfer Protocol (FTP). It also covers file and directory operations.

Chapter 8, *Working with Databases*, discusses how to query and modify databases. Several database protocols are supported, permitting access to the vast majority of databases.

Chapter 9, *Working with XML*, explores the retrieval, parsing, and generation of XML documents. One of NQL's most attractive features is its ability to convert data to and from XML.

Chapter 10, *Accessing Web Sites*, focuses on interfacing with the World Wide Web. Material covered includes accessing Web pages, submitting forms, crawling Web sites, and downloading files. Web services are also covered.

Chapter 11, *Processing HTML*, attacks the problem of how to programmatically mine data from Web pages. Multiple methods of extracting information from HTML documents are covered.

Chapter 12, *Sending and Receiving E-mail*, examines NQL's messaging capabilities. The mechanics of sending e-mail messages, receiving them, and acting upon them are covered.

Chapter 13, *Accessing Newsgroups*, shows how to access newsgroups, also known as forums or discussion boards. Reading newsgroup postings allows your programs to gather information on public perceptions of a subject or who the interested parties are.

Chapter 14, *Searching Directory Servers*, covers how to access directory servers with the LDAP protocol. Searching directory servers allows your programs to connect better with their local environment and locate people, systems, and resources.

Chapter 15, *Terminal Emulation*, enters the world of connecting to UNIX systems, minicomputers, and legacy systems. Your programs can mimic terminal-based interactive users, sending commands, waiting for responses, and capturing screens of output.

Chapter 16, *Interacting with Applications*, is about cooperation between your programs and other software. OLE Automation permits you to exchange information with other programs or control them. Exporting allows your scripts to create documents, spreadsheets, and presentations.

Chapter 17, *Socket Communications*, describes TCP/IP socket programming in NQL, which you would use to implement a protocol NQL does not already support, or your own custom protocol.

Chapter 18, *Serial Communications*, discusses how to perform serial port communications in NQL, which you would use to read and write data to and from a local device connected to your computer.

Chapter 19, *Synchronization*, is concerned with creating distributed applications, where multiple agents must cooperate to work as a team. Methods of communications and synchronization are covered.

Chapter 20, *E-Commerce*, explores electronic transactions, which NQL supports for several major credit card providers. Open Financial Exchange is also covered, which makes automated interaction with financial institutions possible.

Chapter 21, *Graphics*, explains how to work with graphic images in NQL. Images can be dynamically created, modified, saved, and converted to other formats. The chapter also covers reading images from digital devices.

Chapter 22, *Agent Characters*, discusses how to create animated agent characters that speak, move, and perform actions.

Chapter 23, *Fuzzy Logic*, describes the use of fuzzy logic, a form of soft computing. Fuzzy logic allows your programs to represent real-world quantities more accurately and make better decisions.

Chapter 24, *Neural Networks*, covers how to create, train, and run neural networks. Neural networks are powerful due to their ability to learn, and are useful for applications that require pattern matching or prediction.

Chapter 25, *Bayesian Inference*, focuses on Bayes' Theorem and how it can be applied to compute probabilities. Bayesian inference allows causes to be inferred from effects, rather than the usual inferring of effects from causes.

Chapter 26, *Interacting with the Desktop*, enumerates the options NQL offers for interacting with the desktop. This includes opening documents, displaying console windows, rendering text-to-speech, and playing multimedia files.

Chapter 27, *Network Monitoring*, focuses on the use of the SNMP protocol to interact with network devices. Network devices can be discovered, monitored, and controlled.

Chapter 28, *Web Applications*, explores the use of NQL to drive solutions that can be accessed from Web browsers. Like ASP and Perl, NQL scripts can power Web applications.

Chapter 29, *Supporting Mobile Devices*, explains how to write Web applications that can be accessed by mobile devices. A variety of PDAs and phones can be supported.

Chapter 30, *System Actions and Information*, covers actions the operating system can perform, such as powering down the computer; and information the operating system can supply, such as the current user's privileges.

Chapter 31, *Calling NQL as a Component*, is about the use of NQL as a callable object from other development environments. It is possible to gain the benefits of NQL without giving up the use of your favorite programming language.

Chapter 32, *NQL Cookbook*, is just what it sounds like: a collection of recipes for putting NQL to work for a particular task. For each task listed in this chapter, a step-by-step procedure and sample script are included.

Appendix A, *NQL Versions and Editions*, is your ammunition against version and platform confusion. It describes editions of the language and platform differences. Past and present versions of NQL are also described.

Who Should Read This Book

If you need to develop connected applications quickly, this book (and Network Query Language) are for you. Since NQL appeals to both junior and experienced developers, the book doesn't assume that you have a deep background in computer science. It is assumed that you understand the rudiments of programming, however. Programmers, IT personnel, Web masters, and content engineers can all put NQL to good use in short order.

If you really want a firm grounding in NQL, then by all means read the book straight through and try each of the tutorials. If you're short on time or have a looming deadline, here are some recommendations:

If you're a junior programmer, read Chapters 1 through 5 carefully, which will give you a good grounding in essentials. Go through the tutorials in these chapters and make sure you understand them. The remaining chapters of the book are topical, so when you need to do something new, like sending mail messages, go to the appropriate chapter. You might want to briefly scan through all of the chapters to develop a feeling for what NQL is capable of. You will find many common tasks addressed in Chapter 32, including scripts that can serve as templates for your own solutions.

For experienced programmers, you'll probably want to skim through Chapters 1 through 5 to get a feel for the language, taking note of NQL's distinguishing characteristics and pausing for an extended read when you come across something interesting. The remaining topical chapters can be read in any order desired, and contain tutorials for getting up to speed quickly. If you plan to use NQL to drive Web applications or mobile applications, you'll want to read Chapters 28 and 29 carefully. If you plan on calling NQL as a component from another language, see Chapter 31. The cookbook in Chapter 32 provides many ready-to-run scripts for putting NQL to use right away.

Conventions

This book uses certain conventions for showing code examples and reference forms of statements and functions.

Names of Statements, Functions, and Parameters

In the body text of the book, names of statements, functions, and parameters are often italicized. For example, "the *replace* function requires two parameters, a *search-string* and a *replace-string*."

Code Examples

Code examples appear in monospaced type.

```
open filename
while read(line)
{
    show line
}
close
```

When part of a code example is left to the imagination, an italicized line beginning and ending with an ellipsis (...) appears.

```
open filename
while read(sLine)
{
    ...do something with sLine...
}
close
```

Small code examples are in-line with the body text; code examples that are larger or are especially significant are cited by a listing number, such as "Listing 8.1."

```
//movie-reviews - reads movie reviews out of sample Access database
and displays results

dbpath = "c:\\Program
Files\\NQL\\Samples\\Scripts\\Database\\movies.mdb;"

dbprotocol "ado"
if opendb("Driver=Microsoft Access Driver (*.mdb); DBQ=" dbpath)
{
    boolean bHaveData = select("select * from movies")
    while bHaveData
    {
        dumpvars
        bHaveData = nextrecord()
    }
}
else
{
    show "Unable to open the database " dbpath
}
closedb
```

Listing 8.1 Database example.

Statement and Function Reference Forms

When a statement or function is first introduced, its reference form is usually given. Reference forms appear in monospaced type. Although NQL will accept upper-, lower-, or mixed-case keywords, the official rendering of these keywords in the language is lower case, and that is how they appear in this book.

```
end
```

Parameters are italicized, indicating the places where you need to supply values. Statement reference forms indicate a keyword and some number of parameters. Function reference forms indicate a return type, a keyword, and some number of parameters in parentheses.

```
open filespec
boolean open(filespec)
```

Parameters that must be variables are indicated by the word *var*. When a variable list (one or more variables) is needed, the term *var-list* is used.

```
clear var
output var-list
```

Square brackets [ ] indicate optional elements. An ellipsis (...) within square brackets indicates elements that can repeat multiple times.

```
nextline [var]
for var = [initial, ] final [, increment]
sendmessage recipient, subject, message1, type1 [ ... , messageN, typeN]
```

If there are many statements or functions to be listed, they are arranged in a table.

Planned Features

Some NQL language features described in this book are described as *planned features*. This means that these features were on the drawing board for Network Query Language and were considered likely to be included in the 2.0 release, but a final decision had not been committed to as of the time of this writing. In tables of statements and functions, an asterisk signifies a planned feature. In body text, the phrase *Planned feature:* precedes discussion of planned features.

Names of Web Sites

Many of the sample scripts in this book access Web sites. Rather than using actual Web site names, this book often uses generic placeholder names such as www.my-site.com, www-my-news-site.com, and www.my-retail-site.com.

Tools You Will Need

To get the most out of this book, you'll need a copy of Network Query Language so that you can try things out. Even if you don't own a copy, you can obtain an evaluation edition, either from the book's companion CD-ROM or the Internet. The version of NQL reflected in this book is Version 2.0, so you'll do best to have NQL 2.0 or later installed. If you can only get NQL 1.1 for your platform, this book will still be useful to you. Appendix A will help you keep your sanity if you happen to be using a different version of NQL.

What's on the CD-ROM

The companion CD-ROM contains software and code examples.

The software on the CD-ROM includes three versions of Network Query Language:

- Network Query Language for Windows version 2.0 (60-day evaluation)
- Network Query Language for Windows version 1.1 (60-day evaluation)
- Network Query Language for Java version 1.1 (60-day evaluation)

As of this writing, the Windows edition of NQL 2.0 was just nearing readiness. If you're running on a non-Windows platform, visit www.nqli.com or www.networkquerylanguage.com to determine what the latest version of NQL is, and what platforms it is available for. If you can't get your hands on NQL 2.0 or later for your type of computer, take heart. NQL 1.1 is available in both a Windows edition and a Java edition that runs on all major computing platforms. If you have to use NQL 1.1 instead of NQL 2.0, see Appendix A for some helpful information.

The CD-ROM also contains all of the NQL script code cited in the book as tutorials. When you go through each chapter's tutorial, you have the choice of entering script code by hand or copying the script from the CD-ROM. If a tutorial script requires supporting files, these are included as well.

Lastly, the scripts from Chapter 32, the NQL cookbook, are also included on the CD-ROM. These ready-made scripts can reduce your learning curve significantly.

Where To Go from Here

It's my sincere desire to see you and the rest of the world's software developers do more powerful things in a shorter timeframe. Network Query Language can be instrumental in making that happen. If it becomes part of your standard collection of programming tools you will find yourself being a programming superstar more often, and with less effort! In this book, I've sought to justify the reasons for NQL's existence; entice you with its power, brevity, and ease of use; educate you through tutorials; and equip you with essential reference information. The rest is up to you and your imagination.

I am happy to hear from readers, and can be reached at nqlbook@hotmail.com.

David Pallmann
Mission Viejo, California

Acknowledgments

Although I am the inventor of Network Query Language, I can hardly take sole credit for it! Many talented people have contributed to the NQL effort over the years. Christopher Hunter and John Ross have been integral to the language all along, from its earliest inception to present day. In version 2.0, Brandon Bethke and Jed Stumpf also served major architectural roles, taking charge of the compiler and runtime systems. The language also benefited from the ongoing engineering efforts of Robert Apodaca, Scott Bishop, Mike Campbell, Brian Connolly, Matt Grofsky, Ashur Novick, Peter Tran, and Bill Walker. Gentlemen, I salute you all.

It takes more than engineers to see a new product developed. I would like to thank Denny Michael and Doug Tullio, both of whom believed in NQL enough to dedicate years to seeing it advance from an interesting concept into a commercial-grade product. Their tireless efforts provided the conditions under which NQL could be developed.

To my wife Becky and daughters Susan and Debra, I must express my heartfelt thanks for your support, without which it would have been impossible to write a book. Putting up with a husband and dad with his nose in a laptop every waking moment can’t have been enjoyable. The good news is, I’m back!

I am indebted to the fine folks at John Wiley & Sons for making the writing and editing process easy for me, and for their dedication to professionalism.

CHAPTER 1

Presenting Network Query Language (NQL)

This chapter gives you a first look at Network Query Language (NQL). It is often instructive to gaze at something new from 30,000 feet before zooming in on the minute detail, and that is certainly true of computer languages. Before diving into the nuts and bolts of NQL, we want to consider the big picture: Its proper uses, its design philosophies, and its unique features. This chapter offers a first look at NQL, discusses appropriate uses for the language, and examines its unique qualities. The chapter ends with a tutorial in which you will enter, save, and run your first NQL program.

A First Look

It is traditional when introducing new languages to show a “Hello, World” program. So let’s get that out of the way. The one-line script that follows displays “Hello, World” on the screen.

```
show "Hello, World!"
```

That’s hardly very interesting, so let’s look at something slightly more ambitious (and useful). The following script is a complete NQL program.

```
get "http://www.my-news-site.com"
match '<li><a href="{link}">{headline}</a>'
```

```
while
{
     output headline, link
     nextmatch
}
```

Even if you've never seen NQL before, you can probably make an educated guess about the purpose of this program. It accesses a news Web site, locates the news stories, and outputs the headlines and links as eXtensible Markup Language (XML). A few things should strike you about this program:

- It is very small.
- It is very clear.
- It does something that is not easy (or even possible) in many mainstream programming languages.

Appropriate Uses for NQL

Network Query Language is ideally suited for creating connected applications such as bots, agents, spiders, middleware, Web applications, and mobile applications.

Bots are automated programs (the term bot comes from the word robot). Bots are essential because of infoglut. The amount of information on the Internet is constantly growing, and the same is true of electronic information within corporate networks. Yet a human being's ability to read, digest, and work with information is essentially fixed. Thus, there is an *information gap* that grows worse each every day. Bots come to the rescue by automating tasks for people and making life manageable again.

Agents are bots that serve an individual. In the real world, we employ human agents for all kinds of purposes: Travel agents, insurance agents, real estate agents, accountants, attorneys, and so on. Agents specialize in an area and represent our personal interests. The same is true of software agents: They are very good at a specific task, and they have a clear understanding of their master's wishes.

Spiders are bots that crawl the Internet, indexing Web pages and constructing the databases that power search engines. Spiders can be used for other purposes beside search engines, such as creating site maps.

Middleware applications connect two or more systems or applications, helping them to work in harmony as part of a larger system. Middleware is valuable because it avoids having to change major systems, instead taking the approach of interfacing between them.

Web applications are the modern gateway to interfacing with companies and other organizations. Web applications perform such diverse functions as conveying information, searching databases, initiating actions, and effecting transactions.

Mobile applications are Web applications that support Personal Digital Assistants (PDAs) and cellular phones rather than desktop Web browsers. Mobile applications have to be concerned with supporting a variety of devices and accommodating a small display size, different output formats, and a simple user interface.

Why a New Language?

Network Query Language came about after repeated disappointment with mainstream programming languages. It wasn't that these languages were especially deficient; rather, what has changed is the focus of modern software development. Since the rise of networking and the Internet, there has been a shift toward new kinds of applications that stress communication, content transformation, distributed processing, and smarter software. That means the statements and functions that have been standard in languages for decades no longer serve us so well. In an age where programmers are performing new tasks and meeting new challenges, we need some new tools in our toolbox.

Design Philosophies

It is no accident that NQL scripts are powerful yet remain small and elegant—that is a reflection of the design philosophies that went into the language. The benefits to you are more power, clarity of code, speedier development, and the ability to extend the language further.

Power

NQL is an extremely powerful language. It has far more built-in capabilities than other programming languages. There are over 500 statements and functions in NQL, many of which aren't found elsewhere. This unrivaled collection of building blocks is at your immediate disposal. Many of the one-liners in NQL accomplish things that would normally require large, exceedingly complex programs in their own right.

Many difficult tasks become simple in NQL. It might be beyond you to directly write programs that interact with Web sites, send and receive email, export data to applications, or perform artificial intelligence—but these kinds of activities are ridiculously easy in NQL.

Clarity

NQL is intended not only for professional programmers, but also for anyone who needs to quickly create a connected application, such as IT personnel, content engineers, and Webmasters. With this in mind, NQL makes no assumptions about computer science background or prior programming experience. The result is a language that has a simpler syntax than Visual Basic yet retains its power.

The names of statements and functions in NQL have been selected to be as straightforward as possible. Whereas some languages use ornate names for historical reasons, NQL's keywords tend to be one or more real words found in the dictionary. It is easy to discern the meanings of statements with intuitive names like *openmail*, *sendmessage*, and *firstunreadmessage*.

Likewise, the use of cryptic punctuation has been strenuously avoided. If an NQL statement requires parameters, they simply appear to the right of the statement. If there is more than one parameter, they are separated by commas. A newline separates statements, so there's no need for an end-of-statement character such as the semicolon used in C++ and Java.

Rapid Development

NQL promotes rapid development. The aim is to turn months into weeks, weeks into days, and days into hours. Features that facilitate rapid development include the following:

- A fast, friendly development environment
- A scripting language design that does not require a separate compilation process
- Powerful building blocks built directly into the language
- A clear, compact syntax
- A safe, forgiving runtime environment

When you craft a program in a standard programming language, you typically combine hundreds, thousands, or tens of thousands of instructions. Naturally this takes time to write and debug. When programming in NQL, programs are composed of far fewer statements because of the powerful building blocks built into the language. Sometimes a complete NQL script is just a handful of statements. Tiny programs mean faster development, a smaller likelihood of errors, and faster debugging.

Let's consider a case in point. Say you want to crawl a Web site and download files, a task that is easy to talk about conceptually but is not trivial to program. Site crawling requires a careful understanding of Web technologies in order to be done well and normally an intermediate to senior level developer would be needed to implement such a program. In a language like C++ or Java, crawling a Web site requires 20 or so pages of code. In NQL, the task can be accomplished in just one statement, because site crawling and file downloading are concepts built into the language.

Familiarity

NQL also stresses familiarity. Some languages make the error of being unique to the point of alienating developers. Although NQL has many unique features and a unique grammar, it also provides plenty of things programmers will find familiar. Among these are the following:

- Typed variables
- Operators and expressions
- Multi-statement code blocks

```
// Retrieve news headlines from a news site main page,
// output to a text file (news.text), and
// open the results on the desktop.

String headline
String link
Integer stories

Create "news.txt"

Get "http://www.my-news-site.com"

stories = 0

Match '&#149;&nbsp;<A href="{link}">{headline}</A>'
While
{
    Write headline & ","
    Write link & "\r\n"
    stories = stories + 1
    NextMatch
}
Close

OpenDoc "news.txt"
```

Listing 1.1 Visual Basic-like phrasing.

- Standard flow of control statements
- User-defined functions

To illustrate the point, take a look at the script code in Listings 1.1, 1.2, and 1.3. All three scripts perform identically, but they don't look all that similar. If you're from a Visual Basic background, Listing 1.1 probably looks most familiar to you.

If you come from a C++ or Java background, you're probably more at-home with Listing 1.2. NQL supports a number of conventions from Visual Basic, C++, and Java, possessing aliases in many cases. Most statements can also be treated as functions.

If you've used assembly language or a stack-oriented language such as Forth or PostScript, you may relate best to Listing 1.3. The use of NQL statements and the stack allows one to program in terms of instructions that work together, just like the instructions in a microprocessor.

As you can see, there is a great deal of freedom in how programs are expressed in NQL. Capitalization is immaterial, as is indenting and the use of white space (that is, spaces, tabs, and blank lines).

```
/*************************************************
* Retrieve news headlines from a news site page, *
* output to a text file (news.text), and         *
* open the results on the desktop.               *
*************************************************/

string headline, link
int stories = 0

if create("news.txt")
{
    get "http://www.my-news-site.com"
    while match('&#149;&nbsp;<A href="{link}">{headline}</A>')
    {
        write headline ","
        write link "\r\n"
        stories=stories+1
    }
    close
    opendoc "news.txt"
}
```

Listing 1.2 C++/Java-like phrasing.

Customizable and Extensible

NQL is a customizable language. Every keyword in the language can be redefined. The same is true for every error message. This level of customization makes it easy to handle internationalization. All it takes to change keywords or messages is editing and compiling a message file.

As powerful as NQL is, it can be extended. Anyone can write new statements for the language, which become seamless additions indistinguishable from the core statements in the language. In the Windows edition of NQL, a Dynamic Link Library (DLL) (created in any language) can implement new statements. In the Java edition of NQL, a Java class can implement new statements. Third parties with experience in specific areas of technology or industries can leverage their expertise and make contributions to NQL.

The Right Building Blocks

What are the building blocks that make NQL special? They fall into four main categories: *communication*, *data conversion*, *automation/distributed processing*, and *intelligent behavior*. Connected applications frequently need to draw on building blocks in these categories. For example, consider an agent that retrieves news stories from a Web site every 24 hours and stores them as XML in a local database. The agent utilizes commu-

```
; Retrieve news headlines from a news site page,
; output to a text file (news.text), and
; open the results on the desktop.

        STRING  HEADLINE
        STRING  LINK
        INT     STORIES

        CREATE  "news.txt"
        GET     "http://www.my-news-site.com"

        STORIES =0

        MATCH   '&#149;&nbsp;<A href="{LINK}">{HEADLINE}</A>'

        WHILE
        {
                WRITE   HEADLINE | ","
                WRITE   LINK | "\r\n"
                STORIES =STORIES+1
                NEXTMATCH
        }
        CLOSE

        OPENDOC "news.txt"

        END
```

Listing 1.3 Assembler-like phrasing.

nication to retrieve Web pages, conversion to generate XML, automation to run every 24 hours, and intelligent behavior to identify news articles on Web pages.

Communication

NQL is full of support for Internet and network protocols. The communication facilities allow you to retrieve information from just about anywhere, whether on the Internet or corporate network. They also allow you to deliver information, results, or notifications to people and systems, whether nearby or remote. The communication facilities include the following:

Web sites. The ability to read Web pages, crawl Web sites, and interact with Web sites.

File input/output. Reading and writing disk files.

File Transfer. Sending and receiving files between computers.

Databases. Querying databases, and adding, updating, and deleting database records.

Email. Sending and receiving email.

Newsgroups. Scanning newsgroups.

Directory Servers. Querying directory servers.

Terminal Emulation. Accessing UNIX systems and legacy systems.

Interaction with Applications. Controlling applications under program control, and exporting.

Sockets. TCP/IP input/output.

Serial Communications. Serial port input/output.

e-Commerce. Processing credit card transactions and accessing financial servers.

Network Monitoring. Detecting network devices, reading their status, and sending them commands.

Mobile Devices. Recognizing mobile devices and outputting data in their content languages.

Web Services. Accessing XML-based services over the Internet.

Text-to-speech. Speaking content on the desktop.

Audio. Playing sound and music files on the desktop.

Video. Playing movie files on the desktop.

Data Conversion and Analysis

Because NQL can work with data in different ways and perform conversions between common data types, it is possible for scripts to transform content. Coupled with NQL's communication capabilities, scripts are able to acquire content, analyze and transform it, and deliver the results to users in just the format they desire. Some of NQL's data-related capabilities include the following:

XML. Parsing, validating, creating, importing, and exporting eXtensible Markup Language, the standardized data language for representing all kinds of content.

HTML. Parsing, generating, importing, and exporting HyperText Markup Language, the data language of Web pages.

Delimited data. Reading, writing, importing, and exporting delimited data files.

Images. Creating, manipulating, and converting graphic images.

Date/Time. Obtaining current date and time, converting to and from various formats.

Filtering. Removing unwanted data.

Ranking. Analyzing and reordering information so that the most relevant items appear first.

Conversion. Transforming data into other formats.

Strings. Manipulating strings and text, and generating formatted output.

Automation and Distributed Processing

Connected applications are frequently automated and/or distributed. Automated solutions run all by themselves, either on a schedule or in response to events. Distributed solutions require cooperation between multiple programs working together as a team.

Scheduling. Performing tasks at set dates and times, and repeating tasks at regular intervals.

Inter-process communication. Communication between agents for cooperative processing on a large scale.

Synchronization. Locking and unlocking to manage access to shared resources and to coordinate processing.

Application control. Launching applications, on local or remote systems, and passing them command line arguments, data, or keystrokes.

Intelligent Behavior and Analysis

An essential ingredient in constructing reliable software with good decision-making capabilities is intelligence. NQL includes the following features that can be used to make software smarter:

Neural Networks. Adaptive software that learns and can be applied to pattern recognition, prediction, and organizational tasks.

Bayesian Inference. Predictive statistics that determine causes from effects.

Fuzzy Logic. Algorithms dealing with fine degrees of belief and probability.

Pattern Matching. Ability to recognize objects based on patterns or landmarks.

Categorization. Procedure of examining text and Web pages and assigning them categories.

Miscellaneous

In addition to the four main categories just covered, there are some additional NQL capabilities worth mentioning. They include the following:

Agent characters. Desktop characters that can display information, speak, and perform actions.

Registry access. Reading and writing information to the system registry.

System information. Determining operating system, current directory, available memory and resources, and other information.

System actions. Powering down, shutting down, or rebooting the computer, and logging off the current user.

Role Independence

Network Query Language stresses role independence, allowing you to utilize NQL wherever convenient. NQL scripts can execute on the desktop, on a server, as a Web application, and as a component. This means you can use NQL to be that missing puzzle piece in your enterprise, taking on the size, shape, and location needed.

Desktop

On workstations, NQL scripts may run as desktop agents. The agents can be out of sight or can take on a console appearance that resembles an MP3 player. The desktop is a logical choice for these kinds of applications:

- Applications that need to speak, play audio, or play video on the desktop
- Applications that need to interact with users
- Applications that need to be on the desktop for permission/security content reasons, such as an agent that reads incoming email messages
- Applications that need to perform on-screen notification, such as displaying messages or animating agent characters
- Applications that need to launch, export to, or otherwise interact with other desktop applications

Servers

On servers, NQL scripts can perform actions that service entire departments or organizations rather than individuals. Servers are logical places for these kinds of applications:

- Applications that need to respond to clients
- Applications that need to process large amounts of data based on queues, lists, or databases, such as shopping bots
- Middleware that interconnects two or more systems
- Scheduled applications that run in the background

Web Servers

On Web servers, NQL applications can operate as CGI applications, just as applications in languages like ASP, JSP, PHP, and Perl do. Web servers are a logical place for the following kinds of applications:

- Web applications, accessible from desktop Web browsers
- Mobile applications, accessible from PDAs and phones
- Applications that need to be accessible from anywhere in the world

Components

NQL can be called as a component, which gives you the luxury of continuing to use other languages and development environments but still tap into NQL's power when you need it. Some instances where you might want to call NQL as a component include the following:

- When you are extending existing projects already developed in another language
- When you have programmers already comfortable in working with a different language
- When you want to add communication, conversion, and intelligence features easily to extensible applications, such as Microsoft Office

Tutorial: Directory

Now that you know something about NQL's capabilities, let's develop and run a complete application. In this tutorial, you will create a script that examines a disk directory, creates a Web page that lists the files, and opens the Web page in a browser on the desktop. The result will be a Web page like the one shown in Figure 1.1.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Directory script.
3. Run the Directory script.
4. Understand the Directory script.

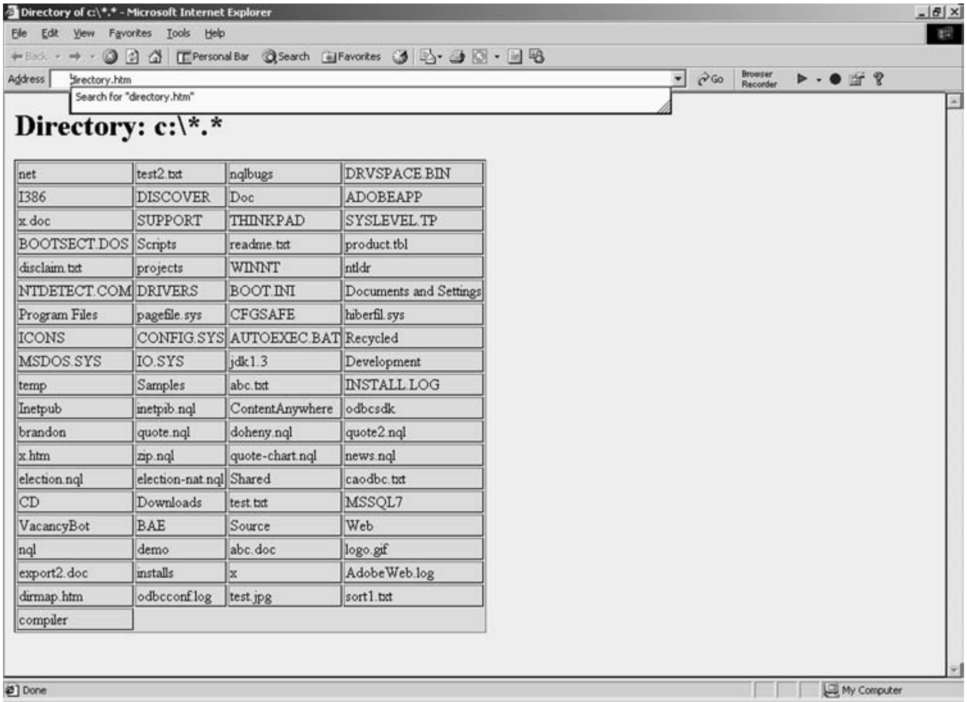
When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve the names of files in a disk directory.
- Create a Web page and write HTML to it.
- Open a Web page in a browser on the desktop.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or command line method of launching the NQL Client.


Directory of c:\*.* - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address directory.htm
Search for "directory.htm"
Go
Browser Recorder
Done
My Computer

Directory: c:\*.*

net	test2.txt	nqlbugs	DRVSPACE.BIN
I386	DISCOVER	Doc	ADOBEAPP
x.doc	SUPPORT	THINKPAD	SYSLEVEL.TP
BOOTSECT.DOS	Scripts	readme.txt	product.tbl
disclaim.txt	projects	WINNT	ntldr
NTDETECT.COM	DRIVERS	BOOT.INI	Documents and Settings
Program Files	pagefile.sys	CFGSAFE	hiberfil.sys
ICONS	CONFIG.SYS	AUTOEXEC.BAT	Recycled
MSDOS.SYS	IO.SYS	jdk1.3	Development
temp	Samples	abc.txt	INSTALL.LOG
Inetpub	inetpib.nql	ContentAnywhere	odbcsdk
brandon	quote.nql	doheny.nql	quote2.nql
x.htm	zip.nql	quote-chart.nql	news.nql
election.nql	election-nat.nql	Shared	caodbc.txt
CD	Downloads	test.txt	MSSQL7
VacancyBot	BAE	Source	Web
nql	demo	abc.doc	logo.gif
export2.doc	installs	x	AdobeWeb.log
dirmap.htm	odbcconf.log	test.jpg	sort1.txt
compiler			

Figure 1.1 Directory Web page.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter your first NQL script.

Step 2: Enter the Directory Script

In the NQL development environment, enter the script shown in Listing 1.4 and save it under the name directory.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy directory.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select directory.nql from the Tutorials\ch01 folder.

At this point, the script in Listing 1.4 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Directory Script

Now, take a deep breath and run the script. You can do this by selecting *Build, Run* from the menu, clicking the Run toolbar button, or pressing F5. Almost immediately, a

```
//directory - creates a Web page that shows the contents of a disk
directory

string path = "c:\\"

int size = length(path)

path = path & "*.*"

if !dir(path) { end }

create "directory.htm"

write "<html><head><title>Directory of {path}</title></head>"
write "<body bgcolor=#000088 text=#ffffff>"
write "<h1>Directory: {path}</h1>"
write "<table border=1 bgcolor=#0000FF><tr>"

while nextline(file)
{
    file = mid(file, size+1)
    write "<td>" file "</td>"
    files = files+1
    if files>=4
    {
        write "</tr><tr>"
        files = 0
    }
}

write "</tr></table>"
write "</body></html>"

close

opendoc "directory.htm"

end
```

Listing 1.4 Directory script.

Web page should open up on the desktop that shows a directory of the files in c:\. If this does not happen, check the following:

1. Make sure you have a c: drive. If you don't, change the path in the script.
2. Make sure there are files in the c:\ root directory. If there aren't, change the path in the script.

3. Ensure that you are running directory.nql from a disk directory that you can write to. If not, copy the script to a different location, such as your primary hard drive.
4. Ensure that you have a default application for displaying .htm files (most systems assign a Web browser for this purpose).
5. Check the NQL client's Errors tab for error messages.

At this point you should have seen the directory script run, generating a Web page that lists the files in a disk directory. Although we haven't discussed how it works yet, you've seen the results of an NQL script first-hand, in this case exercising NQL's abilities to read disk directories, generate Web pages, and open files on the desktop. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Directory Script

We now want to make sure we understand every part of the directory script. The first line is a comment line.

```
//directory - creates a Web page that shows the contents of a disk
directory
```

Next, a variable named *path* is declared and initialized. The word string indicates that *path* is a string variable. The path is set to c:\, the directory to be displayed in the Web page.

```
string path = "c:\\"
```

A second variable named *size* is now declared and initialized. The data type is integer, and the value is the length of the path string. The variable *size* will be used to separate the filename from the path prefix later in the script.

```
int size = length(path)
```

The path is appended with *.* so that it is now a full wildcard directory specification of c:\*.*.

```
path = path & "*.*"
```

A directory of the path is requested using the *dir* function. If a directory cannot be obtained, *dir* returns false and the script ends. If the directory is obtained, it is stored on the stack as a list of filenames separated by newlines.

```
if !dir(path) { end }
```

It is now time to create the Web page and write some initial HTML to it. The *create* statement creates a new file and opens it, overriding any previous file of the same name. We call the Web page directory.htm. The *write* statements output some initial HTML, ending in the beginnings of a table.

```
create "directory.htm"

write "<html><head><title>Directory of {path}</title></head>"
write "<body bgcolor=#000088 text=#ffffff>"
write "<h1>Directory: {path}</h1>"
write "<table border=1 bgcolor=#0000FF><tr>"
```

Now to loop through the filenames returned by the *dir* function earlier. A *while* loop uses *nextline* to get the next line from the data on the stack. The loop runs as long as there are files left to process, leaving the current file to work with in the variable named *file*.

```
while nextline(file)
{
```

Within the loop, the filename is reduced to eliminate the path (drive and folder) portion, using the *mid* function and the *size* variable calculated earlier. The file name is then written out as an HTML table element. If more than four files across have been written, additional HTML is written out to move to the next table row.

```
    file = mid(file, size+1)
    write "<td>" file "</td>"
    files = files+1
    if files>=4
    {
        write "</tr><tr>"
        files = 0
    }
}
```

Once all of the files have been written to the HTML, the table needs to be finished and the final HTML suffix code needs to be output. More *write* statements take care of this.

```
write "</tr></table>"
write "</body></html>"
```

The HTML file is now finished and can be closed with *close*.

```
close
```

Lastly, we want to open the newly created Web page on the desktop. The *opendoc* statement accomplishes this.

```
opendoc "directory.htm"
```

An *end* statement terminates the script.

```
end
```

At this point, you've not only seen an NQL script in action, you've also had a look under the hood as well. The remaining chapters of this book go into more detail about the families of NQL statements and functions.

Further Exercises

You could amend this example in a number of ways. Once you've learned more about NQL from the chapters that follow, you may want to come back to this example and try your hand at extending it. Here are some interesting modifications that can be made:

- Show different types of files in different colors. For example, .doc files in blue and .xls files in green.
- Show more information about each file than just its name, such as the size of the file.
- For text files, display an excerpt from the file.

Chapter Summary

Network Query Language is oriented toward the development of connected applications such as bots, agents, spiders, middleware, Web applications, and mobile applications. The design philosophies behind NQL yield power, clarity, and rapid development while retaining familiarity to programmers and permitting extensibility.

NQL contains over 500 building blocks that fall into four main categories: communications, data conversion, automation/distributed processing, and intelligent behavior. Powerful scripts can be created quickly from just a handful of these building blocks.

NQL features role independence, which means it can be put to use anywhere it is needed. Scripts can run on the desktop, on servers, on Web servers, and as components. Although complete applications can be developed in NQL, it is also possible to combine NQL with other languages.

You've had your first look at NQL. Hopefully you've seen a lot to get excited about. The chapters that follow explore the language's rules and capabilities in detail, with plenty of examples along the way.

CHAPTER 2

Developing in NQL

Working effectively in a programming language involves more than the language itself. Various processes must be mastered, including entering and revising code, running programs, debugging programs, and deploying programs. The NQL development system includes a graphical development environment with an integrated debugger. This chapter describes the development environment and the procedures for writing programs, testing them, and deploying them. In the tutorial at the end of the chapter you will enter, run, debug, revise, save, and deploy an NQL script that creates pie charts.

The Development Environment

The NQL development environment is neither as primitive as Notepad nor as sophisticated as development tools that have been around for many years. What you'll find is a reasonable code editor with the primary features a developer needs. Of course, you're free to use another development environment if you choose. Since NQL scripts are just text files, various programming editors and text editors can be used to write and modify NQL scripts. Most likely, though, you'll want to use NQL's development environment for the simple reasons that you can run and debug your scripts from the same environment in which you write them, and have direct access to the integrated documentation.

Depending on your operating system, the NQL development environment may be launched in different ways. It is normally referred to as the NQL client. Under Microsoft Windows, the NQL client can be accessed from both a desktop icon and the Start Menu. Clicking on a .nql file will also cause it to open in the NQL client. Figure 2.1 shows the NQL development environment.

Capabilities

First and foremost, the development environment lets you enter programs, save them, load them, and revise them. It also lets you run them and debug them. The full set of capabilities provided by the development environment is listed below.

Coding. Entering and editing code.

Opening scripts. Loading existing scripts from disk into the editor.

Saving scripts. Saving code from the editor to disk.

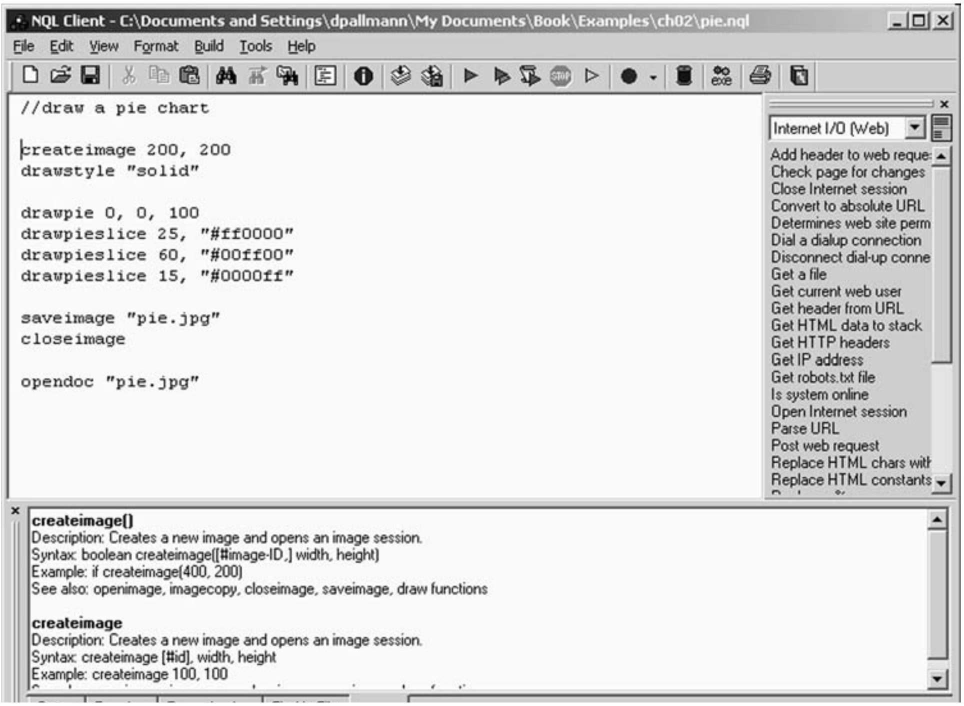
Foreground execution. Running scripts in the primary thread (foreground).

Background execution. Running scripts in worker threads (background).

Debugging scripts. Single-stepping scripts through the debugger.

Halting execution. Stopping a running script.

Searching. Finding and replacing text in scripts.



```
//draw a pie chart

createimage 200, 200
drawstyle "solid"

drawpie 0, 0, 100
drawpieslice 25, "#ff0000"
drawpieslice 60, "#00ff00"
drawpieslice 15, "#0000ff"

saveimage "pie.jpg"
closeimage

opendoc "pie.jpg"
```

createimage()
Description: Creates a new image and opens an image session.
Syntax: boolean createimage([#image-ID,] width, height)
Example: if createimage(400, 200)
See also: openimage, imagecopy, closeimage, saveimage, draw functions

createimage
Description: Creates a new image and opens an image session.
Syntax: createimage [#id], width, height
Example: createimage 100, 100

Figure 2.1 The NQL development environment.

Supplying input. Entering input data to scripts in XML or other formats.

Viewing output. Viewing output data from scripts in XML or other formats.

Reviewing execution. Viewing the execution log after a script has run.

Reviewing errors. Viewing the error log after a script has run.

Bot generation. Compiling a script into a desktop .bot file.

Executable file generation. Compiling a script into a desktop .exe file.

Printing. Printing scripts.

Authorization. Viewing or updating license information.

Documentation. Displaying the NQL online documentation.

Formatting. Formatting code.

Reporting problems. Submitting technical support inquiries.

Updating. Checking for updated software.

Scheduling. Registering a script for execution at a specific date and time.

The User Interface

Familiarity with the development environment's user interface is an important first step in becoming comfortable with NQL. The main screen of the development environment includes a menu, toolbars, an editing area, and a series of tabbed windows at the bottom of the screen.

The menu includes *File*, *Edit*, *View*, *Format*, *Build*, *Tools*, and *Help* main topics.

The toolbars provide easy access to the most commonly used menu items.

The editing area is the dominant part of the main window, where code is entered, viewed, and revised. This area is formatted in a monospaced Courier 10 font by default, but this can be changed in the options settings. Like most editors, text entered is normally inserted. By pressing the Insert key, you can toggle between text that inserts and text that overwrites.

To the right of the code window is a reference window for looking at topical families of NQL statements and functions. Double-click on an entry to insert NQL statements into your code.

The tabbed areas at the bottom of the window are labeled *Output*, *Error Log*, *Execution Log*, *Find in Files*, and *Help*.

The *Output* tab shows script output. As a script runs, it may produce output in XML or other formats. After a script completes, any output produced by it will be displayed in the *Output* tab. By clicking on the *Output* tab, you can view this output or copy it to the clipboard.

The *Error Log* tab lists any errors that occurred during script execution. If you click on an error in the error log, you will be taken to the source code line where the error is.

The *Execution Log* tab lists a complete summary of the script execution. The execution log can be turned on, turned off, or routed to a file rather than memory. You probably don't want the execution log on during production use of your scripts, but it certainly is helpful when you are initially testing and debugging them. You can turn the Execution log on or off from the *Tools, Options* menu item.

The *Find in Files* tab is used to show the results of the *Find in Files* menu item, which searches multiple files on disk for a search string.

The *Help* tab is for dynamic help. As you move throughout the code editor, every time you set focus on an NQL keyword in your code, the Help tab becomes topmost and shows you reference information about that statement or function. You can turn this behavior on or off from the *Tools, Options* menu item.

The Development Process

Entering and running a script is easy. At its simplest, one launches the development environment, types in a script, and runs it. Since NQL is a scripting language, there's no compilation step. Running a script compiles and executes it at the same time. This is similar to how other scripting languages such as Active Server Pages or Perl work.

Entering a New Script

To enter a new script, simply launch the NQL development environment. If the development environment is already loaded, a new script can be started by selecting *File, New* from the menu; clicking the *New* toolbar button; or entering Ctrl+N. The code editing area blanks, ready for you to enter your script.

Once a script is entered, you will probably want to save it and try it out.

Saving a Script to Disk

After entering a script, it can be saved to disk by selecting *File, Save* from the menu; clicking the *Save* toolbar button; or entering Ctrl+S. There are two variations of the *Save* command:

- *File, Save* saves the current document. There is no prompting for the script name if it is already known. If the document is new, this command is the same as *File, Save As*.
- *File, Save As* always prompts for a directory and script name.

NQL scripts are always saved with a file type of nql.

Loading a Script from Disk

An existing script may be loaded into the code editor by selecting *File, Open* from the menu; clicking the *Open* toolbar button; or entering Ctrl+O. A dialog prompts for the directory and script file name to open. The script opens in the code editor, where it may be viewed or revised.

Executing a Script

A script in the editor can be executed in several different ways. The standard way to execute scripts is in the foreground. Other ways to run scripts are in the background or through the debugger.

Initially, NQL displays a run dialog each time you run a script, as shown in Figure 2.2. This dialog may be turned off by clearing the *Show this dialog each time a script is run* check box. The run dialog can be recalled when running a script through the debugger.

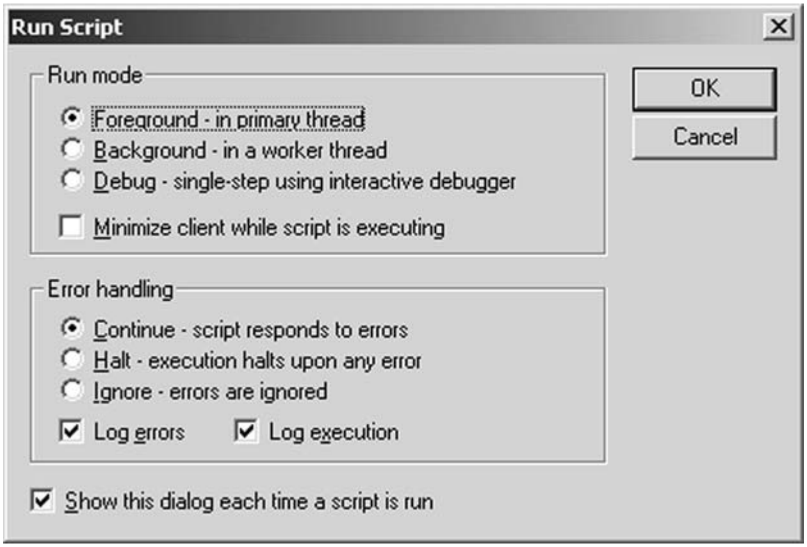
Running a Script in the Foreground

To run a script in the foreground, select *File, Run* from the menu; click the *Run* toolbar button; or enter F5. The script begins execution and an hourglass (or your operating system's equivalent) display. The development environment displays a message when the script completes. If you want to halt the script prematurely, you may select *File, Cancel All Scripts* or click the *Stop* toolbar button.

After a script completes running, there may be updated information in the *Output*, *Error Log*, and/or *Execution Log* tabs.

Running a Script in the Background

Running a script in a background thread allows you to continue working in the editor while it runs. To run a script in the background, select *File, Run in Background Thread*



Run Script

Run mode

● Foreground - in primary thread
○ Background - in a worker thread
○ Debug - single-step using interactive debugger

□ Minimize client while script is executing

Error handling

● Continue - script responds to errors
○ Halt - execution halts upon any error
○ Ignore - errors are ignored

☑ Log errors ☑ Log execution

☑ Show this dialog each time a script is run

Figure 2.2 The run dialog.

from the menu; click the *Run in Background* toolbar button; or enter F6. The script begins execution, but you are free to enter and run scripts. The development environment displays a message when the script completes. If you want to halt the script prematurely, you may select *File, Cancel All Scripts* or click the *Cancel All Scripts* toolbar button.

After a script completes running in the background, there may be updated information in the *Output*, *Error Log*, and/or *Execution Log* tabs.

Running a Script in the Debugger

To run a script in the debugger, select *File, Trace (Single Step)* from the menu; click the *Trace run* toolbar button; or enter Shift+F5. The script begins execution and an hourglass (or your operating system's equivalent) display. The development environment displays a message when the script completes. If you want to halt the script prematurely, you may select *File, Cancel All Scripts* or click the *Stop* toolbar button.

After a debugging session completes, there may be updated information in the *Output*, *Error Log*, and/or *Execution Log* tabs.

The Debugger

The debugger allows you to trace through scripts a line at a time, examining variables and other internals as execution progresses. The debugger displays a debug console dialog each time it is about to execute a statement in the script, shown in Figure 2.3. This dialog is the control panel for the debugging session.

Information on the debug console includes the following:

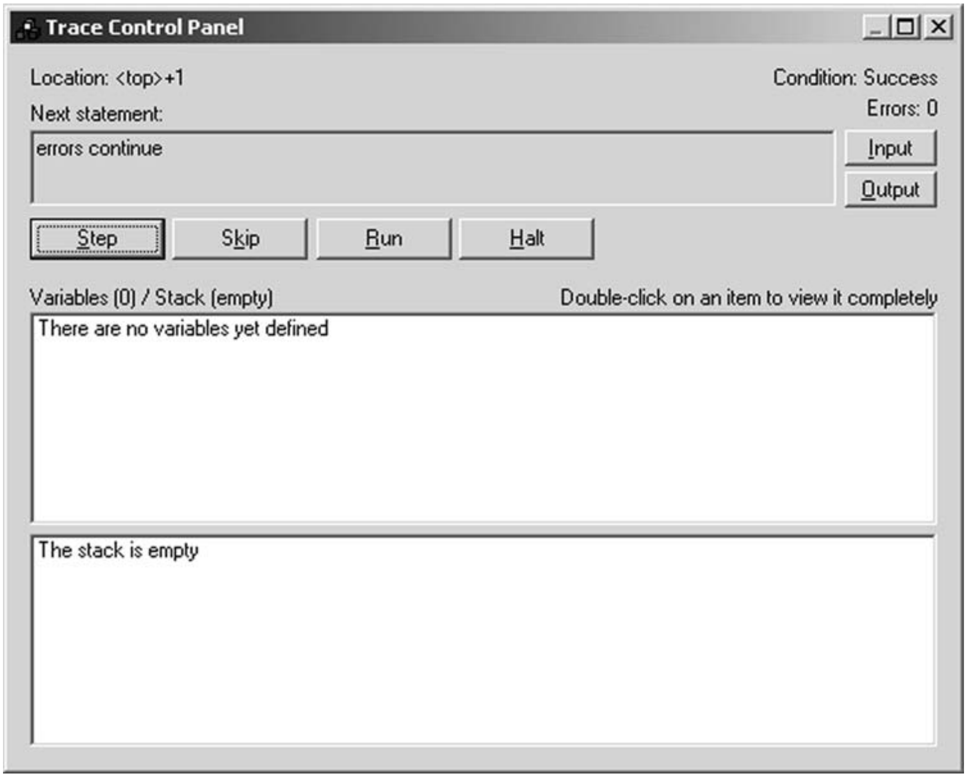
- The current location in the script
- The current success/failure condition
- The next statement to be executed
- The number of errors
- The contents of all defined variables
- The contents of the stack

Debug Commands

From the debugging console, the following command buttons are available:

Step. Execute the next statement, then redisplay the debug console. The Enter key is mapped to this button by default, so by repetitively pressing Enter you can step through your script one statement at a time.

Skip. Pass by the next statement without executing it.



Location: <top>+1

Condition: Success

Errors: 0

Next statement:

errors continue

Variables (0) / Stack (empty)

Double-click on an item to view it completely

There are no variables yet defined

The stack is empty

Figure 2.3 The debug console.

Run. Run the remainder of the script normally, without the debug console.

Halt. Terminate the script and end the debugging session.

Input. View the input stream, if any.

Output. View the output stream, if any.

Debug Statements

Some statements in NQL are designed to help debugging. The debugging-related statements are listed below.

- The *show* statement displays a message on the screen.
- The *dump* statement shows the contents of the stack on the screen.
- The *dumpvars* statement shows the names and contents of all variables on the screen.
- The *breakpoint* statement returns control to the debugger.

Compiling NQL Scripts

Although it is not necessary to explicitly compile an NQL script, there may be times when you want to deploy an NQL-powered solution without placing the source code in plain view. With most scripting languages this is not possible, but NQL provides two methods for distributing programs without putting the source code in plain view. The NQL client can compile NQL scripts in two ways, .bot files and .exe files.

.bot Files

NQL scripts can be compiled into an intermediate form known as a .bot file. A .bot file allows you to place an NQL program in compiled form on another computer without the source code.

Creating a .bot file is easy. From the development environment, select *Tools, Make Bot File*. A file with the same name as your script and a type of *bot* will be created.

The .bot file will run on any system that has the NQL runtime system installed.

.exe Files

Under Microsoft Windows, a desktop .exe file can be generated from an NQL script. An .exe file allows you to deliver an NQL program in the form of a Windows standard executable file.

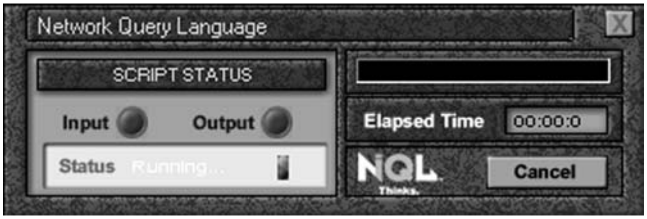
Executable files are visible on the desktop when they run. You can choose from a variety of skins, or appearances. Figure 2.4 shows the default skin.

Creating an .exe file is easy. From the development environment, select *Tools, Make Agent EXE*. A file with the same name as your script and a type of *exe* will be created.

The .exe file will run on any Windows system that has the NQL runtime system installed.

Deploying NQL Scripts

Once you've created an NQL script and have tested it, you will need to decide how to deploy it. NQL scripts can be run from many different contexts, including the following:



Status Running...

Elapsed Time 00:00:0

Figure 2.4 Desktop .exe appearance.

- As a manually-launched desktop application
- As a scheduled application
- As a server application
- As a Web application
- As a component

Manually-Launched Applications

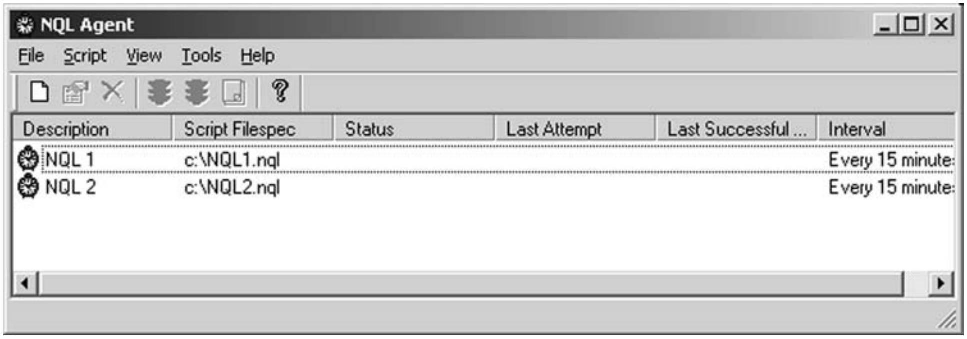
To manually launch an NQL application, place the .nql script file on the target computer. Under Windows, the script is run by right-clicking the .nql file and selecting *Run*. You can also create Windows desktop shortcuts to NQL scripts just as you can to any file or program.

You can also generate a .bot or .exe file in place of the .nql file. To execute a .bot file, right-click it and select *Run*. A generated .exe is launched like any other application, through a desktop icon or the Start menu.

Scheduled Applications

The NQL development system includes NQL Agent, a scheduling agent for putting script execution on a timer. The scheduler launches scripts at pre-specified times. On Windows, NQL Agent is launched by selecting *Programs*, *NQL Network Query Language*, *NQL Agent* from the Start Menu. NQL Agent shows up as an alarm clock icon in the system tray (bottom right) area of the desktop. Double-clicking the icon makes a scheduler window visible on the desktop, shown in Figure 2.5.

Scripts can be scheduled by date and time in NQL Agent. They can also be run manually. Scheduled scripts can be repeated at regular intervals. For example, you might schedule a script that performs backups to run at midnight and repeat every 24 hours.



Description	Script Filespec	Status	Last Attempt	Last Successful ...	Interval
NQL 1	c:\NQL1.nql				Every 15 minute:
NQL 2	c:\NQL2.nql				Every 15 minute:

Figure 2.5 NQL Agent.

Server Applications

NQL scripts can be run on servers, either because they respond to clients or because they perform organization-wide services. Server NQL applications can be launched manually, set up as NT services, or scheduled.

Web Applications

NQL scripts can be run as CGI applications, in the same way that languages like Perl, ASP, and JSP scripts can. When NQL is installed on a Web server, it offers to configure the Web server to recognize .nql files in a Web address as NQL scripts to execute. To deploy an NQL script on a Web server, simply place the .nql file(s) in a Web directory. Chapters 28 and 29 describe the use of NQL on Web servers in detail.

Tutorial: Pie

To apply what we've learned in this chapter, this tutorial will have you enter code in the development environment, debug and correct the program, and deploy it on the desktop. The program will create a pie chart such as the one shown in Figure 2.6. The focus of this tutorial will be the process you go through to develop in NQL, not the mechanics of how this particular script works (if you want to know that, refer to Chapter 21, which discusses graphics).

There are six steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the pie script.

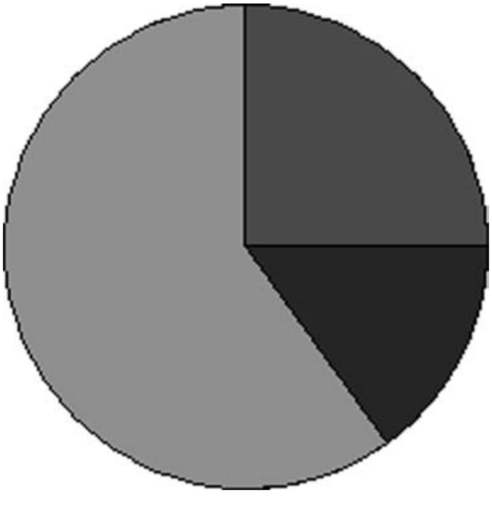


Figure 2.6 Pie chart.

3. Run the pie script.
4. Revise the script to correct errors.
5. Save the script.
6. Generate a desktop agent.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Enter a script by hand in the development environment.
- Run a script.
- Correct a faulty script.
- Save a script.
- Deploy a script.

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Pie Script

In the NQL development environment, enter the script shown in Listing 2.1. You do this by clicking in the code editor window to set focus, then typing in the script. Even if you prefer copying sample scripts from the companion CD, you should enter the script by hand in this case since the purpose of this tutorial is to acquaint you with the development environment.

At this point, the script in Listing 2.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Pie Script

Now, take a deep breath and run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5.

If all is well with the script, a pie chart will be created and opened on the desktop. However, that does not happen here. If you entered the script as described in the previous step, you receive an error message instead from the compiler. That's because we

```
draw a pie chart

createimage 200, 200

drawpie 0, 0, 100
drawpieslice 25, "#ff0000"
drawpieslice 60, "#00ff00"
drawpieslice 15, "#0000ff"

saveimage "pie.jpg"
closeimage

opendoc "pie.jpg"
```

Listing 2.1 Pie script.

deliberately gave you a script with some faults, so that you can get some experience in correcting errors.

At this point, you should have a not-quite-perfect script that is crying out for correction.

Step 4: Revise the Script to Correct Errors

The first problem with the script, pointed out by the compiler, is that something is wrong with the first line.

```
draw a pie chart
```

This is clearly a comment, but it is missing the initial characters for a comment. In NQL, // or ; begins a comment. Correct the code in the editor by following these steps:

1. Click or use the arrow keys so that the text cursor is at the beginning of the first line of the script.
2. Enter // so that the first line of the script now reads as follows:

```
//draw a pie chart
```

Now run the script again, by pressing F5. A pie chart will open up on the desktop, as shown in Figure 2.7.

There's good news and bad news at this point. The compiler error on Line 1 no longer appears, which is good. But the pie chart is not painting the pie slices in colors, which is bad. We have another error to fix.

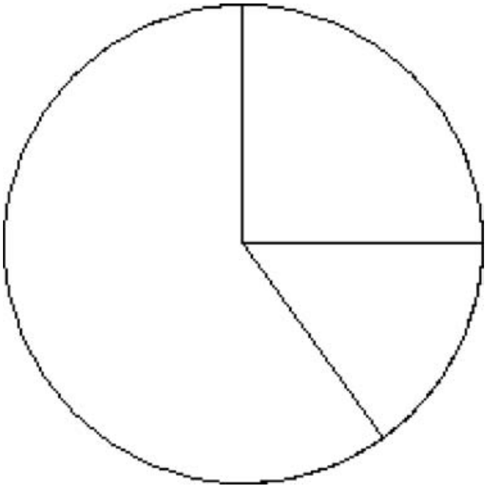


Figure 2.7 Faulty pie chart.

The problem here is a missing statement. We need to add a statement to the script that causes it to draw shapes that are filled, rather than hollow. To make this change, move to the blank line after *createimage* in the script, and add this line:

```
drawstyle "solid"
```

The complete script should now look like Listing 2.2.

Now, run the script once again. This time, the pie chart displays as expected.

```
//draw a pie chart

createimage 200, 200
drawstyle "solid"

drawpie 0, 0, 100
drawpieslice 25, "#ff0000"
drawpieslice 60, "#00ff00"
drawpieslice 15, "#0000ff"

saveimage "pie.jpg"
closeimage

opendoc "pie.jpg"
```

Listing 2.2 Corrected script.

At this point, you should have a properly working script that generates a three-color pie chart.

Step 5: Save the Script

Now that we have a working script, let's be sure to save it. Do so with these steps:

1. Select *File, Save* from the menu (or, click the *Save* toolbar button or press Ctrl+S).
2. In the *Save As* dialog window that appears, save the script as pie.nql in a disk directory (such as c:\).

At this point, your working script should be saved on disk as pie.nql.

Step 6: Generate a Desktop Agent

Lastly, we want to deploy the script on the desktop so it is easy to run at any time. We'll do this by generating an .exe file for the desktop. Here are the steps:

1. From the development environment, select *Tools, Make Agent EXE* from the menu. The *Make Agent EXE* dialog window appears.
2. If you wish, make changes to the options on the dialog window, for example to customize the appearance of the desktop agent.
3. Click OK.
4. A pie.exe file is created in the same directory where you saved pie.nql.
5. Create a desktop shortcut to the pie.exe file.

Now try running your agent by double-clicking its desktop icon. It should briefly appear as it runs, leaving a pie chart open on the desktop.

At this point, you should have successfully created and corrected an NQL script, created a desktop .exe file from it, and deployed it on the desktop.

Further Exercises

You could amend this example in a number of ways. Once you've learned more about NQL from the chapters that follow, you may want to come back to this example and try your hand at extending it. Here are some interesting modifications that can be made:

- Drive the pie chart with real data instead of fixed values. For example, the pie chart could reflect values read from a disk file or a database.
- Give the graphic that is generated a title and captions for the pie slices.
- Put the agent on a schedule, so that it runs at a pre-determined date and time.

Chapter Summary

The NQL Client is the development environment for Network Query Language. As an editor, it allows you to enter, revise, load, and save code as script files. You can also run and debug scripts. Online help is integrated into the development environment.

The debugger allows you to single-step through scripts, examining stack data, variables, and other indicators as a script proceeds.

The NQL client can generate .exe files from NQL scripts, allowing programs to be self-contained as .exe files with a desktop console appearance. The client can also generate .bot files from scripts, which are binary forms of scripts that avoid the need to distribute source code to end-user systems.

NQL scripts can be deployed in numerous ways, including manual execution, scheduled execution, installation on a server, part of a Web site, or as a component called by other programming languages.

You now know the basics of the development process. You can move forward with confidence, trying things out in NQL as you learn more about the language.

CHAPTER 3

Language Fundamentals

Like any other programming language, NQL has basic rules of the road. This chapter covers the rules of syntax and describes statements, functions, literals, variables, expressions, operators, and comments. There is also discussion of two central elements of NQL, the stack and the success/failure code. The tutorial at the end of the chapter obtains the current time from an atomic clock and sets your computer's time.

Statements

As in most languages, the fundamental unit of code in NQL is the statement. There are three kinds of statements in NQL: Standard statements, assignment statements, and function calls. Statements normally appear on a line by themselves. The following three-line script contains exactly three statements.

```
string Name
Name = "Gary Botka"
show Name
```

Standard statements are made by specifying keywords that are not case-sensitive. Some statement keywords are followed by parameters; if so, multiple parameters are separated by commas. Parameters can be terms (variables, literals, or function calls) or expressions combining terms and operators. The following statements are all standard statements, some of which have a parameter or two: Line 1 has no parameters, line 2 has one parameter, line 3 has three parameters, and line 4 has no parameters. In this case, all of the parameters are string literals.

```
openmail
messageattachment "pricelist.xls"
sendmessage "New Price List", "jim@my-company.com", "Latest price list."
closemail
```

Assignment statements evaluate an expression and assign the result to a variable. They are formulated as a variable name, an equals sign, and an expression. The following statements are examples of assignment statements.

```
x = 5
total = subtotal + tax + freight
FirstName = left(Name, Find(Name, " ")-1)
```

Function call statements are calls to functions you define. You can call functions within expressions, but you can also call a function on a line by itself as if it were a statement. The following statements don't exist in NQL natively, but if you had defined your own functions with these names, this is how you could call them. The first function, *ObtainSearchCriteria*, takes no parameters. The second function, *SubmitSearch*, takes two parameters.

```
ObtainSearchCriteria
SubmitSearch "Google", sTerms
SubmitSearch "Yahoo", sTerms
SubmitSearch "Excite", sTerms
SubmitSearch "AltaVista", sTerms
```

Continuation Lines

Like Visual Basic, NQL statements are one-per-line; as with C, C++, and Java, there is no terminating semicolon required at the end of each statement. What if a statement is too long to fit on a single line? If a statement ends in an operator or a comma, NQL realizes it should continue on to the next line. The following code shows several statements that continue on to multiple lines.

```
TimeString = sDayOfWeek & ", " & sMonthName & " " & sDayOfMonth &
             ", " & sYear & " " & sHours & ":" & sMinutes & ":"
y = x + x/1 + x/2 + x/3 + x/4 + x/5 +
        x/6 + x/7 + x/8 + x/9
show CurrentWeekSalesDomestic, CurrentWeekSalesForeign,
     CurrentMonthSalesDomestic, CurrentMonthSalesForiegn,
     CurrentQuarterSalesDomestic, CurrentQuarterSalesForeign
```

It is important to understand that none of the preceding three statements would be accepted by NQL if it were not for the dangling operator or comma at the end of each line. The first statement assembles a string, and the final character on the line is the concatenation operator (&), which cues NQL to include the next line as part of the statement. The second statement performs an arithmetic computation, and the final element of the line is the addition operator (+), which indicates that the next line is also part of the statement. The third statement is made up of three lines, which NQL realizes because the first two lines end in commas.

Multiple Statements per Line

Although NQL statements are normally phrased one-per-line, it is possible to combine them. For example, consider the following code, which you might wish to consolidate into fewer lines.

```
if sSort1 > sSort2
{
    sTemp = sName1
    sName1 = sName2
    sName2 = sTemp
}
```

The colon (:) allows multiple statements to be strung along on the same line. The preceding code can now be rewritten in a single line.

```
if sSort1 > sSort2 { sTemp = sName1 : sName1 = sName2 : sName2 = sTemp }
```

Code Blocks

Statements can be grouped into sections called code blocks, surrounded by curly brace characters ({ }), which are executed or skipped as a complete unit. The following code shows the use of a code block with an *if* statement. If the condition is true (that is, if *count* is greater than or equal to 500), the statements in the code block execute. If the condition is false, the entire code block is skipped.

```
if count >= 500
{
    show "The count has reached 500"
    close
    end
}
```

The *if*, *then*, *else*, *while*, *for*, *thread*, *switch*, and *every* statements all use code blocks. Code blocks can be nested, as the following code illustrates. Here, an *if* statement occurs within a *while* statement within a *for* block.

```
for f = 1, files
{
    open filename[f]
    while read(line)
    {
        if line != ""
        {
            show line
        }
    }
    close
}
```

One difference between C, C++, and Java is that code blocks are not optional. In the preceding program, you might be tempted to leave out the curly braces for the *show line* statement, since there is just one statement following the *if* statement. This is not legal, however. In NQL, statements like *if*, *for*, and *while* are *always* followed by code blocks.

Expressions

Any statement parameter may be an expression, and the right side of any assignment statement may be an expression. An expression is one or more terms, where a term is a variable, a number, a string literal, or a function call. Multiple terms in an expression are combined with operators. Chapter 5, "Expressions, Operators, and Functions," describes expressions and operators in detail. Each statement in the following code contains an expression.

```
OrderSubtotal = 0.00
OrderDate = date() & " " & time()
OrderSubtotal = OrderTotal+ItemPrice("PDC-001")
OrderSubtotal = OrderTotal+ItemPrice("PDC-300")
OrderSubtotal = OrderTotal+ItemPrice("PDC-018")
OrderTax = OrderSubtotal * TaxRate
OrderTotal = OrderSubtotal + OrderTax
CustomerCode = upper(left(CustID, 2))
```

Functions

One of the terms that may appear in an expression is a function. As mentioned earlier, functions can be viewed as statements or as part of an expression. Functions accept a parameter list (zero or more terms), perform a task, or return a result. NQL has about 80 built-in functions, and you may define your own custom functions. When calling a function in an expression, the function name is followed by parentheses. If there are parameters, they appear within the parentheses, comma-separated. The following code calls a *getpage* function to retrieve a Web page and a *clip* function to return a section of the page.

```
Page = getpage(URL)
Body = clip(Page, "<BODY", "</BODY>")
```

When you define your own functions, you can elect to return a result or not. If your function does not return a result, you cannot use it within expressions; however, you will be able to call the function on a line by itself as if it were a statement.

Comments

Comments allow you to add your own notes outside of the formal program code. There are three kinds of comments in NQL:

- End-of-line comments
- Comment lines
- Comment blocks

An *end-of-line comment* is a note to the right of a program statement. The double slash (//) characters or the semicolon (;) character begin the comment. The comment ends when the line ends. Each of the following statements has an end-of-line comment.

```
x = 0               // Clear the X-axis value.
y = 0               // Clear the Y-axis value.
z = 0               // Clear the Z-axis value.
```

A *comment line* begins with the double slash (//) characters or the semicolon (;) character. The comment ends when the line ends. Each of the following lines is a comment line.

```
; The following function extracts product results
; from the shopping Web site.
;
```

A comment block begins with the slash asterisk characters (/*) and ends with the asterisk slash (*/) characters. Like C, C++, and Java, comment blocks may span multiple lines. The following lines are a single comment block.

```
/* The following code has not been
formally tested yet.
Please exercise caution. */
```

Labels

Labels are bookmarks to locations in your code. You can jump to labels, call labels as subroutines, and trap errors to labels. You can also use them as a device for self-documenting code. You phrase a label by entering a name followed by a colon. A label can be on a line by itself, or it can precede a statement. The following code contains two labels, *GetUserInput* and *ErrorTrap*.

```
    errorhandler ErrorTrap
        ...
GetUserInput:
    Amount = prompt("Enter Amount", "What is the amount?")
    Rate = prompt("Enter Tax Rate", "What is the tax rate?")
    if Amount=="" or Rate<0.0
    {
        goto GetUserInput
    }
        ...
ErrorTrap:
    show "An error has occurred."
```

Literals

A literal is a value placed directly in the program. For example, in the statement *x = x + 5*, *5* is a numeric literal. Similarly, in the statement *name = "David"*, *"David"* is a string literal. NQL allows Boolean, integer, floating-point, and string literals.

Boolean literals are logical true and false values. The words *true* and *false* may appear in NQL scripts to indicate Boolean values.

Integer literals are whole numbers that consist of one or more digits (a leading minus sign indicates a negative number): 3, –16, 1054, and 0 are all integer literals.

Floating-point literals are real numbers that consist of one or more digits and include a decimal point (a leading minus sign indicates a negative number): 15.3, 10550.304, and –0.001 are all floating-point literals.

String literals are character strings. They are sequences of zero or more characters, and are enclosed in single quotation mark (‘) or double quotation mark (“) characters. “James Madison”, 'A', and “O’Reilly’s Bar & Grill” are all string literals.

The backslash character (\) has special meaning in string literals and is reserved for escape sequences that represent non-printing characters. Table 3.1 lists the string literal escape codes. To specify an actual backslash character, a double backslash (\\) is specified.

Unicode string literals are wide character strings where each character is made up of two bytes in the Unicode character set. Unicode string literals are formulated exactly like string literals, but are preceded with the letter U, as in U”Omaha.”

Variables

Variables are named storage areas that your program may use. Variable names may be any sequences of letters, digits, and special symbols but must begin with a letter. Symbols that have other meanings in the language, such as operators and curly braces,

Table 3.1 Escape Sequences

ESCAPE SEQUENCE	CHARACTER INDICATED	HEX CHARACTER CODE
\\	Backslash	5C
\'	Single quote	27
\"	Double quote	22
\b	Blank (space)	20
\n	Newline (line feed)	0A
\r	Carriage return	0D
\t	Horizontal tab	09
\z	Null	00

may not be used in variable names. *Total*, *x*, *nItems*, *sName*, *picture_width* and *GammaRayExposure* are all valid variable names.

An unusual feature of NQL is that you can force it to accept invalid variable names by enclosing the name in square brackets, as in *[odd+varname]*. The reason for this feature is that NQL must interface to a large number of technologies, including Web forms, XML documents, and databases, that differ in their rules for valid identifier names.

There are twelve data types in NQL, as listed:

Binary. A variable-length chunk of bytes.

Boolean. A logical true or false value.

Currency. A currency value, such as a U.S. dollar amount.

Datetime. A date and/or time value.

Float. A floating point (real) number.

Fuzzy. A fuzzy logic value (degree of truth between 0.0 and 1.0).

HTML. A string containing HyperText Markup Language.

Integer. An integer (whole) number.

Object. A pointer to an object.

String. A variable-length string of single-byte characters.

Unicode. A variable-length string of double-byte characters.

XML. A string containing eXtensible Markup Language.

NQL automatically converts between data types as needed. Although there are ways to explicitly cast one type to another, it is not usually necessary. For example, the following code assigns the result of a numeric computation to a string variable.

```
string result = (130000./7)
```

Declaration of variables in NQL is optional. NQL can be instructed to allow or disallow the use of variables that have not been declared. The *declare* statement causes NQL to require variables to be declared before they can be used.

```
declare
```

The *autodeclare* statement causes NQL to permit variables to be used even if they have not been formally declared. You can start using variables without declaring them, in which case they are automatically created as string variables. The default condition is *autodeclare*.

```
autodeclare
```

Many of NQL's statements and functions automatically create variables, such as pattern matching and database queries.

binary

Binary variables hold binary values (sequences of bytes). They are declared with the *binary* keyword, followed by one or more variable names. Their default value is empty.

```
binary header
binary NetworkPacket, SendBlock, ReceiveBlock
```

Any value assigned to a binary variable is stored byte-for-byte. The easiest way to put information into a binary variable is to take it off of the stack. You can obtain binary data from outside sources such as a file (using the *load* statement) or a Web page (using the *get* statement), then pop the contents into a binary variable.

bool, boolean

Boolean variables are logical values. There are two possible values for a Boolean variable, *true* and *false*. Define Boolean variables with the *bool* or *boolean* keyword, followed by one or more variable names. Their default value is false.

```
bool ForResale
boolean bHaveName, bHaveEmailAddress, bHavePhoneNumber
ForResale = true
```

Any value assigned to a Boolean variable is evaluated as a true/false logical value. NQL considers zero to mean false and any non-zero value to mean true. The constants *true* and *false* in NQL equate to 1 and 0 respectively.

currency

Currency variables hold monetary amounts. They are declared with the *currency* keyword, followed by one or more variable names. Their default value is 0.00.

```
currency amount
currency Credits, Debits
amount = 15.00
```

Any value assigned to a currency variable is evaluated as a real number.

datetime

Datetime variables hold dates and/or times. They are declared with the *datetime* keyword, followed by one or more variable names. Their default value is empty.

```
datetime Today
datetime CreationDate, ModificationDate, BackupDate
Today = "06/30/02"
```

Any value assigned to a *datetime* is evaluated as a date, date/time or time string.

float

Float variables hold floating-point (or real) numbers. They are declared with the *float* keyword, followed by one or more variable names. Their default value is 0.0.

```
float Ratio
float Subtotal, Tax Total
Ratio = 0.35
```

Any value assigned to a floating-point variable is evaluated as a real number.

fuzzy

Fuzzy variables hold fuzzy values, which are explained in Chapter 23, “Fuzzy Logic.” They are like floating-point variables, but their values are always in the range 0.0–0.1. They are declared with the *fuzzy* keyword, followed by one or more variable names. Their default value is 0.0.

```
fuzzy Tallness
fuzzy GoodPrice, GoodGenre, GoodPrice
Tallness = 0.83
```

Any value assigned to a fuzzy variable is stored as a floating-point number. If you attempt to assign a fuzzy variable a value less than 0.0, 0.0 will be stored. If you attempt to assign a fuzzy variable a value greater than 1.0, 1.0 will be stored.

html

HTML variables hold character string values that represent HyperText Markup Language. They are declared with the *html* keyword, followed by one or more variable names. Their default value is empty.

```
html Page
html search_page, results_page
Page = getpage(URL)
```

Any value assigned to an *html* variable is stored as a character string; HTML variables can be of any length, dynamically resizing when their contents change.

PLANNED FEATURE The roadmap for future NQL development includes plans to allow access to the particulars of an HTML variable using a dotted notation. This will permit a natural and easy syntax for extraction of HTML document elements within expressions.

int, integer

Integer variables hold whole-number values. They are declared with the *int* or *integer* keyword, followed by one or more variable names. Their default value is 0.

```
int x, y
integer nItems
nItems = 6
```

Any value assigned to an integer is interpreted as a whole number, which may be positive or negative.

object

Object variables hold pointers to objects and are used with systems like OLE to communicate with applications and components. They are declared with the *object* keyword, followed by one or more variable names. Their default value is null.

```
object pointer
object Excel, Worksheet
```

Any value assigned to an *object* variable is interpreted as a memory address (pointer) or handle. Object values are normally set or referenced by NQL's OLE Automation statements.

string

String variables hold character string values. They are declared with the *string* keyword, followed by one or more variable names. Their default value is empty.

```
string city
string First, Last, Middle
city = "Los Angeles"
```

Any value assigned to a *string* variable is stored as a character string. String variables can be of any length, dynamically resizing when their contents change.

unicode

Unicode values are strings with two-byte characters in the Unicode character set, which is capable of representing symbols from all the world's written languages.

```
unicode name
name = u"Schön"
```

Any value assigned to a *unicode* variable is stored as a character string. Unicode variables can be of any length, dynamically resizing when their contents change.

PLANNED FEATURE **Many of the NQL statements, functions, and operators work only on single-byte character strings. The intention over time is to support Unicode strings to the same extent as single-byte character strings.**

xml

XML variables hold character string values that represent eXtensible Markup Language. They are declared with the *xml* keyword, followed by one or more variable names. Their default value is empty.

```
xml invoice
xml WebServiceRequest, WebServiceResponse
```

Any value assigned to an *xml* variable is stored as a character string; XML variables can be of any length, dynamically resizing when their contents change.

PLANNED FEATURE **The roadmap for future NQL development includes plans to allow access to the particulars of an XML variable using a dotted notation. This will permit a natural and easy syntax for the extraction of XML elements within expressions.**

Arrays

Any variable type can be declared as an array containing multiple elements. When variables are declared, following the variable name with an element count in square brackets indicates that an array is to be allocated. The following code allocates an integer variable *x* (one instance) and an array named *y* (10 elements).

```
integer x, y[10]
```

Array handling varies between languages and can be difficult to remember. Depending on your prior experience, you might expect *x[10]* to range from elements 1 through 10, or perhaps 0 through 9. To avoid trouble NQL allocates one more element than the number specified, ranging from 0 to the specified number. Thus, in the preceding code there are in fact eleven elements allocated for *y*: *y[0]* through *y[10]* inclusive.

When referencing an array variable, any expression may appear within square brackets.

```
y[0] = 7
y[x] = -3
y[(a/2)+(b*d)-4] = 11
y[y[y[2]]] = 0
```

Arrays can be cleared to default values with the *clear* statement. They can be deallocated with the *erase* statement. When specifying an array for these statements, it is not necessary to include an element number.

```
clear y
```

Only one dimension of array is supported; multi-dimensional constructs such as *y[a][b]* are not permitted.

The Stack

A fundamental concept of NQL is its *stack*. Depending on your previous experience with programming languages, you may or not be familiar with the concept of a stack. The stack is a data storage area that resembles a stack of dinner plates on a table, where you can add more plates to the top of the stack or remove plates from the top of the stack. Adding a new value to the stack is called a *push* operation, and removing the top value is called a *pop* operation. The stack is a *last in, first out* mechanism.

Stacks are useful for many different reasons. They are not of fixed size, which makes them handy for recursive functions such as crawling a Web site or processing a list. They are also handy when temporary storage is needed, but you don't need to create a named variable. The stack can be used to exchange values between variables, such as when performing a sort.

Many of NQL's statements and functions add, remove, or modify values on the stack. Consider the following script, which retrieves a Web page from a news site and locates news articles on the page.

```
get "http://www.my-news-site.com"
match '<a href="{link}">{headline}</a>'
while
{
    output link, headline
    nextmatch
}
```

Here's what happens stack-wise as the script executes. The *get* statement on line 1 retrieves a Web page and pushes the HTML page onto the stack. The *match* statement on line 2 assumes there is an HTML page on the stack. If the *match* statement finds a news article, it sets values into the variables *link* and *headline*, and reduces the HTML page on the stack to only the portion that remains after the match. This allows the *nextmatch* statement at the bottom of the *while* loop to function properly. This simple example shows how different statements share information on the stack as a way of working together.

The two common NQL statements for directly getting at the stack are *push* and *pop*. The *push* statement adds another value to the stack. The *pop* statement removes a value from the stack; if you specify a variable after *pop*, the value is stored in that variable. There is also a function version of *pop*() that returns the value removed from the stack.

```
push "George"
push "Smithers"
push "Mr."
Salutation = pop()
pop LastName
pop FirstName
dumpvars
```

There are other stack-related statements and functions in NQL, which are covered in Chapter 5.

A Stack Example

Since the stack is such an essential part of NQL, let's study the stack in action to make sure we understand it. We'll be using the code in Listing 3.1, which computes and displays a subtotal of a variable number of items using the stack.

The first ten statements in the script are all *push* statements. They place a number of item names, quantities, and prices on the stack. Lastly, the number of items is also placed on the stack. By the time all of the *push* statements complete, the stack contains the ten items shown in Figure 3.1. The last item pushed, the number of items, is the top stack item.

The script next calls the *ComputeSubtotal* function, which declares some variables, initializes a subtotal to 0.00, and gets the number of items to process. This is done with the *pop* function, which returns (and removes) the top item from the stack.

```
    integer items = pop()
```

```
push "Television"
push 2
push 435.00

push "DVD player"
push 1
push 250.00

push "VCR"
push 1
push 175.00

push 3

tax = ComputeSubtotal()

show tax
end

currency ComputeSubtotal()
{
    currency amount
    integer qty
    string itemname
    currency subtotal = 0.00
    integer items = pop()
    for item = 1, items
    {
      pop amount
      pop qty
      pop itemname
      subtotal = subtotal + (qty*amount)
    }
    ComputeSubtotal = subtotal
}
```

Listing 3.1 Computation using the stack.

We now know the number of items to process—three—and the remaining data on the stack are three sets of product names, quantities, and amounts. A *for* loop can be used to process each item.

```
for item = 1, items
{
    pop amount
    pop qty
    pop itemname
```

```
        subtotal = subtotal + (qty*amount)
    }
```

For each item, the amount, quantity, and item name are popped off of the stack, in reverse order to how they were pushed. The amount is multiplied by the quantity and added to the accumulating subtotal. Let's imagine that this loop has just finished executing for the first item, with two items remaining to be processed. The stack now looks like Figure 3.2. There are six items remaining, because we have already removed the item count and the first item's amount, quantity, and name.

The *for* loop continues for two more iterations, returning the proper subtotal. The stack is now empty.

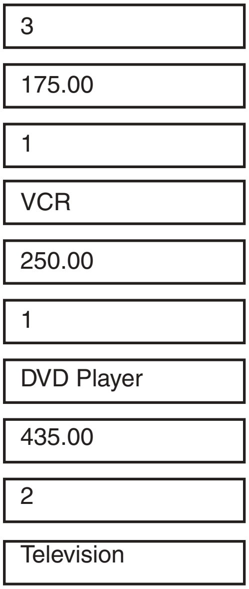


Figure 3.1 Stack with 10 items.

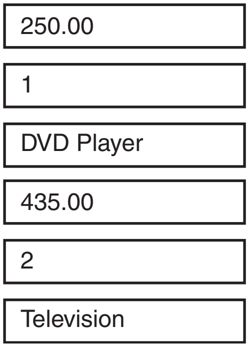


Figure 3.2 Stack with six items.

The stack is useful for recursion and for processing multiple items when you don't know in advance how many there will be.

The Success/Failure Condition

Another fundamental element of NQL is called the *success/failure condition*. Most (but not all) NQL statements indicate their success or failure by affecting this condition, allowing you to check for errors at junctures of your choosing. The following code shows a *get* statement that retrieves a Web page. If the retrieval fails, a failure condition results which can be checked, in this case with an *else* statement.

```
get URL
else
{
    show "The Web page could not be retrieved"
}
```

The success/failure condition is an idea borrowed from assembly language. Microprocessor instructions usually set conditions as they execute, which other instructions can test and act upon. The same idea is in play in NQL, where you can test the success of any action.

The *if*, *then*, *else*, and *while* statements all have forms that leave out the usual condition expression, using the success/failure condition instead. The following code shows the use of the success/failure condition to read through a full set of mail messages.

```
openmail
firstmessage
while
{
    ...do something with mail message...
    nextmessage
}
closemail
```

This script could not work without the success/failure condition. If *firstmessage* finds at least one mail message in the inbox, it sets the success/failure condition. This causes the *while* statement to enter the code block, which it would not do if there was a failure condition (no mail messages).

In the *while* loop, the message is processed. At the bottom of the loop is a *nextmessage* statement, which sets success if there is another message or failure if there are no further messages. In the case of a success condition, the *while* loop executes again.

The NQL documentation includes reference sheets for each statement and function that describe their effect on the success/failure condition. Some statements deliberately leave the success condition untouched, including *push*, *pop*, *show*, and assignment statements.

Error Handling

NQL provides several different kinds of error handling, because it can be used in many different environments. You may choose to handle errors differently in a desktop agent than you would in a Web application. The three types of error handling are *errors ignore*, *errors halt*, and *errors continue*.

In *errors ignore* mode, the script keeps running no matter what errors occur, regardless of how disastrous they are, and will not report any errors. The *errors ignore* statement enables this kind of error handling. You might choose to use this kind of error handling on a server, where it is imperative the script keep running for the next user under all circumstances.

In *errors halt* mode, the script halts immediately when an error occurs. The *errors halt* statement enables this kind of error handling. You might choose to use this kind of error handling when it is better to have the program stop rather than continue without having completed all actions successfully.

In *errors continue* mode, the script keeps running no matter what errors occur, regardless of how disastrous they are. Errors are reported, however, in the Error Log and Execution Log. The *errors continue* statement enables this kind of error handling. This is the most common type of error handling used; it allows you to check for errors at places in your script that are most convenient.

The default error handling method in NQL is *errors continue*. You can change the default in the NQL client (in the Run Script dialog), but that applies only to scripts run directly out of the development environment.

Directives

Some statements in NQL are not statements in the usual sense, but are directives to the compiler. That is, they don’t perform specific actions at runtime, but the compiler uses them.

autodeclare, declare

Two directives have already been discussed earlier in this chapter: *declare* and *autodeclare*, which determine whether or not variables must be declared before they can be used. The default state is *autodeclare*.

case matters, case ignore

The *case* directives control whether or not string comparisons take differences in case into account. The *case matters* directive is sensitive to differences in case. The *case ignore* directive ignores differences in case, and is the default setting.

include

The *include* directive causes the compiler to include another file (script) in your program. The *include* keyword is followed by one parameter, a filename.

```
include filespec
```

The file is included in your program by the compiler as part of the script. Include files may in turn also use the *include* statement to include other files. The use of include files allows you to keep one copy of code that is shared among multiple scripts, rather than having to maintain parallel sets of code.

Tutorial: Setting the Time

Now it is time to put the preceding information to work in an actual NQL program. In this tutorial, you will create a program that retrieves the correct time of day from the U.S. Naval Observatory Web site and sets your local PC's time accordingly. The script will illustrate many of the concepts we've learned about in this chapter, including the use of statements, comments, variables, literals, expressions, functions, and code blocks.

There are 4 steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the time script.
3. Run the time script.
4. Understand the time script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Declare variables.
- Phrase statements, literals, expressions, code blocks, and comments.
- Observe the cooperative use of the stack and the success/failure condition by NQL statements.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or command line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Time Script

In the NQL development environment, enter the script shown in Listing 3.2 and save it under the name time.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//time - set the time from the U.S. Naval Observatory Atomic Clock

//retrieve the time string from the data source

get "http://132.163.135.130:14/"
else
{
    show "Unable to access data source, cannot set the time"
    end
}

//alter the time string for optimal pattern matching

trim
pop sTime
sTime = '(' & sTime
push sTime

//parse the time string into its component pieces

match '({JulianDate} {Date} {Time} {DST} '
else
{
    show "Invalid or unexpected data, cannot set the time"
    end
}

//Convert the time to local time (PST).
//If daylight savings time, then subtract 7 from the hour to get our
time.
//Otherwise, subtract 8.

if DST > 0
{
    SubtractHours = 7
}
else
{
```

continues

Listing 3.2 Time script.

```
    SubtractHours = 8
}

//convert the hour to our local time (PST)

push Time
left 2
pop Hour
push Time
mid 3
pop Time

if Hour >= SubtractHours
{
    Hour = Hour - SubtractHours
}
else
{
    Hour = 24 - (SubtractHours - Hour)
}
Time = Hour & Time

//set the local computer time

settime Time

//acknowledge that the time has been set

show "The computer time has been set to " Time
```

Listing 3.2 Time script (continued).

If you prefer, you may copy time.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select time.nql from the Tutorials\ch03 folder.

At this point, the script in Listing 3.2 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Time Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the Run toolbar button, or pressing F5. If all goes well, a few seconds will pass and the script will display the message, *The computer time has been set to xx:xx*. Your computer's date and time will have been set accordingly. If this does not happen, check the following:

- Make sure you have entered the script correctly.
- Make sure you have a working Internet connection.
- Check the NQL client's Errors tab for error messages.

At this point you should have seen the time script run, obtaining an accurate time from an atomic clock and setting your computer's time to match. In the next step, we'll dissect the script and explain exactly how it works so that you appreciate how the various elements in NQL interacted to make this happen.

Step 4: Understand the Time Script

We now want to make sure we understand every part of the time script. The first line is a comment line.

```
//time - set the time from the U.S. Naval Observatory
```

The next few lines retrieve the time from the Naval Observatory's Web server. After a comment line, there is a *get* statement which accesses a Web address. If *get* is successful, a success condition results and the Web server response is pushed onto the stack. If *get* fails, a failure condition results and the *else* code block executes, displaying an error message and ending the script.

```
//retrieve the time string from the data source

get "http://132.163.135.130:14/"
else
{
    show "Unable to access data source, cannot set the time"
    end
}
```

The time string that comes back from the server is in an odd format. To make it easier to work with the time string, the next section of the script changes the time string a bit. Remember, the time string—the response from the Web server—is on the stack at this point. A *trim* statement removes leading or trailing spaces from the time string. It is then popped off of the stack into a variable named *sTime*. A dummy character is inserted at the beginning of *sTime*, and the resultant string is pushed back onto the stack. We still have just one value on the stack, a time string, but it is now in a normalized format that we can process easier.

```
//alter the time string for optimal pattern matching

trim
pop sTime
sTime = '(' & sTime
push sTime
```

Now we are ready to interpret the time string. A match statement divides the time string into its component pieces: A Julian date number, a date, the time of day, and a daylight savings time indicator. The *match* statement does this by comparing its pattern with the time string on the stack, setting variables and the success condition if a match is found. If *match* can't make sense of the data, a failure condition results and the script executes the *else* code block, displaying an error message and ending.

```
//parse the time string into its component pieces

match '({JulianDate} {Date} {Time} {DST} '
else
{
    show "Invalid or unexpected data, cannot set the time"
    end
}
```

The time needs to be converted to local time, which is assumed to be Pacific Standard Time in this script. The first step is to determine the number of hours to subtract from the reported time to come up with local time. The daylight savings time indicator is factored into the computation.

```
//Convert the time to local time (PST).
//If daylight savings time, then subtract 7 from the hour to get our time.
//Otherwise, subtract 8.

if DST > 0
{
    SubtractHours = 7
}
else
{
    SubtractHours = 8
}
```

Once the variable *SubtractHours* has been set, the hour needs to be separated from the rest of the time so it can be adjusted. This is done by pushing the time on the stack, keeping just the left two characters, and popping the result off as the variable *Hour*. The remainder is determined by pushing the time on the stack, discarding the first two characters, and popping the result as *Time*.

```
//convert the hour to our local time (PST)

push Time
left 2
pop Hour
push Time
mid 3
pop Time
```

We can now subtract *SubtractHours* from *Hour* to come up with the right hour of the day. With the adjusted hour, a complete local time can be computed, which is stored in *Time*.

```
if Hour >= SubtractHours
{
    Hour = Hour - SubtractHours
}
else
{
    Hour = 24 - (SubtractHours - Hour)
}
Time = Hour & Time
```

Now at last the local computer's time can be set. The *settime* statement sets the computer date and time from the time string.

```
//set the local computer time

settime Time
```

One final item of business remains. A *show* statement displays a message window, acknowledging that the time has been set.

```
//acknowledge that the time has been set

show "The computer time has been set to " Time
```

To be sure, there are shorter ways to write this script, but this version serves to illustrate how NQL statements interoperate in a sort of longhand way.

At this point, you've observed the elements of NQL working together to perform a task. You've seen statements, functions, variables, literals, the stack, and the success/condition code unite to retrieve data from a Web site, adjust it for local needs, and use the results.

Further Exercises

You could amend this example in a number of ways. Once you've learned more about NQL from the chapters that follow, you may want to come back to this example and try your hand at extending it. Here are some interesting modifications that can be made:

- Rewrite the time script to be much shorter. Use functions and expressions in place of statements where possible. See if you can get the script to less than half of its original size.
- Find a different time source than the one supplied and adapt the script to work with it. You'll have a different Web address to issue and a different response format to interpret.

Chapter Summary

There are three kinds of statements in NQL:

- Standard statements consists of a keyword followed by parameters.
- Assignment statements set a variable to the value of an expression and take the form variable = expression.
- Function calls allow user-defined procedures to be treated like statements. They can also be used within expressions.

Some statements are followed by code blocks.

- A code block is a group of statements enclosed in curly braces.
- The *if*, *then*, *else*, *while*, *for*, *thread*, *switch*, and *every* statements are all followed by code blocks.

Any statement's parameter may be an expression, and the right side of any assignment statement may be an expression.

- Expressions are one or more terms joined by operators.
- Terms may be literals, variables, or the results of functions. Functions in turn also take parameters which may themselves be expressions.

There are twelve data types in NQL for variables: *binary*, *boolean*, *currency*, *datetime*, *float*, *fuzzy*, *html*, *integer*, *object*, *string*, *unicode*, and *xml*.

- Variable declaration can be required or left optional through the use of the declare and autodeclare directives. Non-declared variables are always of type string.
- Variables of any data type can be declared as one-dimensional arrays.
- NQL automatically converts between data types as needed.

The stack is an integral part of NQL. The stack is a general-purpose *last-in, first-out* work area that can be used as a temporary workspace. The stack is very useful for recursive processing. Many NQL statements work together through the stack.

- The *push* and *pop* statements add data to the stack and remove data from the stack, respectively.

The success/failure condition is set by most NQL statements.

- The *if*, *then*, *else*, and *while* statements can respond to the success/failure condition.

There are three types of error handling: *errors ignore*, *errors halt*, and *errors continue*. The default is errors continue.

Scripts may include code from other scripts via the *include* directive.

You now know the fundamentals of NQL. Chapter 4, "Flow of Control," and Chapter 5 will build on that knowledge with explanations of how flow of control, expressions, operators, and functions work in NQL.

CHAPTER 4

Flow of Control

The control statements and structures of a language are your tools for expressing the flow of logic. Network Query Language contains the typical constructs for testing conditions, repeating code while certain conditions are true, looping, and selecting code based on a case value. The language also provides multi-threading capabilities for parallel processing. Other control-related features include spawning new processes and timing out execution after a specified interval. This chapter describes all of the language features that relate to flow of control and parallel execution. The tutorial at the end of the chapter uses multi-threading to check multiple search engines and find the fastest one at the moment.

Control Statements and Functions

The NQL statements related to flow of control are enumerated in Table 4.1.

There are function versions of many of the statements. The NQL flow-of-control functions are enumerated in Table 4.2.

Each of the statements and functions are described individually in the following sections.

Table 4.1 Control Statements

STATEMENT	PARAMETERS	DESCRIPTION
elapsedtime		Returns the script's (or thread's) elapsed time.
else		Executes a code block if there is a failure condition.
elseif	[*condition*]	Executes a code block if there is a failure condition and the specified condition is true.
end		Terminates the script.
every	*interval*	Repeats a code block at regular intervals
for	[*start,*] *end* [, *increment*]	Executes an iteration loop.
result-type function-name	(*parameter-list*)	Declares a function.
function-name	*parameter-list*	Calls a function as a statement.
goto	*label*	Jumps to the specified label.
if	*condition*	Executes a code block if the condition is true.
isthread	[*run-ID*]	Checks to see if a spawned thread is running.
isthreadrunning	[*run-ID*]	Checks to see if a thread is still active.
loop	*var*, *label*	Decrements a variable and, if >= 0, jumps to the specified label.
runid		Returns a thread's run ID.
share	*var-list*	Shares variables with spawned threads.
then		Executes a code block if there is a success condition.
thread	[*run-ID*]	Spawns a thread to execute the code block that follows.
threaderrors	[*run-ID*]	Returns the number of errors a thread has encountered.
threadstatus	[*run-ID*]	Returns a status phrase for a thread.

Table 4.1 Continued

STATEMENT	PARAMETERS	DESCRIPTION
timeout	*interval*	Sets a timeout interval for the script.
waitallthreads		Waits for all spawned threads to complete.
waitthread	[*run-ID*]	Waits for a specific thread to complete.
while		Executes a code block as long as there is a success condition.
while	*condition*	Executes a code block as long as the condition is true.

Table 4.2 Control Functions

FUNCTION	DESCRIPTION
datetime elapsedtime()	Returns the script's (or thread's) elapsed time.
type function invocation(*varies*)	Calls a user-defined function to return a value.
boolean isthread([*run-ID*])	Checks to see if a spawned thread is running.
string isthreadrunning([*run-ID*])	Checks to see if a thread is still active.
int runid()	returns a thread's run ID.
int threaderrors([*run-ID*])	Returns the number of errors a thread has encountered.
string threadstatus([*run-ID*])	Returns a status phrase for a thread.

If, Then, and Else Statements

Most languages have conditional action statements, as follows: if *condition*, then *action*. Furthermore, many also support the extended version: if *condition*, then *action-1*, else *action-2*. NQL supports these constructs with *if*, *then*, *else*, and *elseif* statements.

The *if* statement has two forms. The first form checks a condition and executes a code block if the condition is true.

```
if condition
{
     statement(s)
}
```

The following code example shows the *if* statement being used to display a result count only if the result count is greater than zero.

```
if ResultCount>0
{
     show ResultCount " new records were processed"
}
```

The second form of the *if* statement tests an expression and sets the NQL success/failure condition. This can be combined with other statements such as *then*, *else*, and *while* to control program flow.

```
if condition
```

The *then* statement executes a code block if there is a success condition. *Then* can be combined with *if* as well as any other NQL statement that sets the success/failure condition. The following code checks for new mail and executes a code block only if there is new mail.

```
openmail
firstunreadmessage
then
{
     ...do something with new mail...
}
closemail
```

The counterpart to *then* is *else*. *Else* can be combined with *if* and *then* for traditional if-then-else logic, as the following code demonstrates.

```
if count = 1
then
{
    show "1 item was located"
}
else
{
    show count " items were located"
}
```

It is interesting to note that the preceding code functions perfectly well even if the *then* statement is missing altogether. The reason for this is as follows. As written previously, the *if* statement sets only the success/failure condition because it is not immediately followed by a code block. The *then* statement checks the condition and decides whether or not to execute its code block. In the version of the code without a *then* statement, the program works this way: The *if* statement both tests a condition and decides whether or not to execute the code block that immediately follows it.

The fact that *then* and *else* are separate statements leads to interesting flexibility in programming. In addition to coupling if-then and if-then-else in the traditional way, statements like *then* and *else* can be used all by themselves as a way of testing the results of other statements. For example, the following code uses *else* to handle a communication failure when retrieving a Web page.

```
get WebAddress
else
{
    show "Unable to retrieve web page"
}
```

The *elseif* statement is, as its name implies, a combination of *else* and *if*. Like *else*, *elseif* executes a code block if there is a failure condition. Like *if*, *elseif* tests an expression that must evaluate to true in order to execute its code block. Both must be true in order for the code block to execute. The following code shows the use of *elseif*.

```
if x==3
{
    ...
}
elseif x==4
{
    ...
}
elseif x==5
{
    ...
}
else
{
    ..
}
```

Since *if*, *then*, *else*, and *elseif* are dependent on the success/failure condition, you may be concerned that statements within the code blocks could foil the expected execution. This does not occur because NQL saves and restores the state of the success/ failure condition when entering and leaving code blocks.

Loop Statements

There are five types of looping statements in NQL:

- The *while* loop
- The *for* loop
- The *every* loop

- The *goto* statements
- The *loop* statement

The types of loops are described in the following sections.

while

Like most modern languages, NQL includes a *while* statement whose purpose is to repeatedly execute a block of code as long as a specified condition is met. There are two syntax forms for *while.* The first is the form found in most other languages: A condition is specified, and the code block continues to execute as long as the condition is true.

```
while condition
{
    ...statement(s)...
}
```

The second form of *while* does not state a condition. Instead, the *while* loop cycles as long as the NQL success/failure condition is *success*. This form is often useful in NQL programming because it allows the result of the previous statement to be linked to the flow of program control.

```
while
{
    ...statement(s)...
}
```

The *while* loop may not execute even once, should the specified condition evaluate false from the start. It is also possible for a *while* loop to continue endlessly, should its specified condition remain true. More typically, a *while* loop executes a number of times until its condition is no longer met; execution then proceeds with the statement following the *while* block.

The following code sample uses a *while* loop to find repeatedly all occurrences of news stories from a Web page.

```
get URL
while match(Pattern)
{
     output link, headline
}
```

The following code sample accomplishes the same thing, but uses the second form of the *while* statement. In this case, it is very important that the final statement in the code block, *nextmatch*, set the success/failure condition.

```
get URL
match Pattern
while
```

```
{
    output link, headline
    nextmatch
}
```

You can exit a *while* loop prematurely with the *break* statement.

for

The NQL *for* loop resembles the iterative loop feature of many languages, with a starting value, a final value, and an increment. The form of *for* is shown here:

```
for [starting-value,] ending-value [, increment]
{
    ...statement(s)...
}
```

The starting value, ending value, and increment are all expressions that are evaluated numerically as integers. The following code uses a *for* loop to count to 10.

```
for i = 1, 10, 1
{
    show i
}
```

If only two parameters are specified in a *for* statement, the increment is presumed to be 1. If there is only one parameter, both the starting value and the increment are presumed to be 1. Thus, all three versions of *for* shown are functionally identical.

```
for pos = 1, Len, 1
for pos = 1, Len
for pos = Len
```

You can exit a *for* loop prematurely with the *break* statement.

every

To repeat a code block at regular intervals, the *every* statement is used. This parameter is a time string that specifies the interval, such as “1:00” for one minute.

```
every interval
{
    ...statement(s)...
}
```

The code block executes, then the script sleeps until the next execution interval. The sleep time subtracts the time it took the code block to execute from the specified interval, so that the code block’s repetition is precise and predictable.

The following code sample uses an *every* loop to check for new mail.

```
every "15:00"
{
    if firstunreadmessage()
    {
        ...alert user about new mail...
    }
}
```

Every loops never exit of their own accord, repeating endlessly. You can break out of an *every* block with the *break* statement.

goto

The *goto* statement transfers control to another point in the program. The parameter is a label name.

```
goto label
```

The following code uses *goto* to jump when a Web page cannot be accessed.

```
get URL
else
{
    goto CommFailure
}
    ...
end

CommFailure:
    show "Error: unable to access web site"
    end
```

loop

The *loop* statement decrements a variable and jumps to a label if the variable is greater than 0. It is very much like a decrement-and-branch-greater-than-zero machine instruction. The parameters are a variable and a label.

```
loop variable, label
```

The variable is decremented by one and, if still positive, execution jumps to the specified label. The following code shows the use of *loop*.

```
    items = 5
        ...
    ListProduct:
        ...
    loop items, ListProduct
```

You can achieve the same functionality as *loop* with the more conventional *for* statement.

Functions

Network Query Language supports user-defined functions as well as simple call-return subroutines. There is also a special kind of function known as a condition function that is used in fuzzy logic, described in Chapter 23, "Fuzzy Logic."

User-defined Functions

Functions perform a task, like statements, but they also return a result value, allowing them to be used in expressions. There are two kinds of functions in NQL: Built-in functions and user-defined functions. The built-in functions are described in Chapter 5, "Expressions, Operators, and Functions."

User-defined functions are expressed with a function declaration, which takes the following form:

```
result-type function-name(parameter-list)
{
    ...function-body...
}
```

The *result-type* is any NQL variable type (*binary*, *boolean*, *currency*, *datetime*, *float*, *fuzzy*, *html*, *int*, *object*, *string*, *unicode*, or *xml*), or the keyword *void* which indicates that the function does not return a result at all. The *parameter-list* is zero, one, or more parameters that the function uses. Each parameter specifies a type and a name, such as *int x*. If there are multiple parameters, they are separated by commas. The following function has three parameters.

```
float Total(float SubTotal, float TaxRate, float Shipping)
{
    Total = Subtotal + (SubTotal*TaxRate)+Shipping
}
```

The function result is indicated by setting the function name to a result value (*Total* in the preceding code).

User-defined functions can be invoked by treating them as statements or by using them in expressions. To invoke a function as a statement, use the function name as if it were a statement keyword, following it with any parameters. The following statement invokes the function *BackupDatabases* by treating it as a statement.

```
BackupDatabases BackupDate, BackupMedia
```

The same function could be used in an expression, but would require specification of parentheses around the parameters:

```
if BackupDatabases(BackupDate, BackupMedia)
```

To invoke user-defined functions in expressions, specify the function name and any parameters enclosed in parentheses (the parentheses must be present, even if there are no parameters). The function may be specified adjacent to operators, just as variables and constants are. The following assignment statement invokes two functions, *CustomerName* and *CustomerAddress*, using them within an expression.

```
ShipLabelText = CustomerName(CustNo) & "\n" & CustomerAddress(CustNo)
```

User-defined functions of type *void* cannot be used in expressions, because they do not return a result.

Subroutines

NQL has a simple subroutine capability similar to the call-return or gosub-return feature found in may versions of Basic. The *call* statement invokes a subroutine, specifying a label.

```
call label
```

The current location is remembered, and execution switches to the specified label. The code at the label executes a return statement to return control to the original point in the script.

```
return
```

The following code shows the use of *call* and *return*, and contains two subroutines.

```
    call GetName
    call MakeCustID
    show "{First}, your customer ID number is: {CustID}"
    end

GetName:
    first = prompt("First Name", "Please enter your first name")
    last = prompt("Last Name", "Please enter your last name")
    return

MakeCustID:
    CustID = left(upper(last), 4) & left(upper(first),1)
    return
```

For more sophisticated procedures, consider the use of user-defined functions rather than subroutines.

switch-case

Like C, C++, and Java, NQL contains a *switch* statement whose purpose is to branch to one of several code handlers based on a value. The *switch* statement has one parameter, a value to branch on, and is followed by a code block.

```
switch value
{
    ...sections...
}
```

Within the *switch* block, one or more sections must exist to handle different cases of the value. The sections begin with the word *case* or the word *default*. The *case* keyword is followed by a numeric constant or a string constant and a colon. The *default* keyword has no parameters and is followed by a colon.

```
case value1:
    ...
    break
case value2:
    ...
    break
        ...
case valueN:
    ...
    break
default:
    ...
    break
```

After a *case* line, specify statements to handle the value. End with a *break* statement, which is necessary to prevent falling through to the next case handler. The *default* line is just like *case*, but is a catch-all for values not already handled by an earlier *case* statement. The following code uses a *switch* statement to implement a simple power function.

```
switch power
{
    case 1:
        n = x
        break
    case 2:
        n = x*x
        break
    case 3:
        n = x*x*x
        break
    case 4:
        n = x*x*x*x
        break
    case 5:
        n = x*x*x*x*x
        break
    default:
        show "unsupported power"
        n = x
        break
}
```

The values specified in *case* statements may be numeric constants or string literals, but they cannot be variables or complex expressions. However, the value specified to *switch* itself may be any kind of expression.

break

The *break* statement exits a code block. Execution continues after the end of the next code block. You can specify an optional parameter to *break*, the number of levels of code block to break through.

```
break [levels]
```

The following code uses *break* to break out of two loops.

```
for i = 5000
{
    while i < j
    {
        ...
        break 2
    }
}
```

Parallel Processing

Many supposedly easy languages eventually let people down because they cannot perform operations in parallel. NQL includes full capabilities for performing tasks in multiple threads of execution. Modern connected applications, which are often distributed in nature, frequently require software to perform multiple tasks concurrently.

Spawning Processes

NQL scripts can spawn other NQL scripts to run as separate processes. The *run* statement spawns a script process and requires a script name as its first parameter. An input stream value and an output stream variable are optional.

```
run script-file [, input-stream [, output-stream]]
```

The *input-stream* parameter, if specified, is a string value that becomes the input stream for the spawned script. If the script you are spawning expects input, provide this parameter.

The *output-stream* parameter, if specified, is a variable that receives the output stream from the spawned script. If you want to capture the output of the spawned script, specify this parameter.

The following code sets up an input stream, runs another script, and captures its output stream.

```
string sInput = "<query>Barbecues</query>"
string sOutput
run "script2.nql", sInput, sOutput
show sOutput
```

The run statement waits for the spawned process to complete before your script continues with execution.

Multi-threading

NQL scripts can execute more than one code stream in parallel. This feature is known as multi-threading. Multi-threading can make a startling improvement in execution time. To appreciate the benefits of multi-threading, look at the following code that accesses a Web site five times in a row. The Web site will be accessed, then accessed again, and again, until it has been visited five times in all, sequentially.

```
for n = 5
{
    get URL
}
```

Figure 4.1 shows the possible timing of this script when accessing a Web site that generally responds in about 1 second. The total time to execute is the sum of each individual access, 4.6 seconds in all.

Now contrast the previous code with the version below. Each Web access is a thread block, which allows them to run in parallel without waiting for each other.

```
for n = 5
{
    thread
```

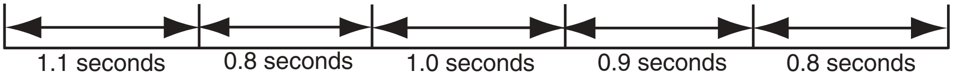


Figure 4.1 Single-thread execution.

```
        {
            get URL
        }
    }
```

In the multi-threaded version, the five Web sites are accessed in parallel. Figure 4.2 shows the possible timing of this script. The total time to execute is 1.1 seconds—less than one-fourth the time of the original version.

As useful as threads are, you need to be careful not to overdo it. Your operating system, system speed, and available memory place practical limits on the number of threads you can reasonably expect to use at one time.

Spawning Threads

A thread block is the language structure for indicating that code is to run in a separate thread from the main program. The format of a thread block is shown below.

```
thread [run-ID]
{
    ...statements...
}
```

A numeric run ID may be specified after the keyword *thread*, which can be used as an identifier later on to check or act on the thread. If the run ID is omitted, NQL will compute its own run ID for the thread.

A thread can determine its run ID with the *runid* statement or function. The *runid* statement pushes the thread ID on the stack. The function version of *runid* returns the thread ID as an integer.

```
runid
int runid()
```

The following statement retrieves and displays the current thread's run ID.

```
show "This thread's Run ID is " & runid()
```

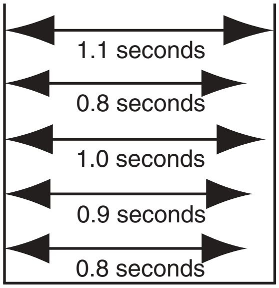


Figure 4.2 Multi-threaded execution.

You can generally expect run IDs to begin with 1 and increment by 1. The primary thread of a script usually has *run ID -1*. Note that this may not be true for all operating systems.

Terminating Threads

Left on their own, threads run to completion and are destroyed when they reach the end of their thread block. If the main thread of a program terminates, it will wait until all spawned threads have also completed before halting script execution.

NQL does not provide statements for terminating threads. Threads must complete of their own accord.

Sharing Variables

Threads normally run in a separate environment with their own variables and stack, but a parent thread may explicitly share some of its variables with a child thread. The *share* statement designates one or more variables to be shared between threads.

```
share var-list
```

The effect of *share* is that any future spawned threads receive copies of those variables from the parent thread. This sharing is in only one direction, however: The threads cannot alter the values seen by the parent. The following code sets variables, shares them, and then spawns a thread that makes use of them.

```
URL = "www.my-book-site.com"
FormData = "type=book&author=King"
share URL, FormData
thread
{
    post URL, FormData
    save "response.htm"
}
```

If the *share* statement had not been specified in this example, the thread block would not have worked correctly. It would have had its own variables *URL* and *FormData*, which would have been empty.

Merged Output

The output stream of threads are merged into the main script's output stream. This is the only direct mechanism whereby the main thread of a script can get results from a thread. The following code generates an output stream of *Alabama Arkansas Alaska California*.

```
output "Alabama "
thread
{
    sleep 2
    output "Alaska "
}
thread
{
    output "Arkansas "
}
waitallthreads
output "California "
```

Managing Threads

Code can check whether or not it is in a spawned thread using the *isthread* statement or function.

```
isthread
boolean isthread()
```

The statement form of *isthread* sets the success condition if a spawned thread is executing. The *isthread* function returns a Boolean value, true if a spawned thread is executing. *Isthread* is useful in user-defined functions, where you might want to include or skip certain actions depending on whether or not it was called by the main thread or a spawned thread. The following code calls the same function twice, once from the main thread and once from a spawned thread. The function uses *isthread* to display a message only if called from the main thread.

```
GetWebPage URL
thread
{
    GetWebPage URL
}
end

void GetWebPage(string URL)
{
    get URL
    else
    {
        if !isthread()
        {
            show "Unable to access web page"
        }
    }
}
```

Monitoring Threads

NQL provides five statements for monitoring threads or waiting for them to complete: *isthreadrunning*, *threadstatus*, *threaderrors*, *waitthread*, and *waitallthreads*.

isthreadrunning

You can check that a thread is running with the *isthreadrunning* statement or function, which takes a thread run ID as a parameter. If no run ID is specified, the last thread referenced is used.

```
isthreadrunning [ID]
boolean isthreadrunning([ID])
```

The statement form of *isthreadrunning* sets success if the thread is still active, or failure if it has completed and no longer exists. The function form of *isthreadrunning* returns a Boolean result, true if the thread is still active. The following code prints a report if a previously spawned thread has completed.

```
thread 5
{
    ....
}
    ....
if !isthreadrunning(5)
{
    printfile "report.lst"
}
    ....
```

threadstatus

An alternate way to check a thread's status is with the *threadstatus* statement or function, which returns the status string "running" or "completed."

```
threadstatus [run-ID]
string threadstatus([run-ID])
```

The statement form pushes the status string on the stack, while the function returns the status string as a value for use in expressions. Like *isrunning*, a thread run ID may be specified to *threadstatus*, but if omitted the last thread referenced is in view. The following statement obtains the status of thread 1 into a string variable.

```
CurrentStatus = threadstatus(1)
```

threaderrors

Another measure of thread status that may be checked is the number of errors that have occurred. The *threaderrors* statement or function returns this information as an integer.

```
threaderrors [run-ID]
int threaderrors([run-ID])
```

The statement form pushes the error count onto the stack. The function form returns it as an integer for use in expressions. A thread run ID may be specified to *threaderrors*, but if omitted the last thread referenced is in view. The following statement obtains the error count of the last thread spawned.

```
ThreadErrorCount = threaderrors()
```

waitthread

The *waitthread* statement waits for a thread to complete. *Waitthread* takes the following form:

```
waitthread [ID]
```

If no thread ID is specified, the last thread referenced in code is the thread in view. The following code spawns a child thread, then waits for it to complete before the main thread displays a message.

```
thread 11
{
    ftpexec "files1.ftp"
    ftpexec "files2.ftp"
    ftpexec "files3.ftp"
}
waitthread 11
show "The thread has completed"
```

waitallthreads

The *waitallthreads* statement waits for all spawned threads to complete.

```
waitallthreads [ID]
```

The following code spawns three threads, then uses *waitallthreads* to wait for all of them to complete.

```
thread
{
    ftpexec "files1.ftp"
}
thread
{
    ftpexec "files2.ftp"
}
thread
{
    ftpexec "files3.ftp"
}
waitallthreads
show "All file transfers have finished"
```

There is an implicit *waitallthreads* when a script ends: The primary thread of a script lingers until any and all spawned threads have also completed.

Timing Execution

The *elapsedtime* statement or function returns the elapsed time since the script started running.

```
elapsedtime
datetime elapsedtime()
```

If used in a thread block, the value is the time since the thread began executing, The function edition of *elapsedtime* returns a datetime value where the date portion is empty and the time portion contains the execution time in hours, minutes, and seconds. The statement version pushes the value onto the stack. The following code displays its elapsed time.

```
get URL1
get URL2
get URL3
show elapsedtime()
```

Terminating Execution

The *end* statement terminates script execution, after any still-running spawned threads have completed. Open files and sessions are closed automatically and their memory is released. An implied *end* statement is at the end of every NQL script regardless of whether one is explicitly in the code.

```
end
```

Execution can be terminated after a specified interval of time. The *timeout* statement establishes a time limit for the script. The parameter to *timeout* is a time value or string, such as “1:30:00” for one hour and 30 seconds.

```
timeout time
```

Once that time is exceeded, the script will end at the next opportunity. The following script crawls a Web site and collects files, but shuts down if more than 30 minutes have elapsed.

```
timeout "30:00"
download "www.my-tech-site1.com", 4, "pdf", "c:\\downloads"
download "www.my-tech-site2.com", 4, "pdf", "c:\\downloads"
download "www.my-tech-site3.com", 4, "pdf", "c:\\downloads"
download "www.my-tech-site4.com", 4, "pdf", "c:\\downloads"
```

When a script ends, anything open is closed and released. That includes files, databases, mail sessions, FTP sessions, Telnet sessions, and Web sessions.

Tutorial: FastSearch

Now it is time to put the preceding information to work in an actual NQL program. In this tutorial, you will create a program that visits several search engine Web sites in parallel, to see which one responds the fastest. The script will then open up a browser window to the winning site for the user. The script will illustrate the use of multithreading to perform multiple tasks in parallel.

There are 4 steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the FastSearch script.
3. Run the FastSearch script.
4. Understand the FastSearch script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Use thread blocks to perform multiple tasks in parallel.
- Parse the merged output of threads.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter a script.

Step 2: Enter the FastSearch Script

In the NQL development environment, enter the script shown in Listing 4.1 and save it under the name time.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy fastsearch.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial

```
//fastsearch - find the fastest search engine

thread
{
    url = "http://www.altavista.com"
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}

thread
{
    url = "http://www.excite.com"
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}

thread
{
    url = "http://www.google.com"
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}

thread
{
    url = "http://www.lycos.com”
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}

thread
{
    url = "http://www.yahoo.com"
```

continues

Listing 4.1 FastSearch script.

```
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}

waitallthreads

getoutput
setinput
input url
opendoc url
```

Listing 4.1 FastSearch script (continued).

materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select time.nql from the Tutorials\ch04 folder.

At this point, the script in Listing 4.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the FastSearch Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the Run toolbar button, or pressing F5. If all goes well, a few seconds will pass and a browser window will open up on your desktop, navigated to the search engine that happens to have responded soonest. If this does not happen, check the following:

- Make sure you have entered the script correctly.
- Make sure you have a working Internet connection.
- Make sure the sites listed in the script are online and available.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the FastSearch script run, accessing search engine sites in parallel to find the fastest one. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the FastSearch Script

We now want to make sure we understand every part of the FastSearch script. The first line is a comment line.

```
//fastsearch - find the fastest search engine
```

Next, we have a thread block, the first of five. Each thread block is identical except for the Web address to be accessed. Because thread blocks are in use, five worker threads are spawned in addition to the primary thread that has been executing. Each spawned thread goes to work right away, so all five are working in parallel.

```
thread
{
    url = "http://www.altavista.com"
    if get(url)
    {
        speed = elapsedtime()
        output url, speed
    }
}
```

In each of the thread blocks, the Web address to access is assigned to a variable, *url*. The *get* function is invoked to access the page.

```
url = "http://www.altavista.com"
if get(url)
{
    speed = elapsedtime()
    output url, speed
}
```

If the Web retrieval is successful, the code block after the *if* statement is executed. The speed of response is set to the thread's elapsed time. The Web address and the elapsed time are added to the output stream.

After all of the thread blocks have been started, our script needs to wait for them to complete. The *waitallthreads* statement does exactly that.

```
waitallthreads
```

Now we can choose the winner. Each of the thread blocks has output the URL and response time of their Web site if they were able to access it, in XML format. Since thread blocks share the same output stream as the main thread of the script, all of the URLs and response times have been written to the same output stream. If we can recapture that output stream, the first URL listed will be from the thread that finished first.

To get the first URL, we recapture the output stream (using *getoutput*) and set it to be the script's input stream (using *setinput*). That allows us to extract the *url* variable (using *input*).

```
getoutput
setinput
input url
```

All that remains is to open the URL on the desktop. The *opendoc* function does this, opening the URL in the default browser on the desktop.

```
opendoc url
```

At this point, you've witnessed parallel processing through the use of thread blocks, without which the script would have taken much longer to execute.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Find a way to avoid long waits for Web sites that are particularly slow to respond or are down (Hint: You might look into the timeout statement).
- Reduce the amount of redundant code in the example by adding a user-defined function that each of the thread blocks call. See if you can get the code in each thread block down to a single statement that calls the user-defined function.
- Hit each site more than once and tally the results, to make the response speed conclusions more definite.

Chapter Summary

Network Query Language has the usual collection of looping constructs, plus some unique ones.

- The *if*, *then*, *else*, and *elseif* statements behave like the statements of the same name in other languages. However, they are driven by the success/failure condition and can be coupled to the side effects of other NQL statements.
- The *while* statement implements looping on a condition. It can also loop on a success condition.
- The *for* statement provides iterative looping, incrementing a control variable.
- The *every* statement loops at regular intervals.
- The *goto* statement jumps to a label.
- The *loop* statement decrements a variable and jumps to a label if its value is greater than zero.
- The *switch* statement branches to a case handler based on the evaluation of an expression.

User-defined functions implement procedures and can return any of the twelve data types, or can return void (nothing). Functions may be used in expressions if they return a value, and can also be invoked as statements. Functions can take parameters, which are enclosed in parentheses and are separated by commas.

- Function declarations take the form result-type function-name (parameter-list).
- Function invocations at statement level take the form function-name parameter-list.

- Function invocations within expressions take the form function-name (parameter-list).
- In the code of a function, return values are specified by assigning a value to the function name.

Parallel processing can be achieved through the use of thread blocks. Thread blocks share a common output stream with the script's main thread.

- The thread statement declares a thread block.
- The isthread, isthreadrunning, runid, threadstatus, and threaderrors functions monitor thread activity or return information about threads.
- The waitthread and waitallthreads statements wait for one or all threads to complete executing.
- The share statement makes variables from the main script visible to thread blocks.

A script's execution time can be retrieved or controlled.

- The elapsedtime function returns the elapsed time of the script or thread block.
- The end statement ends a script.
- The *timeout* statement enforces a limit on the execution time of a script or thread block.

CHAPTER 5

Expressions, Operators, and Functions

Expressions allow variables, constants, and the results of functions to be combined in endless ways using arithmetic, string, and logical operators. This chapter describes the general rules for working with expressions, operators, and functions in Network Query Language. The casting, string, date/time, stack system, and URL functions are also described in this chapter. The tutorial at the end of the chapter uses expressions, operators, and functions to come up with alphabetic variations of a phone number.

Expressions

Like most programming languages, NQL allows you to combine variables and constants with operators. An *expression* consists of one or more terms, where a *term* is any of the following:

- A *constant*, such as 5, −16.42, or “Once upon a time”
- A *variable*, such as x, FirstName, or TaxRate[State]
- A *function*, such as Left(SSN, 3) or Time( )

If there is more than one term in an expression, they must be combined with operators. Operators are represented by symbols like + or words like *AND*.

Some valid terms are shown here:

```
x
3.1416
subtotal * tax_rate
LastName & ", " & FirstName
Trim(InputLine)
Date()+90
```

Expressions may appear as follows:

- In an assignment statement, to the right of the equals sign:

  ```
  f = ((9./5.)*c) + 32
  ```

- As a parameter for a statement:

  ```
  show subtotal+tax+shipping
  ```

- As a parameter for a function:

  ```
  FullSymbol = trim(Exchange & ":" & TickerSymbol)
  ```

Note that an array variable's subscript number and the parameters to a function are also expressions in their own right.

Operators

All operators modify terms: Binary operators combine two terms and unary operators modify a single term.

Binary Operators

The operators that work on two surrounding terms are called binary operators. If two adjacent terms occur in an expression without an operator between them, NQL assumes you want to concatenate them. Thus, the following two expressions are considered identical by NQL.

```
FirstName LastName
FirstName & LastName
```

The binary operators are listed in Table 5.1.

Unary Operators

Unary operators modify only the term that follows, as shown here:

```
-13
NOT Flag
```

Table 5.1 Binary Operators

OPERATOR	TYPE	DESCRIPTION
=	Assignment	Assigns the value of the term on the right to the term on the left, which must be a variable.
+	Arithmetic	The two terms are added.
-	Arithmetic	The term on the right is subtracted from the term on the left.
*	Arithmetic	The two terms are multiplied.
/	Arithmetic	The term on the left is divided by the term on the right.
==	Comparison	True results if the term on the left is equal to the term on the right.
!=	Comparison	True results if the term on the left is not equal to the term on the right.
<>	Comparison	True results if the term on the left is not equal to the term on the right.
<	Comparison	True results if the term on the left is less than the term on the right.
<=	Comparison	True results if the term on the left is less than or equal to the term on the right.
>	Comparison	True results if the term on the left is greater than the term on the right.
>=	Comparison	True results if the term on the left is greater than or equal to the term on the right.
&	String	The two terms are concatenated together.
|	String	The two terms are concatenated together.
LIKE	String	The term on the left is matched against the wildcard pattern on the right.
PAD	String	The term on the left is padded to the length of the term on the right.
AND	Boolean	The two terms are logically ANDed.
&&	Boolean	The two terms are logically ANDed.
OR	Boolean	The two terms are logically ORed.
||	Boolean	The two terms are logically ORed.
XOR	Boolean	The two terms are logically XORed.*

continues

Table 5.1 Continued

OPERATOR	TYPE	DESCRIPTION
^	Boolean	The two terms are logically XORed.*
FUZZYAND	Fuzzy	The two terms are fuzzy-ANDed.
^&	Boolean	The two terms are fuzzy-ANDed.
FUZZYOR	Fuzzy	The two terms are fuzzy-ORed.
^|	Boolean	The two terms are fuzzy-ORed.

Table 5.2 Unary Operators

OPERATOR	TYPE	DESCRIPTION
-	Arithmetic	Negates the number that follows
!	Logical	Negates the Boolean value that follows
NOT	Logical	Negates the Boolean value that follows

The unary operators are listed in Table 5.2.

The following code shows the use of the ! unary operator to negate a logical value.

```
if !get(url)
{
    show "communication failed"
}
```

Operator Precedence

There is a built-in hierarchy of operator precedence in NQL, as with most languages. Multiplication and division are evaluated before addition and subtraction, for instance. The complete operator hierarchy is listed in Table 5.3. The following assignment statement evaluates to 1.25 due to operator precedence.

```
x = 3. / 4. + 2. / 4.
```

The operator hierarchy may be circumvented through the use of parentheses. Parentheses control the order of term evaluation to your liking. The following statement evaluates to 0.6666, due to the use of parentheses.

```
x = 3. / (4. + 2. / 4.)
```

Operators at the same level are evaluated left to right. The following statement evaluates to 7.5 because of the left-to-right evaluation order.

Table 5.3 Operator Hierarchy

LEVEL	OPERATORS
1	not, !
2	*, /
3	+, -, and, &, or, |, xor, ^, pad, _
4	<, <=, >, >=
5	==, !=, <>
6	&&, ||, ^&, ^|, like, ~
7	= (assignment)

```
3. * 25 / 10
```

Let's take a detailed look at the operators.

The Arithmetic Operators

The arithmetic operators perform basic addition, subtraction, multiplication, and division operations.

+

The + operator adds two numbers together. The following code shows the use of + to produce a sum.

```
height = base_height + obelisk_height
```

If the arguments are non-numeric, they are converted to numeric form if possible, or are otherwise interpreted as 0.

-

The - operator subtracts the right term from the left term. The following code shows the use of - to produce a difference.

```
base_height = height - obelisk_height
```

If the arguments are non-numeric, they are converted to numeric form if possible, or are otherwise interpreted as 0.

*

The * operator multiplies two numbers together. The following code shows the use of * to produce a product.

```
area = depth * width * length
```

If the arguments are non-numeric, they are converted to numeric form if possible, or are otherwise interpreted as 0.

/

The / operator divides the term on the right into the term on the left. There is the possibility of a runtime error if the divisor is 0. The following code shows the use of / to produce a quotient.

```
height = base_height / obelisk_height
```

If the arguments are non-numeric, they are converted to numeric form if possible, or are otherwise interpreted as 0.

The specific action and result type of the arithmetic operators depend on the type of terms. Addition, subtraction, multiplication, or division of two integers produces an integer result. When some or all of the terms are floating-point in an operation, the result is floating-point. When string terms are used in arithmetic operations, the strings are interpreted as floating-point or integer values.

The Comparison Operators

The comparison operators return a Boolean result based on comparisons of two terms. The specific action and result type depend on the type of terms.

==

The == operator tests for equality, returning true if the terms are identical. The data types of the terms determine whether the equality test is numeric or string in nature. The following code shows the use of == to test for equality.

```
if x==13
{
    ...
}
```

The == operator will be familiar to C, C++, and Java developers, but if you're used to a language like Visual Basic you'll be tempted to use a single equals sign (=). The compiler will allow the use of = in *if* and *while* statements.

! =, < >

The != or <> operator tests for inequality. The data types of the terms determine whether the comparison is numeric or string in nature. The following code shows the use of != to test for inequality.

```
if k != 3
{
    ...
}
```

<

The < operator tests for a less-than condition, returning true if the term on the left is less than the term on the right. The data types of the terms determine whether the comparison is numeric or string in nature. The following code shows the use of < to test for a less-than condition.

```
if SortName1 < SortName2
{
    ...
}
```

<=

The <= operator tests for a less-than-or-equal-to condition, returning true if the term on the left is less than or equal to the term on the right. The data types of the terms determine whether the comparison is numeric or string in nature. The following code shows the use of <= to test for a less-than-or-equal-to condition.

```
if SortName1 <= SortName2
{
    ...
}
```

>

The > operator tests for a greater-than condition, returning true if the term on the left is greater than the term on the right. The data types of the terms determine whether the comparison is numeric or string in nature. The following code shows the use of > to test for a greater-than condition.

```
if SortName1 > SortName2
{
    ...
}
```

>=

The >= operator tests for a greater-than-or-equal-to condition, returning true if the term on the left is greater than or equal to the term on the right. The data types of the terms determine whether the comparison is numeric or string in nature. The following code shows the use of >= to test for a greater-than-or-equal-to condition.

```
if SortName1 >= SortName2
{
    ...
}
```

If the comparison operators are applied to strings, they will be sensitive to differences in case if the *case matters* directive is in effect. The comparison operators will ignore differences in case if the *case ignore* directive is in effect (the default condition).

The Logical Operators

The logical operators perform Boolean logic operations. The logical operators accept terms of any data type, but the result is always of type *boolean*. When string terms are used in logical operations, the strings are interpreted as numeric values, where 0 means false and any other value means true.

Some of the logical operators have more than one representation in order to make programmers from varying backgrounds feel at home. For example, the logical *and* function can be written as && or AND, allowing for the conventions of both C and Basic. Likewise, the *or* function can be written as `||` or OR.

and, &&

The *and* operation performs a logical AND function, yielding a true result only if both terms are true; otherwise the result is false. The following code shows the use of *and*.

```
if bHaveURL and get(sURL)
{
     bHaveWebPage = true
}
```

An alias for *and* is &&. The preceding code can also be written this way:

```
if bHaveURL && get(sURL)
{
     bHaveWebPage = true
}
```

The *and* operator has a different action if one or both terms are of type fuzzy: It behaves like the *fuzzyand* operator. For all other data types, the traditional Boolean logic applies.

or, ||

The *or* operation performs a logical OR function, yielding a true result if either term is true, or if both are true. Only if both terms are false is a false result returned. The following code shows the use of *or*.

```
if bFailed or bCanceled
{
    show "Cannot continue"
    end
}
```

An alias for *or* is `||`. The preceding code can also be written like this:

```
if bFailed || bCanceled
{
    show "Cannot continue"
    end
}
```

The *or* operator has a different action if one or both terms are of type fuzzy: It behaves like the FUZZYOR operator. For all other data types, the traditional Boolean logic applies.

xor, ^

An *xor* operation yields a true result if either term is true, but not both. If both terms are true or both terms are false, a false result is returned. The following code shows the use of *xor*.

```
if bMale xor bFemale
{
    bGenderSupplied = true
}
```

An alias for *xor* is ^. The preceding code can also be written this way:

```
if bMale ^ bFemale
{
    bGenderSupplied = true
}
```

The Fuzzy Operators

The fuzzy operators perform logical operations on fuzzy values.

fuzzyand, ^&

The *fuzzyand* operator returns the minimum of the two terms. An alternative notation for *fuzzyand* is ^&. The following code shows the use of *fuzzyand*.

```
Corpulence = Tallness FUZZYAND Girth
```

fuzzyor, ^|

The *fuzzyor* operator returns the maximum of the two terms. An alternative notation for *fuzzyor* is ^|. The following code shows the use of *fuzzyor*.

```
InsuranceRate = Age FUZZYOR Class
```

You can read more about fuzzy logic in Chapter 23, "Fuzzy Logic."

The String Operators

The primary string operator is concatenation. Most other string actions are performed by the string functions described later in this chapter. The concatenation operator is designated by & or |. The following two statements are identical in behavior:

```
fullName = firstName & middleName
fullName = firstName | middleName
```

When non-string terms are supplied to the concatenation operator, the terms are converted to string representations first. Thus, the expression in the following code is valid and evaluates to "abc3".

```
show "abc" & 3
```

In addition, there is implied concatenation: Two adjacent terms without an operator are automatically concatenated. The following code shows the use of implied concatenation.

```
fullName = firstName middleName
show "Your total bill comes to " amount ", which includes a gratuity."
```

Another string operator is *like*, which is similar to checking for string equality, but it allows asterisk wildcards to appear at the beginning and/or end of the second term. The following code shows the use of *like.*

```
if sName like "*Corp." { bCorporateName = true }
if sName like "*Inc." { bCorporateName = true }
if sName like "*Ltd." { bCorporateName = true }
```

One other string operator is *pad*, which pads the term on the left to the length specified by the right term. By default, the padding is with null characters (bytes containing 0), but this can be changed using the *padvalue* statement. The following code pads a name to a length of 30 characters, padding with spaces.

```
padvalue " "
string Name = "Debra"
PaddedName = Name pad 30
```

Functions

Network Query Language includes over 80 built-in functions that may be used in expressions. In addition, your own functions may be used in expressions. All functions are called by specifying a function name keyword and a pair of parentheses, as in *Trim(x)* or *Date()*. Some functions accept no parameters, others accept one parameter, and others accept multiple parameters. The parameters are specified within the pair of parentheses and are separated by commas, as in *Right(Name, 1)*. Function parameters are expressions themselves, as in *Right(Left(Text, 4), 1)*.

Built-in Functions

The NQL built-in functions cover a wide range of subjects, many of which are covered in other chapters of this book. The remainder of this section describes the casting functions, string functions, date/time functions, stack functions, and URL functions.

Casting Functions

The casting functions convert a value into a specific data type. The casting functions have the same names as variable types: int, float, string, and so on. Table 5.4 enumerates the casting functions.

String Functions

The string functions perform actions on string values and/or return string results. Most of the string functions parallel the common string functions found in languages like Basic, C++, and Java. Table 5.5 enumerates the string functions.

Each of the string functions is individually described in the following sections.

clip

The *clip* function returns a subsection of a main string by locating starting and ending text within the string. The parameters to *clip* are a main string, a starting substring, and an ending substring.

```
string clip(main-string, start-substring, end-substring)
```

Table 5.4 The Casting Functions

FUNCTION	DESCRIPTION
binary binary(value)	Converts a value to type *binary*
boolean bool(value)	Converts a value to type *boolean*
boolean boolean(value)	Converts a value to type *boolean*
currency currency(value)	Converts a value to type *currency*
datetime datetime(value)	Converts a value to type *date/time*
float float(value)	Converts a value to type *float*
fuzzy fuzzy(value)	Converts a value to type *fuzzy*
html html(value)	Converts a value to type *html*
int int(value)	Converts a value to type *int*
int integer(value)	Converts a value to type *int*
object object(value)	Converts a value to type *object*
string string(value)	Converts a value to type *string*
unicode unicode(value)	Converts a value to type *unicode*
xml xml(value)	Converts a value to type *xml*

Table 5.5 The String Functions

FUNCTION	DESCRIPTION
string clip(*value*, *start*, *end*)	Returns a substring of a string based on a starting and ending substring
boolean contains(*string*, *substring*)	Checks for the existence of a substring within a string
string convertnumber(*source*, *format*)	Converts between U.S. and European number formats
int count(*string*, *substring*)	Counts the occurrences of a substring in string
string eval(*value*)	Substitutes embedded variable names in a string with their current values
int find(*string*, *substring* [, *position*])	Locates the position of a substring in a string
string format(*specifier*, *value-list*)	Creates a formatted string
string insert(*string*, *position*, *substring*)	Inserts a substring with a string

Table 5.5 Continued

FUNCTION	DESCRIPTION
boolean isalpha(*string*)	Checks to see if a string is alphabetic
boolean isalphanumeric(*string*)	Checks to see if a string is alphanumeric
boolean isnumeric(*string*)	Checks to see if a string is numeric
string left(*string*, *length*)	Returns the left-most portion of a string
int length(*string*)	Returns the length of a string
string lower(*string*)	Returns the lower-case version of a string
string mid(*string*, *position* [, *length*])	Returns the middle section of a string
string normalize(*string*)	Normalizes a string
string remove(*string*, *substring*)	Removes all occurrences of a substring from a string
string removetags(*string*)	Removes all tags from a string
string replace(*string*, *oldstring*, *newstring*)	Replaces substrings within a string
string reverse(*string*)	Reverses a string
string right(*string*, *length*)	Returns the right-most portion of a string
int similarity(*string1*, *string2*)	Computes the similarity between two strings
int stringcompare(string1, string2 [,*case-sensitive*])	Compares two strings
string substring(*string*, *start-pos*, *end-p*os)	Returns a portion of a string
string trim(*string*)	Removes leading and trailing white space from a string
string upper(*string*)	Returns the upper-case version of a string
int val(*string*)	Converts a string to a number

The string returned is the section of the main string between the substrings, not including the substrings themselves. If either the starting or ending substrings are not found in the main string, an empty string is returned. The following code shows the use of *clip* for extracting the header and body sections of a Web page.

```
string sPage, sHeader, sBody
sPage = getpage(sURL)
```

```
sHeader = clip(sPage, "<head>", "</head>")
sBody = clip(sPage, "<body>", "</body>")
```

The *clip* function is useful for extracting a subset of a string.

contains

The *contains* function checks to see if a main string contains a substring or not. The parameters to *contains* are a main string and a substring.

```
boolean contains(string, substring)
```

The result is true if the substring occurs at least once within the main string. A false result indicates that the substring does not occur within the string. The following code shows the use of *contains* to determine whether a Web page contains any tables.

```
string sPage
sPage = getpage(sURL)
if contains(sPage, "<table")
{
    ...process table...
}
```

The *contains* function is useful for checking for expected content in text, HTML, and XML.

convertnumber

The *convertnumber* function creates a formatted string representation of a number in U.S. or European number format. The parameters are a value and a format string. The format string is “us” or “euro.”

```
string convertnumber(number, format)
```

The result is a string representation of the number with the appropriate decimal point and unit separator characters. In the U.S. format, numbers such as 3456.78 are formatted like 3,456.78 where a period indicates the decimal point position and a comma is a unit separator. In the European format, the result would be 3.456,78 with a comma indicating the decimal point position and a period as the unit separator. The following code converts a number to a string in European format.

```
currency Price = 1500.00
string sLine = "Der Preis ist DM " convertnumber(Price, "euro")
```

The *convertnumber* function is useful for placing unit separators in numbers, and for displaying the appropriate regional characters for decimal point and unit separators.

count

The *count* function counts the number of occurrences of a substring within a main string. The parameters are a main string and a substring.

```
int count(string, substring)
```

The return value is the count of substrings within the main string. If there are no occurrences of the substring, the result is 0. The following code uses count to determine the number of list items in a Web page.

```
string sPage
sPage = getpage(sURL)
int nTables = count(sPage, "<LI>")
```

The *count* function is useful for determining the number of elements in a string or a document.

eval

The *eval* function scans a string for any embedded variable names in curly braces, and replaces the references with the current values of the variables. The parameter is a string.

```
string eval(value)
```

The function returns a copy of the string with all references resolved. The following code displays the string *Good morning, Dave*.

```
TimeOfDay = "morning"
Name = "Dave"
Greeting = eval("Good {TimeOfDay}, {Name}")
show Greeting
```

The *eval* function is useful for creating formatted output.

find

The *find* function searches a string for a substring, and returns its position in the main string. The parameters are a main string, a substring, and an optional starting position which defaults to 1, the beginning of the string.

```
int find(string, substring [, position])
```

The main string is searched, starting at the specified position. If the substring is encountered, its starting position is returned, a number greater than or equal to 1. If the substring is not found, a 0 is returned. The following code uses *find* to extract a first name and a last name from a string in “[*last*], [*first*]” format.

```
int nPos = find(sName, ", ")
if nPos
{
     sLastName = left(sName, nPos-1)
     sFirstName = mid(sName, nPos+2)
```

```
        show sFirstName, sLastName
    }
```

The *find* function is useful for checking that strings contain expected values, and for finding the positions of those values.

format

The *format* function creates a formatting string, and is similar to the *format* or *printf* functions found in many other languages. The parameters are a format specifier string, and any number of additional arguments which provide values for the specifier.

```
string format(specifier, value-list)
```

The specifier is the formula for creating the formatting string. It contains text characters intermingled with format codes. Format codes begin with a percent sign and must be matched with an additional parameter in the function call. For example, the format specifier "I am %d years old" requires an additional value, the number to replace %d with. Table 5.6 lists the format codes that may be used within format specifiers.

The %c format code displays single characters based on an ANSI character code (an integer). The %d format code formats an integer number as a string. The %e and %f format codes display floating point, fuzzy, or currency values. The %g or %n format codes accept any kind of number (Boolean, integer, floating point, fuzzy, or currency) and automatically determine whether %d , %e, or %f should be used to display it. The %s format code accepts a string, HTML, or XML value.

Format codes may also include a size (field width). The following rules apply to sizes:

1. Sizes are specified between the percent sign and the type, as in %3d or %20s.
2. The size is the field width in characters, and in the case of %d, %e, %g, and %n may include a decimal point and the number of digits after the decimal point, as in %0.2f.
3. When size is omitted, the default is the most natural width required to show the data.
4. When a value takes less space than the specified size, leading spaces make up the field.
5. If the size begins with a leading zero, however, the extra space is made up with leading zeroes.

Format strings may also contain the escape sequences allowed in string constants, such as \b for space, \t for tab, and \n for newline. Table 5.7 shows some sample format specifiers and their output.

The following code uses *format* to create a simple report.

```
int NumberOfParts
string PartNumber[100], Desc[100]
currency RetailPrice[100], DiscountPrice[100]
string Line
    ...
```

Table 5.6 Format Codes

SPECIFIER	VALID TYPES	EXAMPLE	RETURNS
%c	character	format(“Grade: %c”, 65)	Grade: A
%d	integer number	format(“I am %d”, 13)	I am 13
%e	floating point number (scientific notation)	format(“%f”, 1.300)	1.30000
%f	floating point number	format(“%f”, 1.300)	1.30000e+000
%g	general number	format(“%g + %g = %g”, 3, 1.5, 3+1.5)	3 + 1.5 = 4.5
%n	general number	format(“%g + %g = %g”, 3, 1.5, 3+1.5)	3 + 1.5 = 4.5
%s	text string	format(“Hello, %s”, “Kim”)	Hello, Kim

Table 5.7 Sample Format Specifiers

SPECIFIER	VALUE	OUTPUT
%d	6	6
%0d	6	6
%3d	6	(2 spaces) 6
%03d	6	006
%f	17.5	17.5000
%6.2f	17.5	(1 space) 17.50
%0.3f	17.5	17.500
%n	6	6
%n	17.5	17.5
%s	John	John
%11s	John	John (7 spaces)
%c	65	A

```
        ...set values...
    ...
create "report.txt"
write "Part No.    Description    Retail Price    Discount Price\n"
write "--------    -----------    ------------    --------------\n"
for p = 1, NumberOfParts
{
    Line = format("%s\t%s\t%f\t%f\n", PartNumber[p], Desc[p],
        RetailPrice[p], DiscountPrice[p]
    write Line
}
close
```

The *format* function is useful for generating formatted output of all kinds.

insert

The *insert* function inserts a substring into a main string. The parameters are a main string, an insert position, and a substring to insert. An insert position of 0 inserts the substring before the main string, a position of 1 inserts after the first character of the main string, and so on.

```
string insert(string, position, substring)
```

The function returns the newly created string. The following code uses insert to convert a five-digit zip code into a nine-digit zip+4 code in a string.

```
sZip = "11779 is your zip code"
sPlusFour = "-0130"
sZip = insert(sZip, 5, sPlusFour)
```

The *insert* function is handy for expanding the contents of a string to include additional information.

isalpha

The *isalpha* function checks a string to see whether it is completely composed of alphabetic characters (letters). The parameter is a string value.

```
boolean isalpha(string)
```

If the string contains any non-letter characters, false is returned. Otherwise, the result is true. The following code checks that a U.S. state code consists of letter characters by using *isalpha*.

```
if !isalpha(State) or length(State)!=2
{
    show "Please enter a 2-digit state code"
}
```

The *isalpha* function is useful for validating input.

isalphanumeric

The *isalphanumeric* function checks a string to see whether it is completely composed of alphabetic and/or numeric characters (letters and digits). The parameter is a string value.

```
boolean isalphanumeric(string)
```

If the string contains anything besides letter and digit characters, false is returned. Otherwise, the result is true. The following code checks that a Canadian postal code consists of letter and digit characters by using *isalphanumeric*.

```
if !isalphanumeric(Postal) or length(Postal)!=6
{
    show "Please enter a 6-digit postal code"
}
```

The *isalphanumeric* function is useful for validating input.

isnumeric

The *isnumeric* function checks a string to see whether it is completely composed of numeric characters (digits). The parameter is a string value.

```
boolean isnumeric(string)
```

If the string contains anything besides digit characters, false is returned. Otherwise, the result is true. The following code checks that a U.S. zip code consists of digit characters by using *isnumeric*.

```
if !isnumeric(Zip) or length(Zip)!=5
{
    show "Please enter a 5-digit zip code"
}
```

The *isnumeric* function is useful for validating input.

left

The *left* function returns the left-most part of a string. The parameters are a main string and length.

```
string left(string, length)
```

The return value is the left part of the string of the specified size. The following code uses *left* to extract the first five digits of a zip+4 U.S. zip code.

```
sZip = left(sZipPlusFour, 5)
```

The *left* function is useful for extracting subsections of strings.

length

The *length* function returns the length of a string. The parameter is a string value.

```
int length(string)
```

The result is the number of characters in the string. The following code uses length to determine whether or not a U.S. state code is the proper length.

```
if length(sState)!=2
{
    show "Please enter a 2-letter state code"
}
```

The *length* function is useful for all kinds of string manipulation.

lower

The *lower* function converts a string to lowercase form. The parameter is a string value.

```
string lower(string)
```

The string returned is identical to the source string, except that all uppercase letters are transformed to lowercase. The following code uses *lower* to normalize a command to lowercase before attempting to execute it.

```
command = lower(command)
switch command
{
    case "open":
        ...
    case "close":
        ...
    case "send":
        ....
}
```

The *lower* function is useful for normalizing mixed-case strings prior to comparing them.

mid

The *mid* function returns the middle part of a string. The parameters are a main string, a starting position, and an optional length. If the length parameter is omitted, the full remainder of the string is returned from the starting position on.

```
string mid(string, position [, length])
```

The return value is the middle part of the string, beginning with the stated position (where 1 is the left-most position), of the specified length. The following code uses *mid* to extract the individual portions of a U.S. social security number.

```
ss = "091-62-1958"
ss1 = mid(ss, 1, 3)
ss2 = mid(ss, 5, 2)
ss3 = mid(ss, 8)
```

The *mid* function is useful for extracting subsections of strings.

normalize

The *normalize* function normalizes a string by removing newlines and redundant white space from it. The parameter is a string value.

```
string normalize(string)
```

The result is a copy of the string with newlines removed and spaces and tabs that are redundant. The following code uses *normalize* to normalize two strings before comparing them.

```
if normalize(string1)==normalize(string2)
{
    show "the strings have the same content"
}
```

The *normalize* function is useful for pre-processing of strings prior to comparing them or extracting data from them.

remove

The *remove* function removes all occurrences of a substring from a main string. The parameters are the main string and the substring to remove.

```
string remove(string, substring)
```

The result is a copy of the main string with the substring removed. The following code uses *remove* to remove titles from a name.

```
sName = remove(sName, "Mr. ")
sName = remove(sName, "Mrs. ")
sName = remove(sName, "Dr. ")
```

The *remove* function is useful for removing unwanted substrings from a string.

removetags

The *removetags* function removes tags from markup text, such as HTML or XML, leaving text untouched. The parameter is a string containing tags and text.

```
string removetags(string)
```

The result is a copy of the string with all tags removed, where a tag is any sequence of characters bounded by angle brackets, such as <*TABLE*>. The following code uses *removetags* to de-tag HTML.

```
sText = removetags(sHTML)
```

The *removetags* function is useful for extracting text from HTML Web pages and other markup language documents.

replace

The *replace* function replaces all occurrences of a substring in a string with a different substring. The parameters are a main string, a search substring, and a replacement substring.

```
string replace(string, find-string, replace-string)
```

The result is a copy of the main string, with all occurrences of the *find* string replaced by the *replace* string. The following code uses *replace* to expand abbreviations in a string.

```
sText = replace(sText, "Inc.", "Incorporated")
sText = replace(sText, "Corp.", "Corporation")
sText = replace(sText, "Co.", "Company")
sText = replace(sText, "Ltd.", "Limited")
```

The *replace* function is useful for all kinds of string manipulation.

reverse

The *reverse* function reverses a string so that its characters appear in opposite order. The parameter is a string value.

```
string reverse(string)
```

The result is a reversed copy of the original string. The following code uses *reverse* to reverse a string.

```
sOptions = "YY10Y"
sFlags = reverse(sFlags)
```

The *reverse* function is useful for some kinds of string manipulation operations, including encryption.

right

The *right* function returns the right-most part of a string. The parameters are a main string and length.

```
string right(string, length)
```

The return value is the right part of the string of the specified size. The following code uses *right* to extract the last four digits of a zip+4 U.S. zip code.

```
sPlusFour = right(sZip, 4)
```

The *right* function is useful for extracting subsections of strings.

similarity

The *similarity* function determines the similarity of two strings. The parameters are two strings to be compared.

```
int similarity(string1, string2)
```

The result is an integer between 0 and 100 representing the percentage of similarity between the two strings. The similarity is computed as follows: The number of unique words in the first string is divided into the total number of unique words from both strings. The following code uses *similarity* to determine the similarity of two weather forecast sentences, which in this case have a similarity of 60.

```
string s1 = "The forecast calls for rain and thunderstorms."
string s2 = "Rain and thunderstorms are called for in the forecast."
show similarity(s1, s2)
```

The *similarity* function is useful in situations where you must match up similar but not necessarily identical values, such as user input of a name with minor misspellings.

stringcompare

The *stringcompare* function compares two strings. The parameters are two strings to be compared, and an optional case sensitivity flag which defaults to true (differences in case matter). A value of false ignores differences in case.

```
int stringcompare(string1, string2 [,case-sensitive])
```

The result is -1, 0, or 1. A 0 result indicates that the two strings are identical. A -1 result indicates that string1 is less than string 2. A 1 result indicates that *string2* is greater than *string1*. The following code uses *stringcompare* to check two strings for equality without regard for case.

```
if stringcompare(sCurrName, sSpecifiedName, false)==0
{
    show "names are identical"
}
```

The *stringcompare* function is useful for general string comparisons as well as sorting and indexing operations.

substring

The *substring* function returns a substring from a main string. The parameters are a main string, a start position, and an end position. Positions are 0-based.

```
string substring(string, start-pos, end-pos)
```

The result is the *substring* starting at the starting position (where 0 is the first character in the main string) and ending at the ending position. The following code uses substring to extract the three parts of a U.S. social security number.

```
ss = "091-62-1958"
ss1 = substring(ss, 0, 2)
ss2 = substring(ss, 4, 5)
ss3 = substring(ss, 7, 10)
```

The *substring* function is a useful alternative to the *left*, *mid*, and *right* string functions, which use 1-based string positions. If you are more comfortable with 0-based string positions, *substring* will probably be easier to work with.

trim

The *trim* function removes leading and trailing white space from a string. The parameter is a string value.

```
string trim(string)
```

The result is a copy of the string value where all leading and trailing spaces, tabs, newlines, and carriage returns have been removed. The following code uses *trim* to remove white space from a string.

```
input name
name = trim(name)
```

The *trim* function is useful for removing spurious white space from strings.

upper

The *upper* function converts a string to uppercase form. The parameter is a string value.

```
string upper(string)
```

The string returned is identical to the source string, except that all lowercase letters are transformed to uppercase. The following code uses *upper* to normalize a command to uppercase before attempting to execute it.

```
command = upper(command)
switch command
{
    case "OPEN":
        ...
    case "CLOSE":
        ...
    case "SEND":
        ....
}
```

The *upper* function is useful for normalizing mixed-case strings prior to comparing them.

val

The *val* function converts a string value into an integer. The parameter is a string.

```
int val(string)
```

The result is an integer value. The following code uses *val* to transform in input string into an integer.

```
int nValue = val(sInput)
```

The *val* function is useful for converting strings to integers. The casting operators may also be used to convert strings to numeric types.

Date/Time Functions

The date/time functions perform actions on date and time values and/or return date/time results. Table 5.8 enumerates the date/time functions.

The date/time functions are described individually in the following sections.

convertdate

The *convertdate* function converts a date to one of three formats: U.S., European, or universal. The parameters are a date value (or string), a source format, and a target format. The source format and target format parameters may be *us*, *euro*, or *universal*.

```
string convertdate(source, sourceformat, targetformat)
```

The result is a string representation of the date in the specified target format. A *us* format date is mm/dd/yy. A *euro* format date is dd/mm/yy. A *universal* format date is yyyy-mm-dd. The following code converts a U.S. date to European format.

Table 5.8 The Date/Time Functions

FUNCTION	DESCRIPTION
string convertdate(*source*, *sourceformat*, *targetformat*)	Converts a date to a different regional format
datetime date( )	Returns the current system date
datetime datevalue(*datetime*)	Returns just the date portion of a datetime value
int datetojulian(*date*)	Converts a date to a Julian number
datetime juliantodate(*number*)	Converts a Julian number to a date
boolean parsedate(*value*, *month*, *day*, *year*, [[*day-of-week*,] *day-of-year*])	Parses a date
boolean parsedatetime(*value*, *month*, *day*, *year*, *hour*, *minutes*, *seconds*)	Parses a date/time
boolean parsetime(*value*, *hour*, *minutes*, *seconds*)	Parses a time
datetime time( )	Returns the current system time
datetime timevalue(*value*)	Returns just the time portion of a datetime value

```
string sEuroDate = convertdate("12/31/01", "us", "euro")
```

The statement form of *convertdate* converts a date on the stack to a different format.

date

The *date* function returns the current system date. The function takes no parameters.

```
datetime date()
```

The function returns the current system date as a date/time value. The following code uses *date* to retrieve the system date.

```
datetime Today = date()
```

The *date* function is useful for determining today's date. The statement form of *date* places the current date on the stack.

datetojulian

The *datetojulian* function converts a date to a Julian number. The parameter is a date.

```
int datetojulian(date)
```

The result is a Julian number. A value of 1 is calibrated to January 1, 4713, B.C. The following code uses *datetojulian* to see if two dates are more than 90 days apart.

```
boolean bOverdue = false
datetime dDueDate = "05/04/02"
datetime dPayDate = "06/19/02"
int nDueDate = datetojulian(dDueDate)
int nPayDate = datetojulian(dPayDate)
if (nPayDate >= nDueDate+90)
{
    bOverdue = true
}
```

The *datetojulian* function is useful for date comparisons in standard units such as weeks, months, and quarters. To convert a Julian number back into a date, use *juliantodate*. The statement form of *datetojulian* replaces a date on the stack with an equivalent Julian number.

datevalue

The *datevalue* function returns the date portion of a date/time value. The parameter is a date/time value.

```
datetime datevalue(datetime)
```

The result is a date/time value with the date part set and the time part empty. The following code uses *datevalue* and *timevalue* to separate the date and time portions of a date/time value.

```
datetime dFull = "12/31/02 4:59 pm"
datetime dDate = datevalue(dFull)
datetime dTime = timevalue(dFull)
```

The *datevalue* function is useful for focusing on the date portion of a date/time value.

juliantodate

The *juliantodate* function converts a Julian number to a date. The parameter is a Julian date number.

```
datetime juliantodate(number)
```

The result is a date. A Julian number of 1 is calibrated to January 1, 4713, B.C. The following code uses *juliantodate* to compute a due date.

```
string DueDate = juliantodate(datetojulian(date())+30)
```

The *juliantodate* function is useful for date computations. To convert a date back into a Julian number, use *datetojulian*. The statement form of *juliantodate* replaces a Julian number on the stack with an equivalent date.

parsedate

The *parsedate* function accepts a string value and parses a date from it. The parameters are a string value and variables to receive the month, day, and year. In addition, variables may also be specified to receive the day-of-week and day-of-year.

```
boolean parsedate(value, month, day, year, [[day-of-week,] day-of-year])
```

The result is true if the date is valid, or false if the date is invalid. If the date is valid, the month, day, and year variables are set. If a day-of-year variable was specified, it receives the day of the year (where January 1 is 1). If a day-of-week variable was specified, it receives the day of the week (where Sunday is 1). Using *parsedate,* the following code determines whether today's date falls on a weekend.

```
boolean bWeekend = false
if parsedate(date(), m, d, y, dw, dy)
{
    if (dw==1 or dw==7) bWeekend = true
}
```

The *parsedate* function is useful for separating a date into its individual components. The statement form of *parsedate* takes the date value from the stack.

parsedatetime

The *parsedatetime* function accepts a string value and parses a date and time from it. The parameters are a string value and variables to receive the month, day, year, hours, minutes, and seconds.

```
boolean parsedatetime(value, month, day, year, hours, minutes, seconds)
```

The result is true if the value contains a valid date and/or time, or false if not. If the value contains a valid date, the month, day, and year variables are set. If the value contains a valid time, the hours, minutes and seconds variables are set. The following code determines the elements of today's date and time by using *parsedatetime*.

```
string sNow = date() & " " & time()
if parsedatetime(sNow, month, day, year, hours, minutes, seconds)
{
    dumpvars
}
```

The *parsedatetime* function is useful for separating a date/time value into its individual components. The statement form of *parsedatetime* takes the date/time value from the stack.

parsetime

The *parsetime* function accepts a string value and parses a time from it. The parameters are a string value and variables to receive the hours, minutes, and seconds.

```
boolean parsetime(value, hours, minutes, seconds)
```

The result is true if the time is valid, or false if the time is invalid. If the time is valid, the hours, minutes, and seconds variables are set. The following code determines whether the current time is during business hours.

```
int hours, minutes, seconds
if parsetime(time(), hours, minutes, seconds)
{
    dumpvars
    if hours >= 8 and hours <= 17
    {
        show "during business hours"
    }
    else
    {
        show "outside business hours"
    }
}
```

The *parsetime* function is useful for separating a time into its individual components. The statement form of *parsetime* takes the time value from the stack.

time

The *time* function returns the current system time. The function takes no parameters.

```
datetime time()
```

The function returns the current system time as a date/time value. The following code uses *time* to retrieve the system time.

```
datetime Now = time()
```

The *time* function is useful for determining the current time. The statement form of *time* places the current time on the stack.

timevalue

The *timevalue* function returns the time portion of a datetime value. The parameter is a date/time value.

```
datetime timevalue(datetime)
```

The result is a date/time value with the time part set and the date part empty. The following code uses *datevalue* and *timevalue* to separate the date and time portions of a date/time value.

```
datetime dFull = "12/31/02 4:59 pm"
datetime dDate = datevalue(dFull)
datetime dTime = timevalue(dFull)
```

The *timevalue* function is useful for focusing on the time portion of a date/time value.

Stack functions

The stack functions and statements push, pop, or otherwise manipulate stack items. Table 5.9 lists the stack functions.

Table 5.9 The Stack Functions

FUNCTION	DESCRIPTION
int checksum(value)	Computes a checksum for a string
boolean empty( )	Checks for a stack-empty condition
string merge( )	Merges two stack items
boolean nextline([*var*])	Returns the next line from the string on the stack
string pop( )	Pops the top item off of the stack
string store( )	Retrieves the top stack item without popping it

Table 5.10 The Stack Statements

STATEMENT	PARAMETERS	DESCRIPTION
binary		Converts the top stack item to the *binary* data type
checksum		Computes a checksum for the top item on the stack
clear		Empties the stack
exchange		Exchanges the top two items on the stack
dump		Displays the contents of the stack in a window
dupe		Copies the top item on the stack
empty		Checks for a stack-empty condition
merge		Merges two stack items
nextline	[*var*]	Separates the top line from the rest of the string on the stack
pop	[*var*]	Pops the top item off of the stack
push	*value*	Pushes a value onto the stack
store	*var*	Retrieves one or more stack values into variables
text		Converts the top stack item to text (*string* data type)

Table 5.10 lists the stack statements.

The stack functions and statements are described individually in the following sections.

binary

The *binary* statement converts the top stack item into the *binary* data type.

```
binary
```

If the stack data is any data type other than *binary*, it is converted into the *binary* data type. The *binary* statement is useful for insuring that stack data is in binary form.

checksum

The *checksum* function computes a checksum. The parameter is a string value.

```
int checksum(value)
```

The result is an integer checksum. The following code uses *checksum* to compute a verification value for a string.

```
int CRC = checksum(sData)
```

The statement form of *checksum* computes a checksum for the top value on the stack. The checksum is pushed onto the stack.

```
checksum
```

Checksums are useful for validating the integrity of data after it has been transmitted from one storage location to another.

clear

The *clear* statement empties the stack.

```
clear
```

The *clear* statement is useful for discharging all previous stack contents.

exchange

The *exchange* statement exchanges the top two items on the stack.

```
exchange
```

The top stack item and the item beneath it are swapped. The following code uses *exchange* to swap items on the stack so that they are in ascending order.

```
push string1
push string2
if string1 > string2
{
    exchange
}
```

The *exchange* statement is useful for rearranging stack items into a more convenient arrangement and for sorting.

dump

The *dump* statement displays the contents of the stack in a window. There are no parameters.

```
dump
```

The *dump* statement is useful for debugging, where you want to verify that the stack contains what you expect.

dupe

The *dupe* statement copies the top stack item. There are no parameters.

```
dupe
```

The top stack item is duplicated on the stack. The following code uses *dupe* to save a Web page in both HTML and text forms.

```
get url
dupe
removetags
pop sText
pop sHTML
```

The *dupe* statement is useful for copying stack items that need to be processed multiple times.

empty

The *empty* function checks for a stack-empty condition. There are no parameters.

```
boolean empty()
```

The result is true if the stack is empty. The following code uses *empty* to process all items on the stack.

```
while !empty()
{
    pop x
}
```

The statement form of *empty* sets the success condition if the stack is empty.

```
empty
```

The *empty* function or statement is useful for testing for a stack-empty condition.

merge

The *merge* function pops the top two items off the stack. There are no parameters.

```
string merge()
```

The result is the two stack items, concatenated. The following code uses *merge* to combine two stack items.

```
push string1
push string2
string3 = merge()
```

The statement form of *merge* merges the top two stack items into a single stack item.

```
merge
```

The *merge* function or statement is useful for consolidating stack data.

nextline

The *nextline* function extracts the top line from the top stack item. The parameter is an optional variable name.

```
boolean nextline([var])
```

The top line is removed from the top item on the stack. The extracted line is stored in the specified variable. If no variable is specified, the top line is pushed onto the stack. The following code uses *nextline* to process each line of a text file.

```
load sFilespec
while nextline(sLine)
{
    ...
}
```

The statement form of *nextline* separates the top line of the top stack item and pushes it onto the stack.

```
nextline
```

The *nextline* function or statement is useful for processing lists and line-oriented data. Many of NQL's statements and functions return information in the form of a line-oriented string pushed onto the stack.

pop

The *pop* function pops the top item off of the stack. There are no parameters.

```
string pop()
```

The result is the top stack item that was popped off. The statement form of *pop* pops the top stack item. A variable may be specified to receive the data.

```
pop [var]
```

The following code uses *pop* to retrieve a Web page off of the stack.

```
get sURL
pop sHTML
```

The *pop* function or statement is useful for working with stack items.

push

The *push* statement pushes a value onto the stack. The parameter is a value to store.

```
push value
```

The specified value is added to the stack. The following code uses *push* and the stack to modify string data.

```
push sString
trim
removetags
pop sString
```

The *push* statement is useful for working with stack items.

store

The *store* function retrieves the top stack item. There are no parameters.

```
string store()
```

The result is the top stack item. Unlike *pop*, *store* does not remove the item from the stack. The following code uses *store* to save copies of stack data at various stages of processing.

```
push sData
store sHTML
removetags
store sText
pop
```

The statement form of *store* has one or more variables as parameters.

```
store var
```

The *store* function or statement is useful for copying stack items without removing them.

text

The *text* statement converts the top stack item into the *string* data type.

```
text
```

If the stack data is any data type other than *string*, it is converted into the *string* data type. The *string* statement is useful for insuring that stack data is in string form.

Table 5.11 The URL Functions

FUNCTION	DESCRIPTION
string absurl(*relative-url*, *base-url*)	Converts a relative URL to an absolute URL
boolean parseurl(*URL*, *Protocol*, *Server*, *Object*, *Port* [,*Path* [,*Filename*]])	Parses a URL
string urldecode(*value*)	Decodes a URL
string urlencode(*value*)	Encodes a URL

URL Functions

The URL functions perform actions on Web addresses. Table 5.11 enumerates the URL functions.

The URL functions are described individually in the following sections.

absurl

The *absurl* function converts a relative URL into an absolute (complete) URL. The parameter is a relative URL and a base URL which is used to resolve the relative URL.

```
string absurl(relative-url, base-url)
```

The result is an absolute URL. The following code shows *absurl* in use.

```
string sURL = "/sports/default.htm"
string sBaseURL = "http://www.my-news-site.com"
sURL = absurl(sURL, sBaseURL)
```

The *absurl* function is useful for resolving relative URLs.

parseurl

The *parseurl* function separates a URL into its individual pieces. The parameters are a URL and variables to receive the protocol, server, object, port, path, and filename. The *path* and *filename* parameters are optional.

```
boolean parseurl(URL, Protocol, Server, Object, Port [,Path
[,Filename]])
```

The *protocol* variable receives the URL's protocol, such as "ftp:" or "https:", and defaults to "http:" if no protocol is specifically identified in the URL. The *server* variable receives the main server name or IP address from the URL, such as "www.my-web-site.com". The *object* variable receives the part of the URL after the server name, such as a path and filename. The *port* variable receives the port number, and defaults to 0 if no port was specified in the URL. The *path* variable, if specified, receives the path part of the object, such as "/sports/". The *filename* variable, if specified, receives the filename part of the object, such as “default.htm”. The following code shows the use of *parseurl* to determine the parts of a URL.

```
string URL = "http://www.my-news-site.com/sports/hockey.htm"
string Protocol, Server, Object, Path, Filename
int Port
if !parseurl(URL, Protocol, Server, Object, Port, Path, Filename)
{
    show "The URL is invalid"
}
```

The *parseurl* function is for separating a Web address into its component parts.

urldecode

The *urldecode* function returns a URL to its original form, replacing encoded characters with the original characters. The parameter is a URL that may contain encoded values such as + for spaces and %xx for hex character sequences.

```
string urldecode(url)
```

The result is the URL with encoding sequences resolved. The following code uses *urldecode* to remove encoding characters from a URL .

```
sDecodedURL = urldecode(sURL)
```

The *urldecode* function has the opposite effect of the *urlencode* function.

urlencode

The *urlencode* function encodes parts of a URL, substituting certain character sequences just as a browser does. The parameter is a URL.

```
string urlencode(url)
```

The result is the URL with certain changes, such as plus signs in place of spaces and %xx hex sequences in place of reserved characters. The following code uses *urlencode* to add encoding characters to a URL.

```
sEncodedURL = urlencode(sURL)
```

The *urlencode* function has the opposite effect of the *urldecode* function.

Function Editions of NQL Statements

Many of the NQL statements can also be called as functions and return a true/false result. Calling a statement as a function is useful for testing success/failure in *if* statements and *while* loops. For example, consider the following code that copies a file.

```
open #file1, sFile1
create #file2, sFile2
read #file1
while
{
    write #file2
    read #file1
}
close #file1
close #file2
```

By using the function version of *open*, *create*, *read*, and *write*, the same code can be written more compactly and with better error checking. Each of these functions returns a Boolean true or false result. The following code is functionally equivalent.

```
if open(#file1, sFile1) and create(#file2, sFile2)
{
    while read(#file1, sLine) { write #file2, sLine }
    close #file1
    close #file2
}
```

Some NQL statements have function versions that return data. For example, the *getpage* function retrieves a Web page and returns its HTML.

```
string sHTML = getpage(sURL)
```

Functions that return data can be combined in expressions. For example, the following code retrieves a Web page, clips the BODY section of the page, and removes its tags in a single line.

```
sText = removetags(clip(getpage(sURL), "<BODY>", "</BODY>"))
```

The chief difference between NQL statements and NQL functions is the dependence on the stack. Network Query Language statements, which hearken back to NQL 1.0, use the stack as the primary means of operating on data. The NQL functions, introduced in NQL 2.0, work on data directly through parameters and can be used in expressions.

User-Defined Functions

User-defined functions are declared by stating their return type, their name, and (in parentheses) their parameters. The following code shows the definition of a function that replicates a string multiple times.

```
string DupeString(string value, int count)
{
    string result
    for i = 1, count
    {
        result = result & value
    }
    DupeString = result
}
```

The return type of a function may be void (no return type) or any of the standard NQL data types (*binary*, *boolean*, *currency*, *date*, *float*, *fuzzy*, *html*, *integer*, *object*, *string*, *unicode*, or *xml*).

To invoke a user-defined function, simply invoke its name, followed by parameter values in parentheses. Even if a function has no parameters, the parentheses must be specified.

```
string s1 = DupeString("*", 20)
string s2 = DupeString("hello", 2)
```

All function parameters are passed by value, which means the function cannot alter any variables passed to it.

Tutorial: Phone

Now that you have the inside scoop on expressions, operators, and functions, let's put them to use in an application. In this tutorial, you will create a script that comes up with all of the alphabetic variations for a seven-digit phone number. The results will be written to a text file that will print when the program finishes. Although this isn't a typical task for NQL, it affords a good opportunity to study NQL's expressions, operators, and functions in action.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the phone script.
3. Run the phone script.
4. Understand the phone script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Use expressions, operators, and functions.
- Declare and use variables and arrays.
- Print a text file.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Phone Script

In the NQL development environment, enter the script shown in Listing 5.1 and save it under the name phone.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//phone - compute alphabetic variations for a phone number

string PhoneNumber, PhoneAlpha

GetNumber:
    PhoneNumber = prompt("Enter Phone Number", "Please enter a 7-digit
phone number", "123-4567")
    if PhoneNumber=="" || PhoneNumber=="end" { end }
    PhoneNumber = remove(PhoneNumber, "-")
    PhoneNumber = remove(PhoneNumber, "(")
    PhoneNumber = remove(PhoneNumber, ")")
    PhoneNumber = remove(PhoneNumber, " ")
    if length(PhoneNumber)!=7 OR !isnumeric(PhoneNumber)
    {
        show "The phone number must be 7 digits."
        goto GetNumber
    }

SetupAlpha:
    int digit[7]

    digit[1] = mid(PhoneNumber, 1, 1)
    digit[2] = mid(PhoneNumber, 2, 1)
    digit[3] = mid(PhoneNumber, 3, 1)
    digit[4] = mid(PhoneNumber, 4, 1)
    digit[5] = mid(PhoneNumber, 5, 1)
    digit[6] = mid(PhoneNumber, 6, 1)
    digit[7] = mid(PhoneNumber, 7, 1)
```

Listing 5.1 Phone script.

```
    string letter1[10]
    string letter2[10]
    string letter3[10]

    letter1[0] = "0" : letter2[0] = "0" : letter3[0] = "0"
    letter1[1] = "1" : letter2[1] = "1" : letter3[1] = "1"
    letter1[2] = "A" : letter2[2] = "B" : letter3[2] = "C"
    letter1[3] = "D" : letter2[3] = "E" : letter3[3] = "F"
    letter1[4] = "G" : letter2[4] = "H" : letter3[4] = "I"
    letter1[5] = "J" : letter2[5] = "K" : letter3[5] = "L"
    letter1[6] = "M" : letter2[6] = "N" : letter3[6] = "O"
    letter1[7] = "P" : letter2[7] = "R" : letter3[7] = "S"
    letter1[8] = "T" : letter2[8] = "U" : letter3[8] = "V"
    letter1[9] = "W" : letter2[9] = "X" : letter3[9] = "Y"

ComputeAlpha:
    TextFile = PhoneNumber ".txt"
    create TextFile
    for d1 = 1, 3
    {
        c1 = Letter(1, d1)
        for d2 = 1, 3
        {
            c2 = Letter(2, d2)
            for d3 = 1, 3
            {
                c3 = Letter(3, d3)
                for d4 = 1, 3
                {
                    c4 = Letter(4, d4)
                    for d5 = 1, 3
                    {
                        c5 = Letter(5, d5)
                        for d6 = 1, 3
                        {
                            c6 = Letter(6, d6)
                            for d7 = 1, 3
                            {
                                c7 = Letter(7, d7)
                                PhoneAlpha = c1 c2 c3 c4 c5 c6 c7
                                write PhoneAlpha "\t"
                                files = files+1
                                if files > 8
                                {
                                    write "\r\n"
                                    files = 0
```

continues

Listing 5.1 Phone script. (continued).

```
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    close
    printfile TextFile
    end

string Letter(int DigitIndex, int LoopIndex)
{
    switch LoopIndex
    {
    case 1:
        Letter = letter1[digit[DigitIndex]]
        break
    case 2:
        Letter = letter2[digit[DigitIndex]]
        break
    case 3:
        Letter = letter3[digit[DigitIndex]]
        break
    }
}
```

Listing 5.1 Phone script. (continued).

If you prefer, you may copy phone.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select phone.nql from the Tutorials\ch05 folder.

At this point, the script in Listing 5.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Phone Script

Now, we are ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Almost immediately, you will be prompted to enter a phone number. If this doesn't happen, check your script for typographical errors and try again.

Enter a phone number and click the OK button. The script will then work out the various alphabetical phone number combinations. When it is finished, a text file named *phone-number* will have been created and will print out on your default printer. The alphabetized phone numbers will be printed eight across.

```
1ADGJMP  1ADGJMR  1ADGJMS  1ADGJNP  1ADGJNR  1ADGJNS  1ADGJOP  1ADGJOR
1ADGJOS  1ADGKMP  1ADGKMR  1ADGKMS  1ADGKNP  1ADGKNR  1ADGKNS  1ADGKOP
```

At this point you should have seen the phone script run, generating a printed report of the alphabetized phone numbers. This script has exercised expressions, functions, variables, and arrays. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Phone Script

We now want to make sure we understand every part of the phone script. The first line is a comment line.

```
//phone - compute alphabetic variations for a phone number
```

Next, two string variables are declared. *PhoneNumber* will hold a seven-digit phone number entered by the user. *PhoneAlpha* will hold the various alphabetized numbers as they are generated.

```
string PhoneNumber, PhoneAlpha
```

The next element of the script is a label, *GetNumber*. The label helps describe this section of the program but will also be needed to jump back to this point in the event that a bad phone number is entered. After the label, a *prompt* function requests a phone number from the user and assigns it to *PhoneNumber*.

```
GetNumber:
    PhoneNumber = prompt("Enter Phone Number",
        "Please enter a 7-digit phone number", "123-4567")
```

The phone number needs to be validated, since the program won't work correctly unless there are exactly seven digits to work with. First, the input is checked for an empty string or "end," which is taken to mean the program should exit. Next, the dash, space, and parentheses are removed from the string, hopefully leaving just seven digits.

```
if PhoneNumber=="" || PhoneNumber=="end" { end }
PhoneNumber = remove(PhoneNumber, "-")
PhoneNumber = remove(PhoneNumber, "(")
PhoneNumber = remove(PhoneNumber, ")")
PhoneNumber = remove(PhoneNumber, " ")
```

Next, the phone number is checked for correctness with an *if* statement. The *if* statement tests two conditions. The first condition uses the *length*( ) function to check that the phone number is seven characters in length. The second condition uses the *isnumeric*( ) function to check that the phone number contains only digits. If either or both conditions are not satisfied, the code block displays an error message, and a *goto* statement prompts the user again for a phone number.

```
if length(PhoneNumber)!=7 OR !isnumeric(PhoneNumber)
{
     show "The phone number must be 7 digits."
     goto GetNumber
}
```

With a valid phone number, the script now proceeds to the second section of the script, labeled *SetupAlpha*. The phone number entered by the user is currently a string, but for our computations it will be more convenient to represent each digit in an array of integers. The script declares an array of seven integers named *digit* for this purpose. The next seven assignment statements set elements 1 through 7 of *digit*, using the *mid*( ) function to extract each character of the phone number string.

```
SetupAlpha:
     int digit[7]

     digit[1] = mid(PhoneNumber, 1, 1)
     digit[2] = mid(PhoneNumber, 2, 1)
     digit[3] = mid(PhoneNumber, 3, 1)
     digit[4] = mid(PhoneNumber, 4, 1)
     digit[5] = mid(PhoneNumber, 5, 1)
     digit[6] = mid(PhoneNumber, 6, 1)
     digit[7] = mid(PhoneNumber, 7, 1)
```

The remaining code in this section declares three string arrays, which will hold the standard telephone letter-digit keypad associations. There are three arrays because the phone keys have up to three letters on them (0 and 1 have none). There are ten elements in each array to handle digits 0 through 9.

```
string letter1[10]
string letter2[10]
string letter3[10]
```

The array is now filled with values. For each possible phone digit (0 through 9), three letter values are set. The characters for digits 0 and 1 are numbers because there are no letters on those keys.

```
letter1[0] = "0" : letter2[0] = "0" : letter3[0] = "0"
letter1[1] = "1" : letter2[1] = "1" : letter3[1] = "1"
letter1[2] = "A" : letter2[2] = "B" : letter3[2] = "C"
letter1[3] = "D" : letter2[3] = "E" : letter3[3] = "F"
letter1[4] = "G" : letter2[4] = "H" : letter3[4] = "I"
letter1[5] = "J" : letter2[5] = "K" : letter3[5] = "L"
letter1[6] = "M" : letter2[6] = "N" : letter3[6] = "O"
```

```
letter1[7] = "P" : letter2[7] = "R" : letter3[7] = "S"
letter1[8] = "T" : letter2[8] = "U" : letter3[8] = "V"
letter1[9] = "W" : letter2[9] = "X" : letter3[9] = "Y"
```

At this point, we have data stored that will allow us easily to reference the three-letter combination assigned to any key. For example, to find out the letters for the digit 2, we merely access *letter1[2]*, *letter2[2]*, and *letter3[2]* ("A", "B", and "C"). We are now ready to begin generating alphabetized phone numbers.

The third and final section of the script begins with the label *ComputeAlpha*. The name of the text file to write to is determined by appending .txt to the entered phone number. A *create* statement creates the text file.

```
ComputeAlpha:
    TextFile = PhoneNumber ".txt"
    create TextFile
```

Seven nested loops follow, with control variables named *d1* through *d7*. Each loop iterates three times. This will cycle through all permutations of letters for each of the seven digits. In each loop, a letter is assigned to string variables *c1* through *c7*.

```
for d1 = 1, 3
{
    c1 = Letter(1, d1)
    for d2 = 1, 3
    {
        c2 = Letter(2, d2)
        ...
```

The letters are computed by calling a user-defined function, *Letter*, that is at the end of the script. Given a position in the phone number (1 through 7) and a loop index (1 through 3), the function returns the appropriate letter. For example, if the user entered phone number "3484848", the result of *Letter(1, 2)* would be E, because digit 1 is 3, and the second of the letters for the 3 key (DEF) is E.

In the innermost seventh loop, there is additional code beyond computing the next letter. Now that all seven letters are known, *c1* through *c7* are appended into a single string and assigned to the variable *PhoneAlpha*. The phone number and a tab are written to the text file. A counter named *files* is incremented, and if greater than eight a newline is written out to the text file and *files* is reset to 0.

```
for d7 = 1,3
{
    c7 = Letter(7, d7)
    PhoneAlpha = c1 c2 c3 c4 c5 c6 c7
    write PhoneAlpha "\t"
    files = files+1
    if files > 8
    {
        write "\r\n"
        files = 0
        }
    }
```

After all of the *for* loops, the text file is closed. A *printfile* statement prints the text file out. Lastly, an *end* statement stops the program.

```
close
printfile TextFile
end
```

The last part of the script is the definition for the user-defined function, *Letter*, we encountered earlier. In order to return the right letter, the function has to accomplish two things. First, it has to transform a positional digit (1 through 7) into the actual digit of the phone number entered (for example, if "555-7688" is the phone number, positional digit 3 must become actual digit 5). Second, it has to take the loop index (1 through 3) into account to get to the correct letter for the digit in question. The *digit[DigitIndex]* expression converts the digit position into the actual digit, and is used as the index to the *letter1*, *letter2*, or *letter3* array. A *switch* statement chooses which of the three arrays to target based on the loop index.

```
string Letter(int DigitIndex, int LoopIndex)
{
    switch LoopIndex
    {
    case 1:
        Letter = letter1[digit[DigitIndex]]
        break
    case 2:
        Letter = letter2[digit[DigitIndex]]
        break
    case 3:
        Letter = letter3[digit[DigitIndex]]
        break
    }
}
```

At this point, you've seen how expressions, operators, and functions work in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Instead of showing all phone number variations, discard those that are not likely to form words. For example, if an alphabetized phone number contains no vowels, it could be skipped.
- Adapt the program for four-letter sequences, such as PQRS, which is commonly listed on the 7 key of cellular phones.

Chapter Summary

Expressions are combinations of terms and operators.

- Terms are constants, variables, or functions.
- There are arithmetic, logical, comparison, string, and fuzzy operators.
- There is a hierarchy to operators that can be manipulated through the use of parentheses.

There are over 80 built-in functions in NQL.

- The casting functions convert values to different data types.
- The string functions search and manipulate string values.
- The date/time functions get, set, and convert dates and times.
- The stack functions push, pop, and otherwise manipulate the stack.
- The URL functions parse, encode, and decode Web addresses.,

Many of NQL's statements have function versions, the majority of which return a Boolean result. Calling statements as functions is convenient for *if* and *while* loops.

User-defined functions allow you to define procedures that can be treated as new statements or used within expressions.

- Functions may return any of the twelve data types, or if declared void, they return nothing.
- Functions invoked as statements are formed as keyword parameter-list.
- Functions invoked within expressions are formed as *keyword*(*parameter-list*).

At this point, you've seen how expressions, operators, and functions work in NQL. The remaining chapters focus on topical discussion, introducing families of statements and functions, and may be read in any order.

CHAPTER

6

Working with Files

Disk files are the primary means of computer storage. Network Query Language provides extensive capabilities for working with files and directories. This chapter describes them in three sections: File input/output, file operations, and directory operations. The tutorial at the end of the chapter implements a Find Files script that searches the files in a disk directory and identifies those containing search text.

File Input/Output

File operations in NQL allow files to be created, opened, read from, written to, and closed. Table 6.1 lists the NQL statements for file input/output.

Note that many file statements may be preceded by a *#session* parameter, which allows more than one file to be open at the same time.

Most of the NQL file input/output statements are also available in function form. Table 6.2 lists the file input/output functions. Most of them return a Boolean result that indicates success/failure.

Some of the file statements and functions are described individually in the following sections.

Table 6.1 File Input/Output Statements

STATEMENT	PARAMETERS	DESCRIPTION
append	[#*session*,] [*filespec*]	Opens a file for appending
close	[#*session*,] *path*	Closes a file
copy	*oldfile*, *newfile*	Copies a file
create	[#*session*,] [*filespec*]	Creates a new file
delete	*filespec*	Deletes a file
dir	[*wildcard*]	Returns a directory listing
fileversion	*filespec* [, *var*]	Returns the version number of a file
getdir	[*var*]	Returns the current working directory
load	[*filespec*]	Loads a file onto the stack
lookup	[*value*]	Tests for a file's existence and returns its size
makedir	*path*	Creates a directory
open	[#*session*,] [*filespec*]	Opens a file for reading or writing
read	[#*session*,] [*var*]	Reads a line from the specified file
read	[#*session*,] #*record*	Reads a record from the specified file
removedir	*path*	Removes a directory
rename	*oldname*, *newname*	Renames a file
save	*filespec*	Saves the top stack item to a file
setdir	[*path*]	Changes the working directory
uniquefilename	[*var*,[*path*]]	Computes a unique filename
write	[#*session*,] [*value-list*]	Writes the specified value(s) to a file
write	[#*session*,] #*record*	Writes the specified record to a file

File Modes

NQL can open a file in one of three modes: *line*, *byte*, or *unicode*. Table 6.3 summarizes the file modes.

Line mode is NQL's default file mode. In line mode, data may be read from a file one line at a time. Line mode is useful for reading line-oriented text composed in a single-byte character set. In line mode, the *read* statement or function reads in an entire line of text each time. A line ends in a newline or carriage return/line feed sequence.

In *byte mode*, data may be read from a file one single-byte character at a time. Byte mode is useful for reading binary files as well as text files composed in a single-byte

Table 6.2 File Input/Output Functions

FUNCTION	DESCRIPTION
boolean append([#*session*,] [*filespec*])	Opens a file for appending
boolean close([#*session*,] *path*)	Closes a file
boolean copy(*oldfile*, *newfile*)	Copies a file
boolean create([#*session*,] [*filespec*])	Creates a new file
boolean delete(*filespec*)	Deletes a file
boolean dir([*wildcard*])	Returns a directory listing
boolean fileversion(*filespec*)	Returns the version number of a file
boolean getdir([*var*])	Returns the current working directory
boolean load([*filespec*])	Loads a file onto the stack
boolean lookup([*value*])	Tests for a file's existence and returns its size
boolean makedir(*path*)	Creates a directory
boolean open([#*session*,] [*filespec*])	Opens a file for reading or writing
boolean read([#*session*,] [*var*])	Reads a line from the specified file
boolean read([#*session*,] #*record*)	Reads a record from the specified file
boolean removedir(*path*)	Removes a directory
boolean rename(*oldname*, *newname*)	Renames a file
boolean save(*filespec*)	Saves the top stack item to a file
boolean setdir([*path*])	Changes the working directory
boolean uniquefilename([*var*,[*path*]])	Computes a unique filename
boolean write([#*session*,] [*value-list*])	Writes the specified value(s) to a file
boolean write([#*session*,] #*record*)	Writes the specified record to a file

Table 6.3 File Modes

MODE	CHARACTER SIZE	LOGICAL UNIT OF INPUT
Line	1 byte	1 line, consisting of multiple characters ending in a newline sequence
Byte	1 byte	1 character
Unicode	2 bytes	1 wide character

character set. In byte mode, the *read* statement or function reads just one 8-bit byte each time.

Unicode mode is for languages that can't be represented in a single byte per character. Unicode supports all of the world's written character symbols through the use of wide (2-byte) characters. In Unicode mode, the *read* statement or function reads a 2-byte wide character each time.

The mode a file is opened in primarily affects reading, since the *write* statement can write any combination of bytes, wide characters, and lines.

Sessions

The NQL file operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

1. A file session is opened with create, open, or append.
2. One or more read or write operations are performed.
3. The file session is closed with *close*.

Multiple File Sessions

NQL uses a very simple syntax for file operations, such as the following line.

```
open "memo.txt"
```

When there is a need to open multiple files simultaneously, identifiers are needed to distinguish one file from another. In NQL, these identifiers are known as file session IDs. A file session ID is a name preceded by a pound sign, such as #*cust* or #*OutputFile*. When a session ID is specified, it is the first parameter of a statement or function. For example,

```
open #source, "memo.txt"
open #dest, "newmemo.txt"
```

Opening Files

There are three NQL statements (and functions) for opening files: *create*, *open*, and *append*. As their names suggest, there is one mode for creating new files, one for opening existing files, and one for appending to existing files. In fact, all three statements will open existing files or create new files if necessary. The differences lie in what happens to the prior contents of a file, if any, and where the initial position is in the file. Table 6.4 contrasts the three methods of opening files.

Each of these methods is described individually in the following section.

Table 6.4 File Open Methods

METHOD	PREVIOUS FILE CONTENTS	FILE POINTER	INTENDED USE
create	destroyed	start of file	For creating new files
open	retained	start of file	For reading from existing files
append	retained	end of file	For appending to new or existing files

Creating a New File

To create a new, empty file ready for writing, use the *create* statement. Even if the specified file already exists, its prior contents will be erased. The first parameter is a file specification. The second parameter, which is optional, controls the file mode (*byte*, *line*, or *unicode*) and defaults to *line*.

```
create filespec [, mode]
```

The following script creates a new file and writes some text to it.

```
create "presidents.txt"
write "George Washington\r\n"
write "John Adams\r\n"
write "Thomas Jefferson\r\n"
close
```

The use of *create* guarantees you are starting with a fresh, empty file each time, regardless of whether that file previously existed.

Opening an Existing File

To open an existing file ready for reading, use the *open* statement. If no file exists, it will be created. If the file does exist, you will be positioned at its start. The first parameter is a file specification. The second parameter, which is optional, controls the file mode (*byte*, *line*, or *unicode*) and defaults to *line*.

```
open filespec [, mode]
```

The following script opens an existing file and reads its contents one line at a time.

```
open "presidents.txt"
while read(line)
{
     show line
}
close
```

The most common use of *open* is to open an existing file for reading, but you may write to the file as well.

Appending to a File

To open an existing file for appending and writing, use the *append* statement. If no file exists, it will be created. If the file does exist, you will be positioned at its end. The first parameter is a file specification. The second parameter, which is optional, controls the file mode (*byte*, *line*, or *unicode*) and defaults to *line*.

```
append filespec [, mode]
```

The following script appends to an existing file.

```
append "presidents.txt"
write "James Madison\r\n"
write "James Monroe\r\n"
close
```

The *append* statement is useful for adding new information to a file.

Closing Files

To close an open file, issue a *close* statement.

```
close
```

If a file was opened with a file session ID, specify that ID when closing it.

```
close #ID
```

If you forget to close a file, NQL will close all open files when a script ends.

Reading from Files

The *read* statement or function reads lines or characters from an open file. The basic unit of reading depends on how the file was opened.

When files are opened in line mode, *read* inputs an entire text line at a time. The following script copies one file to another, reading and writing one line at a time.

```
string line

open #input, "file1.txt", "line"
open #output, "file2.txt", "line"
```

```
while read(#input, line)
{
    write #output, line "\n"
}
close #input
close #output
```

When files are opened in byte mode, *read* inputs a single character at a time. The following script copies one file to another, reading and writing one character at a time.

```
string char

open #input, "file1.dat", "byte"
open #output, "file2.dat", "byte"
while read(#input, char)
{
    write #output, char
}
close #input
close #output
```

When files are opened in unicode mode, *read* inputs a wide character at a time. The following script copies one file to another, reading and writing Unicode characters.

```
unicode char

open #input, "file1.dat", "unicode"
open #output, "file2.dat", "unicode"
while read(#input, char)
{
    write #output, char
}
close #input
close #output
```

Writing to Files

The *write* statement or function writes data to an open file. Any kind of data can be written to a file with *write*, regardless of the mode a file was opened in.

```
write data
```

When multiple files are open, file session IDs are used to distinguish one file from another.

```
write #names, Company
write #balances, Balance
```

To write Unicode wide characters to a file, specify data in a unicode variable, or use the unicode() function to transform single-byte characters to wide characters. The following code writes to a Unicode file. The first *write* passes a variable name directly, since it is of type *unicode*. The second *write* transforms a string variable to 2-byte wide characters with the *unicode()* function.

```
unicode name = u"These are wide characters"
string name2 = "These are single-byte characters"
open "uni.txt", "unicode"
write name
write unicode(name2)
close
```

File Operations

In addition to direct file input/output, NQL can perform a variety of operations on files, including renaming, copying, and deleting them. Table 6.5 lists the NQL file operation statements.

Function versions of the file operation statements are shown in Table 6.6

The file operation statements and functions are described individually in the following sections.

Copying a File

The *copy* statement or function makes a copy of a disk file. *Copy* takes two parameters, the file specification of an existing file and the file specification of the copy to be made.

```
copy oldfile, newfile
boolean copy(oldfile, newfile)
```

Table 6.5 File Operation Statements

STATEMENT	PARAMETERS	DESCRIPTION
copy	*oldfile, newfile*	Copies a file
delete	*file*	Deletes a file
fileversion	*file* [, *var*]	Returns information about a file
load	[*file*]	Loads a disk file in its entirety
lookup	[*file*]	Checks for a file's existence and returns its size on the stack
rename	*oldfile*, *newfile*	Renames a file
save	*file*	Saves the top stack item as a disk file
uniquefilename	[*var* [, *path*]]	Computes a unique file name

Table 6.6 File Operation Functions

FUNCTION	DESCRIPTION
boolean copy(*oldfile, newfile*)	Copies a file
boolean delete(*file*)	Deletes a file
string fileversion(*file* [, *var*])	Returns information about a file
boolean load([*file*])	Loads a disk file in its entirety
int lookup([*file*])	Checks for a file's existence and returns its size on the stack
boolean rename(*oldfile*, *newfile*)	Renames a file
boolean save(*file*)	Saves the top stack item as a disk file
boolean uniquefilename([*var* [, *path*]])	Computes a unique file name

The file is copied, and if no errors occur a success condition results (the function returns true). The following code copies a file with *copy*.

```
copy "pricelist.txt", "pricelist-backup.txt"
```

The *copy* statement/function is useful for duplicating or backing up disk files.

Renaming a File

The *rename* statement or function renames a disk file. *Rename* takes two parameters, the file specification of an existing file and the new name to give the file.

```
rename oldfile, newfile
boolean rename(oldfile, newfile)
```

The file is renamed, and if no errors occur a success condition results (the function returns true). The following code renames a file with *rename*.

```
rename "pricelist-backup.txt", "pricelist.txt"
```

The *rename* statement/function is useful for file management and backup/recovery operations.

Deleting a File

The *delete* statement or function erases a disk file. *Delete* takes one parameter, the file specification of the file to be deleted.

```
delete file
boolean delete(file)
```

The file is deleted, and if no errors occur a success condition results (the function returns true). The following code deletes a file with *delete*.

```
delete "tempfile.dat"
```

The *delete* statement/function is useful for removing unwanted or obsolete disk files.

Loading a File

The *load* statement or function loads a file in its entirety and pushes it onto the stack. The parameter is the file specification to load. If omitted, the file specification is taken from the stack.

```
load [file]
boolean load(file)
```

The specified file is loaded onto the stack, and if no errors occur a success condition results (the function returns true). The following code uses *load* to load a disk file onto the stack.

```
load "mailmerge.txt"
```

The *load* statement/function is useful for reading an entire file into memory rather than having to take the individual steps of opening it, reading from it in chunks, and closing it.

Looking Up a File

The *lookup* statement or function returns the size of a file. *Lookup* takes one optional parameter, a file specification. If omitted, the file specification comes from the stack.

```
lookup [file]
int lookup([file])
```

If the file exists, the statement pushes its size in bytes onto the stack; the function returns the size as its integer result. If the file is not found, the value returned is 0. The following code uses *lookup* to determine whether a file exists.

```
if lookup(sFile)==0 { show "The file does not exist." )
```

The *lookup* statement/function is useful for checking whether a file exists.

Saving a File

The *save* statement or function saves the top stack item to a file. The parameter is a file specification.

```
save file
boolean save(file)
```

The top item on the stack is saved to the specified file, and if no errors occur a success condition results (the function returns true). The following code uses *save* to store a stack item to disk.

```
push "The quick brown fox jumped over the lazy dogs."
save "silly.txt"
```

The *save* statement/function is useful for saving an entire file from memory rather than having to take the individual steps of creating it, writing to it in chunks, and closing it.

Obtaining a File's Version

The *fileversion* statement or function retrieves a file's version number. *Fileversion* takes one required parameter, the file specification of the file, and an optional parameter, a variable to receive the version information.

```
fileversion file [, var]
string fileversion(file)
```

The statement retrieves the file's version number, formats it as a string, and stores it in the specified variable. If no variable was specified, the version number is pushed onto the stack. The function returns the version number as its result. The following code uses *fileversion* to display a file's version number.

```
show fileversion("c:\\Program Files\\NQL\\nql.dll")
```

The *fileversion* statement/function is useful for checking that files are up to date or comparing two editions of the same file.

Obtaining a Unique Filename

The *uniquefilename* statement or function computes the name of a unique (nonexistent) temporary file in the current working directory; it accepts two parameters, both of which are optional. The first parameter is a variable name to receive the name of the file; if omitted, it is pushed onto the stack. The second parameter is a path/file string that is used as a prefix for the filename.

```
uniquefilename [var [, path]]
boolean uniquefilename([var [, path]])
```

The statement looks for 0.tmp, 1.tmp, and so on until a unique filename is found. If a path is specified, it is used in the lookup process. The result is returned in the specified variable or pushed onto the stack. If no errors occur, a success condition results (the function returns true). The following code uses *uniquefilename* to create a temporary file.

```
string sPath = "c:\\Temp\\"
string sFilespec
if uniquefilename(sFilespec, sPath)
{
    create sFilespec
    ...write data to file...
    close
}
```

The *uniquefilename* statement/function is useful for generating new or temporary filenames without concern for overwriting existing files.

Directory Operations

In addition to directly reading and writing files, NQL can perform directory-level operations. This includes the ability to create directories, remove directories, and rename and delete files from directories. The directory statements are summarized in Table 6.7.

The directory statements have equivalent functions listed in Table 6.8.

Table 6.7 Directory Statements

STATEMENT	PARAMETERS	DESCRIPTION
dir	[*wildcard*]	Returns a directory listing of files matching the wildcard specification
getdir	[*var*]	Returns the current directory path
makedir	*path*	Creates a directory folder
removedir	*path*	Deletes a directory folder
setdir	[*path*]	Sets the current directory

Table 6.8 Directory Functions

FUNCTION	DESCRIPTION
boolean dir([*wildcard*])	Returns a directory listing of files matching the wildcard specification
string getdir()	Returns the current directory path
boolean makedir(*path*)	Creates a directory folder
boolean removedir(*path*)	Deletes a directory folder
boolean setdir([*path*])	Sets the current directory

The directory statements and functions are described individually in the following sections.

Obtaining a Directory Listing

The *dir* statement or function returns a directory listing of files. *Dir* takes an optional wildcard parameter such as **.txt* or *c:\\a*.exe*. If the wildcard parameter is omitted, **.** is implied on the current working directory.

```
dir [path]
```

The names of files matching the directory specification are stored on the stack as a single string, where each file name is separated from the next by a newline. The usual way to move through the list of files is with the *nextline* statement or function. The following code obtains a directory, then retrieves each file name in the directory individually.

```
dir
while nextline()
{
     pop filename
     output filename
}
```

The *dir* statement/function is useful for determining the contents of directories.

Getting the Current Working Directory

The *getdir* statement obtains the current working directory. The directory path is stored in the specified variable or is pushed on the stack if no variable name has been specified.

```
getdir [var]
```

The path returned is guaranteed to be normalized. Under Windows, that means the return value will always end in a backslash, such as c:\ or d:\Program Files\Documents\Memos\. A success condition results unless an error occurs.

The following code obtains the current directory path, then uses that path to make a relative filename into an absolute one.

```
string path, file
getdir path
file = path & "test.txt"
show "The complete path for test.txt is " file
```

The function version of *getdir* is the current working directory returned as a string value. There are no parameters.

```
string getdir()
```

The *getdir* statement/function is useful for finding out your current disk location.

Changing the Current Working Directory

The *setdir* statement changes the current working directory. The path is either specified as a parameter or is taken from the top stack item.

```
setdir [path]
```

A success condition results unless an error occurs. The following code uses *setdir* to change the working directory several times.

```
setdir "c:\\newfiles"
load "submission.dat"
delete "submission.dat"
setdir "c:\\archive"
save "submission.dat"
```

The function form of *setdir* has identical parameters and functionality as the statement form, and returns a Boolean true result if successful.

```
boolean setdir(path)
```

The *setdir* statement/function is useful for moving to convenient disk locations.

Creating a Directory

The *makedir* statement creates a directory folder. The parameter is a path.

```
makedir path
```

A success condition results unless an error occurs. The following code uses *makedir* to create a directory folder.

```
makedir "c:\\temp"
```

The function form of *makedir* is identical to the statement form, and returns true if successful.

```
boolean makedir(path)
```

The *makedir* statement/function is useful for creating disk directories.

Removing a Directory

The *removedir* statement deletes a directory folder. The parameter is a path.

```
removedir path
```

A success condition results unless an error occurs. The following code uses *removedir* to delete a directory folder.

```
removedir "c:\\temp"
```

The function form of *removedir* is identical to the statement form, and returns true if successful.

```
boolean removedir(path)
```

The *removedir* statement/function is useful for removing disk directories.

Tutorial: FindFiles

Now that you are familiar with file and directory operations, let's put them to work. In this tutorial, you will create a script that performs a Find Files task, searching files in a directory and identifying those that contain search text.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the FindFiles script.
3. Run the FindFiles script.
4. Understand the FindFiles script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Scan the files in a directory.
- Load files into memory.
- Scan text for occurrences of search text.
- Write results to a disk file.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon or by selecting *Program Files,* *Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the FindFiles Script

In the NQL development environment, enter the script shown in Listing 6.1 and save it under the name findfiles.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy findfiles.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select findfiles.nql from the Tutorials\ch06 folder.

At this point, the script in Listing 6.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the FindFiles Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Almost immediately, you will be prompted to enter a directory path and search text. If this doesn't happen, check your script for typographical errors and try again. For the first prompt, enter a directory path such as c:\ or d:\My Documents. Specify a wildcard file type if you wish, such as **.txt* or **.doc*. If you leave out the file type, *.* will be assumed. For the second prompt, enter search text you want to find, such as a subject or the name of a person, organization, or product.

The script will create a file named search.txt. As files are scanned in the specified directory, a line is written to search.txt each time a file is found that contains the search text. When the script ends, search.txt is opened on the desktop so the user can see the results.

The result lines in search.txt indicate the filename containing a match and the number of matches.

```
Path: c:\*.txt
Search text:

c:\disclaim.txt contains 7 matches
c:\abc.txt contains 20 matches
c:\caodbc.txt contains 8 matches
c:\sort1.txt contains 8 matches
```

At this point you should have seen the FindFiles script run, searching a disk directory for matching files and creating a results file. This script has exercised directory access, loading and searching of file data, and writing of results to an output file. In the next step, we'll dissect the script and explain exactly how it works.

```
//FindFiles - searches for files that contain search text

string SearchText
string Path

Path = prompt("Enter Directory Folder",
    "Please enter the path (directory folder) you wish to search:",
"c:\\*.txt")

SearchText = prompt("Search Text", "Please enter the text you are
searching for", "the")

if find(Path, "*")==0
{
    if right(Path, 1)!="\\" { Path = Path & "\\" }
    Path = Path & "*.*"
}

create "search.txt"
write "Path: " Path "\r\n"
write "Search text: " SearchText "\r\n\r\n"

if dir(Path)
{
    while nextline(File)
    {
        load File
        Matches = count(pop(), SearchText)
        if Matches>0
        then
        {
            write File " contains " Matches " matches\r\n"
        }
    }
}
else
{
    write "Error: unable to access path " Path "\r\n"
}

close
opendoc "search.txt"
```

Listing 6.1 FindFiles script.

Step 4: Understand the FindFiles Script

We now want to make sure we understand every part of the FindFiles script. The first line is a comment line.

```
//FindFiles - searches for files that contain search text
```

Two variables are declared, *SearchText* and *Path*. These will hold the user-entered parameters for the search, a disk directory and text to search for.

```
string SearchText
string Path
```

Next, the user is prompted for the parameters. The prompt function lets the user enter values, and supplies default values of *c:\*.txt* and *the*.

```
Path = prompt("Enter Directory Folder",
    "Please enter the path (directory folder) you wish to search:",
"c:\\*.txt")

SearchText = prompt("Search Text", "Please enter the text you are
searching for", "the")
```

The path is now examined. If it contains an asterisk, it is assumed to be a complete path and wildcard directory specification. If there is no asterisk, *.* is added to the path, after making sure it ends in a backslash.

```
if find(Path, "*")==0
{
    if right(Path, 1)!="\\" { Path = Path & "\\" }
    Path = Path & "*.*"
}
```

The results file, search.txt, is now opened for output with the *create* statement. The parameters for the search are written to the file, and a blank line.

```
create "search.txt"
write "Path: " Path "\r\n"
write "Search text: " SearchText "\r\n\r\n"
```

The directory is now retrieved for the specified path and stored on the stack. If successful, the files in the directory are processed through the use of a *while* loop and *nextline*(), which individually returns each file from the directory listing. Within the *while* block, the file search is handled by loading the entire file onto the stack (with *load*), then feeding the file's content to a *count*() function, which returns the number of occurrences of the search text. The use of *pop*() gives the file data to *count*() and also cleans up the stack. If the number of matches is greater than zero, a line is written out to the results file via a *write* statement.

```
if dir(Path)
{
    while nextline(File)
    {
        load File
        Matches = count(pop(), SearchText)
        if Matches>0
        then
        {
            write File " contains " Matches " matches\r\n"
        }
    }
}
else
{
    write "Error: unable to access path " Path "\r\n"
}
```

After all of the files have been processed, *close* closes the results file. The results file is opened on the desktop via *opendoc*.

```
close
opendoc "search.txt"
```

At this point, you've seen a useful application of file and directory operations.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Check the file type of each file and skip files that are not textual in nature. Processing binary files such as .exe, .dll, and .bin files can be time-consuming and wasteful.
- Handle subdirectories of files. This can be done by differentiating between files and subdirectories in the directory listing.

Chapter Summary

Network Query Language provides standard file input/output capabilities that allow files to be created, opened, appended, read from, written to, and closed.

- Files are opened with the *create*, *open*, or *append* statements, and closed with the *close* statement. The *create* statement creates a new, empty file. The *open* statement opens an existing file for reading or modifying. The *append* statement opens an existing file for appending.
- NQL can open a file in one of three modes: *line*, *byte*, and *unicode*. The mode affects how data is read from files. Line mode reads one line of text at a time.

Byte mode reads one single-byte character at a time. Unicode mode reads one two-byte wide character at a time.

- The *read* and *write* statements perform input and output to open files. The *read* statement reads lines or characters or wide characters from an open file. The basic unit of reading depends on the mode in which the file was opened. The *write* statement writes data to an open file.

In addition to basic reading and writing of files, NQL can perform direct operations on files.

- The copy, delete, and rename statements duplicate a file, erase a file, or change the name of a file.
- The lookup and fileversion statements return information about files.
- The load and save statements move entire files to and from the stack.
- The uniquefilename statement computes a unique file name for a temporary file.

Directory operations perform functions on disk directories (folders).

- The dir statement retrieves a listing of the files in a folder.
- The getdir and setdir statements return or change the current working directory.
- The makedir and removedir statements create or delete a folder.

The file and directory operations covered in this chapter give you the ability to create, change, and use local computer disk storage. Chapter 7, "Transferring Files," describes how to move files to and from other computers.

CHAPTER

7

Transferring Files

The Internet protocol for moving files between systems is called File Transfer Protocol (FTP), which allows a computer to perform certain kinds of file-related tasks on a remote computer: Log on, move to various directories, list files in directories, send or receive files, create files, rename files, and delete files. Network Query Language provides complete support for FTP sessions, allowing your scripts to act as an FTP client, moving files to and from remote systems. This chapter begins with an explanation of FTP itself, then covers NQL's FTP provisions. The tutorial at the end of the chapter uses FTP to retrieve files from a public-domain FTP server.

Background on FTP

Like most Internet protocols, File Transfer Protocol is based on a client-server model. An FTP server accepts connections and responds to FTP commands; an FTP client makes the connection and issues the commands. Interactive FTP clients are used by interactive users, but automated FTP clients can be created by NQL's FTP support. Figure 7.1 illustrates the communication between FTP clients and an FTP server.

The instructions sent to the server by the client are in an internal command language. Table 7.1 lists the commonly used FTP commands.

By default, FTP uses port 21 for communication. However, you may find other port numbers in use for security purposes.

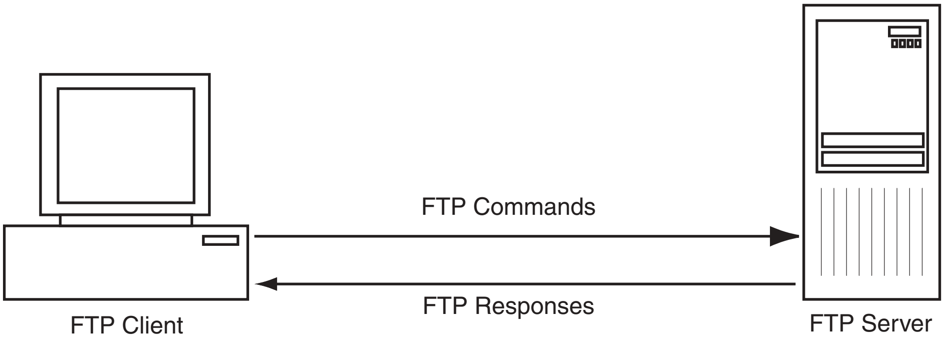


Figure 7.1 FTP Communication.

Table 7.1 FTP Commands

COMMAND	DESCRIPTION
CD *path*	Changes the working directory to the specified path
DELETE *file*	Deletes the specified file
DIR *wildcard*	Returns a directory of files
GET *file1 file2*	Retrieves *file1* from the remote system and stores it locally as *file2*
MKDIR *path*	Creates a directory on the remote system
PUT *file1 file2*	Transmits local *file1* to the remote system and stores it as *file2*
PWD	Returns the current directory on the remote system
QUIT	Disconnects from the remote system
RECV *file1 file2*	Retrieves *file1* from the remote system and stores it locally as *file2*
RENAME *file1 file2*	Renames a file on the remote system
RMDIR *path*	Removes a directory from the remote system
SEND *file*	Transmits local *file* to the remote system
SIZE *file*	Returns the size of the specified file

All FTP servers authenticate, meaning they require a user name and password before they can be accessed. Many FTP servers allow anonymous access, but it is still necessary to provide a user name and password. The convention for anonymous access is to specify the user name "anonymous" and enter your e-mail address as the password.

All FTP servers maintain the concept of a current directory. The initial directory is decided by the FTP server administrator and may be based on your user name.

Depending on your permissions, the FTP server may permit you to move around to different directories and even create or delete directories.

File transfer depends on which of two modes an FTP server is in: ASCII mode or binary mode. In ASCII mode, files are transmitted logically as text, which may introduce changes in the content of the files to hide differences in operating systems. In binary mode, files are transmitted identically byte for byte.

FTP Operations

There are two classes of FTP statements in NQL. For isolated operations such as getting or putting a file, there are all-in-one commands that handle connection, file transfer, and disconnection in a single convenient statement. There are also individual statements that allow you to control every aspect of an FTP session, including connection, changing directory, sending and receiving files, other file and directory operations, and disconnecting.

Table 7.2 lists the all-in-one FTP statements, each of which open, maintain, and close a complete FTP session.

Table 7.3 lists the individual FTP statements. These statements are used in combination to establish an FTP connection, perform operations, and disconnect.

The FTP statements may be preceded by a *#session* parameter, which allows more than one FTP session to be open at the same time.

Table 7.2 All-in-One FTP Statements

STATEMENT	PARAMETERS	DESCRIPTION
ftpdelete	*server*, *path*, *file*	Deletes a file on the FTP server
ftpget	*server*, *path*, *remotefile*, *localfile*	Retrieves files from the FTP server
ftpput	*server*, *path*, *localfile*, *remotefile*	Transmits files to the FTP server
ftprename	*server*, *dir*, *oldname*, *newname*	Renames files on the FTP server

Table 7.3 Individual FTP Statements

STATEMENT	PARAMETERS	DESCRIPTION
ftpcd		Changes directory on the FTP server
ftpcd	*path*	Changes directory on the FTP server
ftpclose		Disconnects from an FTP server

continues

Table 7.3 Continued

STATEMENT	PARAMETERS	DESCRIPTION
ftpdelete	[*path*,] *file*	Deletes a file on the FTP server
ftpdir		Returns a list of files from the current directory
ftpdir	*wildcard*	Returns a list of files matching the wildcard specification in the current directory
ftpdir	*directory*, *wildcard*	Returns a list of files matching the wildcard specification in the specified path
ftpexec	*file*	Executes FTP commands from a script file
ftpget	[*path*,] *remotefile* [, *localfile*]	Retrieves a file from the FTP server
ftpidentity	*userid*, *password*	Specifies a user ID and password for the FTP connection
ftplcd	*path*	Changes directory on the local (client) system
ftpls		Returns a list of files from the current directory
ftpls	[*path*,] *wildcard*	Returns a list of files matching the wildcard specification
ftpmd	*path*	Creates a directory on the FTP server
ftpopen	*server* [, *port*]	Connects to an FTP server
ftpput	[*path*,] *localfile* [, *remotefile*]	Transmits a file to the FTP server
ftppwd		Pushes the current FTP directory onto the stack
ftprename	[*path*,] *oldname*, *newname*	Renames a file on the FTP server
ftprmd	[*path*]	Removes a directory from the FTP server
ftptype	*type*	Sets the FTP server to *ASCII* (text) mode or *image* (binary) mode

All of the FTP statements are also available in function form, as shown in Table 7.4. Most of them return a Boolean result that indicates success/failure.

The FTP statements and functions are described individually in the table.

Table 7.4 FTP Functions

FUNCTION	DESCRIPTION
boolean ftpcd( )	Changes the directory on the FTP server
boolean ftpcd(*path*)	Changes the directory on the FTP server
boolean ftpclose( )	Disconnects from an FTP server
boolean ftpdelete([*path*], *file*)	Deletes a file on the FTP server
boolean ftpdir( )	Returns a list of files from the current directory
boolean ftpdir(*wildcard*)	Returns a list of files matching the wildcard specification in the current directory
boolean ftpdir(*directory*, *wildcard*)	Returns a list of files matching the wildcard specification in the specified path
boolean ftpexec(*file*)	Executes FTP commands from a script file
boolean ftpget([*path*,] *remotefile* [, *localfile*])	Retrieves a file from the FTP server
boolean ftpidentity(*userid*, *password*)	Specifies a user ID and password for the FTP connection
boolean ftplcd(*path*)	Changes the directory on the local (client) system
boolean ftpls([*wildcard*])	Returns a list of files matching the wildcard specification in the current directory
boolean ftpls([*path*,] *wildcard*)	Returns a list of files matching the wildcard specification in the specified path
boolean ftpmd(*path*)	Creates a directory on the FTP server
boolean ftpopen(*server* [, *port*])	Connects to an FTP server
boolean ftpput([*path,]* *localfile* [, *remotefile*])	Transmits files to the FTP server
string ftppwd( )	Returns the current FTP directory
boolean ftprename([*path*,] *oldname*, *newname*)	Renames a file on the FTP server
boolean ftprmd([*path*])	Removes a directory from the FTP server
boolean ftptype(*type*)	Sets the FTP server to *ASCII* (text) mode or *image* (binary) mode

All-in-One FTP Operations

There are all-in-one forms of the *ftpget*, *ftpput*, *ftpdir*, and *ftpdelete* statements. Before using any of them, specify a user name and password with the *ftpidentity* statement.

The all-in-one form of *ftpget* requires four parameters: a server name, a path, a local file specification, and a remote file specification. *ftpget* makes a connection, authenticates, retrieves the specified file(s), and disconnects.

```
ftpget server, path, localfiles, remotefiles
```

The following script retrieves all text files from the root directory of a remote FTP server named *fileserver2*.

```
ftpidentity "anonymous", "dklein@my-domain.com"
ftpget "fileserver2", "", "*.txt", "*.txt"
```

The all-in-one form of *ftpput* requires four parameters: A server name, a path, a remote file specification, and a local file specification. *ftpput* makes a connection, authenticates, transmits the specified file(s), and disconnects.

```
ftpput server, path, remotefiles, localfiles
```

The following script sends all document files beginning with "new" to the *newfiles* directory of a remote FTP server named *documents*.

```
ftpidentity "anonymous", "dklein@my-domain.com"
ftpput "documents", "newfiles", "new*.doc", "*.doc"
```

The all-in-one form of *ftpdelete* requires three parameters: a server name, a path, and a file specification. *ftpdelete* makes a connection, authenticates, deletes the specified file(s), and disconnects.

```
ftpdelete server, path, files
```

The following script deletes all files in the temp directory of a remote FTP server named *fileserver*.

```
ftpidentity "anonymous", "dklein@my-domain.com"
ftpdelete "fileserver", "temp", "*.*"
```

The all-in-one form of *ftprename* requires four parameters: a server name, a path, an existing file name, and a replacement file name. *ftprename* makes a connection, authenticates, renames the specified file(s), and disconnects.

```
ftprename server, path, old-file, new-file
```

The following script renames a file in the temp directory of a remote FTP server named *fileserver*.

```
ftpidentity "anonymous", "dklein@my-domain.com"
ftprename "fileserver", "temp", "order.dat", "order.bak"
```

To perform multiple operations on an FTP server, combine the individual FTP statements.

Individual FTP Operations

For full control of an FTP session, the individual FTP statements (or functions) are used. The standard sequence is as follows:

1. Specify a user name and password with the ftpidentity statement.
2. Connect to an FTP server with the ftpopen statement.
3. Perform any combination of desired FTP operations.
4. Disconnect from the FTP server with the *ftpclose* statement.

Multiple FTP Sessions

NQL uses a very simple syntax for FTP operations, such as the statement shown below.

```
ftpget "*.txt"
```

When there is a need to open multiple FTP sessions simultaneously, identifiers are needed to distinguish one file from another. In NQL, these identifiers are known as FTP session IDs. An FTP session ID is a name preceded by a pound sign, such as #*FileServer* or #*BACKUP*. When an FTP session ID is specified, it is the first parameter of a statement or function. For example:

```
ftpget #FileServer, "*.txt"
```

Specifying an FTP Identity

All FTP servers require identification. Although many FTP servers allow anonymous users, it is always necessary to go through the motions of specifying a user name and a password. By convention, anonymous users are to specify the user name “anonymous” and enter their e-mail address as a password.

The *ftpidentity* statement specifies the user name and password to be used in subsequent FTP connections. The parameters are a user name and password. *ftpidentity* is usually the first FTP command to be used in a script.

```
ftpidentity user, password
```

The following script shows how *ftpidentity* is used to specify an anonymous user prior to connecting to an FTP server.

```
ftpidentity "anonymous", "jstalk@my-domain.com"
ftpopen "ftp.ftp-site.com"
    ...
ftpclose
```

Connecting to an FTP Server

The *ftpopen* statement connects to an FTP server and authenticates. The parameters are a server name and an optional port number. If omitted, the port number defaults to 21, the standard port number for FTP.

```
ftpopen server [, port ]
```

If a connection is successfully established and the user name and password are accepted, a success condition results. The following script shows a connection being made to an FTP server and the code to check for its success or failure.

```
ftpidentity "anonymous", "jstalk@my-domain.com"
if ftpopen("ftp.ftp-site.com")
{
    ...perform FTP operations...
    ftpclose
}
else
{
    show "Unable to connect to FTP server"
}
```

Once a successful connection has been established to an FTP server, the remaining FTP statements and functions may be applied in any order desired. To disconnect the session, issue an *ftpclose* statement.

Changing Directories

During an FTP connection, there are two current directories to manage: the directory on the client side (your system which is running the NQL script), and the directory on the remote FTP server.

The *ftplcd* statement changes the local directory on your system, the FTP client. Specify a valid path as a parameter.

```
ftplcd path
```

The *ftpcd* statement changes the directory on the remote system, the FTP server. Specify a valid path as a parameter.

```
ftpcd path
```

The following script makes a connection to an FTP server, then changes both local and remote directories prior to transferring files.

```
ftpidentity "alpha", "2722&sunflower"
ftpopen "ftp.ftp-site.com"
ftplcd "c:\\My Documents\\Music"
ftpcd "music_files"
ftpget "*.mp3"
ftpclose
```

To find out the current working directory on the remote server, use the *ftppwd* (print working directory) statement or function.

```
ftppwd
```

The working directory is pushed onto the stack.

Obtaining a File Directory

There are two NQL statements for retrieving directories from an FTP server: *ftpdir* and *ftpls*. The chief difference between them is that *ftpdir* returns complete directory information for each file (including date and size), but *ftpls* returns only file names.

The *ftpdir* statement returns a complete directory listing from the FTP server. The parameters are an optional path and a required wildcard file specification. If a path is not specified, the current directory on the FTP server is used. The directory list information is pushed onto the stack as a string.

```
ftpdir [path,] wildcard
```

The *ftpls* statement returns a brief directory listing from the FTP server. The parameters are an optional path and a required wildcard file specification. If a path is not specified, the current directory on the FTP server is used. The directory list of filenames is pushed onto the stack as a single string.

```
ftpls [path,] wildcard
```

The following script connects to an FTP server, retrieves a file directory, and saves it as a local file.

```
ftpidentity "anonymous", "cgrant@my-domain.com"
ftpopen "ftp.ftp-site.com"
ftpcd "downloads"
ftpls
save "downloads.txt"
ftpclose
```

Sending Files

The *ftpput* statement (or function) transmits a file to the FTP server. The parameters are a directory path, a local file specification, and a remote file specification. Only the local file specification is required. The path parameter defaults to the current directory ("") and the remote file specification defaults to the same as the local file specification.

```
ftpput [path,] localfiles [, remotefiles]
```

The following script connects to an FTP server and transmits a file to it.

```
ftpidentity "anonymous", "cgrant@my-domain.com"
ftpopen "ftp.ftp-site.com"
ftpput "budget.xls"
ftpclose
```

To transmit multiple files, specify wildcard characters in the filenames. A question mark (?) represents a single character, and an asterisk (*) represents multiple characters. The following script connects to an FTP server and transmits multiple files to it.

```
ftpidentity "anonymous", "cgrant@my-domain.com"
ftpopen "ftp.ftp-site.com"
ftpput "*.xls"
ftpclose
```

The file transfer mode (ASCII or binary) affects the fidelity of the file transfer.

Receiving Files

The *ftpget* statement (or function) receives a file from the FTP server. The parameters are a directory path, a remote file specification, and a local file specification. Only the remote file specification is required. The path parameter defaults to the current directory ("") and the local file specification defaults to the same as the remote file specification.

```
ftpget [path,] remotefiles [, localfiles]
```

The following script connects to an FTP server and receives a file from it.

```
ftpidentity "anonymous", "cgrant@my-domain.com"
ftpopen "ftp.ftp-site.com"
ftpget "budget.xls"
ftpclose
```

To receive multiple files, specify wildcard characters in the filenames. A question mark (?) represents a single character, and an asterisk (*) represents multiple characters. The following script connects to an FTP server and receives multiple files from it.

```
ftpidentity "anonymous", "cgrant@my-domain.com"
ftpopen "ftp.ftp-site.com"
ftpget "*.xls"
ftpclose
```

The file transfer mode (ASCII or binary) affects the fidelity of the file transfer.

Setting the File Transfer Mode

The *ftptype* statement sets the file transfer mode. The parameter is “ASCII” or “binary.”

```
ftptype type
```

The default file transfer mode is binary.

Deleting Files

The *ftpdelete* statement erases a file from the remote server. A remote directory path and a file to delete are specified. The path defaults to the current working directory ("") if omitted.

```
ftpdelete [path,] file
```

To delete multiple files, use the wildcard characters ? (single character) and * (multiple characters). The statement below uses *ftpdelete* to remove all text files from the server.

```
ftpdelete "*.txt"
```

Renaming Files

The *ftprename* statement changes the name of a file on the remote server. The parameters are a path, the original file name, and the new file name. The path defaults to the current working directory ("") if omitted.

```
ftprename [path,] old-file, new-file
```

The following code shows the use of *ftprename*.

```
ftprename "files.txt", "files.bak"
```

Creating and Removing Directories

The *ftpmd* statement or function creates a new disk directory on the server. The parameter is the name of the directory to create.

```
ftpmd path
```

The *ftprmd* statement or function removes a disk directory from the server. The parameter is the name of the directory to remove.

```
ftprmd path
```

The following code removes an existing disk directory then creates a new disk directory.

```
ftprmd "2001_Files"
ftpmd "2002_Files"
```

Executing Scripts

The *ftpexec* statement or function executes an FTP script, which is a text file containing FTP commands. The parameter is the name of a text file that contains FTP commands.

```
ftpexec script-file
```

Each line of the script is read and executed. The script is responsible for including all of the needed statements to connect and authenticate to the FTP server, along with any desired file transfer or directory operations.

Tutorial: FtpReceive

Now that you are familiar with FTP, let's see it in action. In this tutorial, you will create a script that connects to a remote FTP server and downloads files from it. Before proceeding with this tutorial, you'll need access to an FTP server and an account for accessing it. If you don't have such a system available to you locally, you may be able to find a public FTP server on the Internet.

There are five steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the FtpReceive script.
3. Customize the FtpReceive script.
4. Run the FtpReceive script.
5. Understand the FtpReceive script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Specify an FTP identity.
- Connect to an FTP server.

- Receive files from an FTP server.
- Disconnect from an FTP server.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the FtpReceive Script

In the NQL development environment, enter the script shown in Listing 7.1 and save it under the name FtpReceive.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy FtpReceive.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select FtpReceive.nql from the Tutorials\ch07 folder.

```
//FtpReceive - receive files from a remote system

ftpidentity "anonymous", "your.e-mail.id@your.company.com"

if ftpopen("ftp.yourserver.com")
{
    ftplcd "c:\\letters"        //move to the proper local directory
    ftpcd "backup"            //move to the proper remote directory
    ftpget "*.*"            //receive files to the remote system
    ftpclose
}
else
{
    show "Could not access FTP server"
}
end
```

Listing 7.1 FtpReceive script.

At this point, the script in Listing 7.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Customize the FtpReceive Script

Before we can run the FtpReceive script, it needs to be customized for the FTP server you will be accessing. There are four parts of the script that may need to be changed:

1. The *ftpidentity* statement needs to reflect the user ID and password you will be using to access the FTP server. If the FTP server accepts anonymous connections, you can specify "anonymous" for the user ID parameter and an e-mail address for the password parameter. If you were using a user ID of "cgeorge" and a password of "yellow*hat", you would change the *ftpidentity* statement to look like this:

   ```
   ftpidentity "cgeorge", "yellow*hat"
   ```

2. The *ftpopen* function needs to specify the URL (address) of your FTP server. If your FTP server was at ftp2.my-site.com, you would change the *ftpopen* line as follows.

   ```
   if ftpopen("ftp2.my-site.com")
   ```

3. The *ftplcd* statement should be changed to move to the disk directory where you want to download files to. If you wanted to download files to "d:\downloads", the statement would look like this:

   ```
   ftplcd "d:\\downloads"       //move to the proper local directory
   ```

4. The *ftpcd* statement should be changed to move to the disk directory on the FTP server from where you want to download files. If you wanted to download files from "/music/classical", the statement would look like this:

   ```
   ftpcd "/music/classical"     //move to the proper remote directory
   ```

If you want to stay with the default directory on the FTP server, you can omit the *ftpcd* statement altogether.

Step 4: Run the FtpReceive Script

Now, we are ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the Run toolbar button, or pressing F5. When the script completes, there should be files in the local directory specified in your script. If this does not happen, check the following:

- Check for typographical errors in your script.
- Check that the FTP server address and account information are correct.
- Check that the directories you are specifying, both local and remote, exist.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the FtpReceive script run, connecting to an FTP server and downloading files from it.

Step 5: Understand the FtpReceive Script

We now want to make sure we understand every part of the FtpReceive script. The first line is a comment line.

```
//FtpReceive - receive files from a remote system
```

The *ftpidentity* statement declares your user ID and password for accessing the FTP server. This does not cause an action in itself; the user ID and password will be issued when a connection is opened to the server.

```
ftpidentity "anonymous", "your.e-mail.id@my-domain.com"
```

Next, *ftpopen* connects to the FTP server. An *if* statement handles the possibility of success or failure in connecting. If the connection is successful, the *if* statement's code block executes.

```
if ftpopen("ftp.ftp-site.com")
{
    ...
}
else
{
    show "Could not access FTP server"
}
```

Within the code block, *ftplcd* sets the local directory (on your computer) and *ftpcd* sets the remote directory (on the server). Either of these statements could be left out if you are happy with the default working directories.

```
ftplcd "c:\\letters"     //move to the proper local directory
ftpcd "backup"           //move to the proper remote directory
```

Now for the file transfer. The *ftpget* statement retrieves all of the files from the remote server (that are in its current directory).

```
ftpget "*.*"             //receive files to the remote system
```

Our work is complete at this point, and the FTP session can be terminated with an *ftpclose* statement.

```
ftpclose
end
```

At this point, you've seen how to transfer files from a remote system using File Transfer Protocol.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Reverse the direction of the file transfer, uploading files to the server instead of downloading them from the server.
- Write a script to retrieve files from one FTP server and send them to a different FTP server, using a local directory on your computer as a work area.

Chapter Summary

File Transfer Protocol allows files to be transferred easily between computer systems.

The NQL FTP statements include all-in-one statements, which handle all the details of connection and file transfer, and session statements, where individual statements are combined as needed.

An FTP session is established by *ftpidentity*, which specifies a user ID and password, and *ftpopen*, which makes a connection to an FTP server. A session is closed with *ftpclose*. The other FTP statements are used between *ftpopen* and *ftpclose* to perform file and directory operations on the FTP server.

- The local working directory is set with *ftplcd,* and the remote working directory is set with *ftpcd.*
- Files are sent with *ftpput* and received with *ftpget,* both of which accept wildcard file specifications for transferring multiple files.
- Files may be renamed with *ftprename,* and deleted with *ftpdelete.*
- Directory listings may be retrieved with *ftpdir* or *ftpls.*
- Directories are created with *ftpmd* and removed with *ftprmd.*
- Complete FTP scripts can be executed with *ftpexec*.

With FTP support, your scripts can exchange files with remote systems. For information on reading and writing files, refer to Chapter 6, "Working with Files."

CHAPTER 8

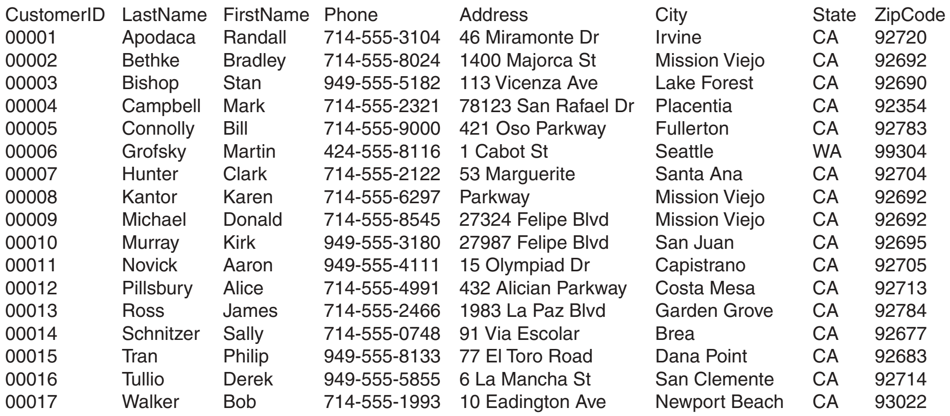
Working with Databases

Databases are the key storage method for large amounts of information. There are many different types of databases available today, ranging from simple PC databases to XML repositories to enterprise-wide relational databases. Network Query Language allows you to work with databases of all types. This chapter provides some background information on databases and their interfaces, followed by coverage of how NQL is used to access databases. The tutorial at the end of the chapter reads movie listings from a database and displays them.

Database Fundamentals

The general premise of a database is simple: It is a place to store information and later recall it. Although disk files do a pretty good job of fulfilling this need on a small scale, they're completely inadequate for dealing with large amounts of information. After nearly 50 years of advancement, today's databases are capable of handling enormous quantities of data reliably and efficiently.

Databases traditionally tend to be organized into *tables*, which consist of *rows* and *columns*. Database tables are similar to tables one encounters in daily life, such as a pricing table or a table of shipping rates. In a typical corporate database, it is not unusual to find tables of prospects, customers, products, orders, inventory, invoices, and payments. Figure 8.1 illustrates a database table.



CustomerID	LastName	FirstName	Phone	Address	City	State	ZipCode
00001	Apodaca	Randall	714-555-3104	46 Miramonte Dr	Irvine	CA	92720
00002	Bethke	Bradley	714-555-8024	1400 Majorca St	Mission Viejo	CA	92692
00003	Bishop	Stan	949-555-5182	113 Vicenza Ave	Lake Forest	CA	92690
00004	Campbell	Mark	714-555-2321	78123 San Rafael Dr	Placentia	CA	92354
00005	Connolly	Bill	714-555-9000	421 Oso Parkway	Fullerton	CA	92783
00006	Grofsky	Martin	424-555-8116	1 Cabot St	Seattle	WA	99304
00007	Hunter	Clark	714-555-2122	53 Marguerite	Santa Ana	CA	92704
00008	Kantor	Karen	714-555-6297	Parkway	Mission Viejo	CA	92692
00009	Michael	Donald	714-555-8545	27324 Felipe Blvd	Mission Viejo	CA	92692
00010	Murray	Kirk	949-555-3180	27987 Felipe Blvd	San Juan	CA	92695
00011	Novick	Aaron	949-555-4111	15 Olympiad Dr	Capistrano	CA	92705
00012	Pillsbury	Alice	714-555-4991	432 Alician Parkway	Costa Mesa	CA	92713
00013	Ross	James	714-555-2466	1983 La Paz Blvd	Garden Grove	CA	92784
00014	Schnitzer	Sally	714-555-0748	91 Via Escolar	Brea	CA	92677
00015	Tran	Philip	949-555-8133	77 El Toro Road	Dana Point	CA	92683
00016	Tullio	Derek	949-555-5855	6 La Mancha St	San Clemente	CA	92714
00017	Walker	Bob	714-555-1993	10 Eadington Ave	Newport Beach	CA	93022

Figure 8.1 Database table.

The columns in a table are also known as *fields*, and the rows are also known as *records*. A field normally has a name and specific attributes, such as a data type and a size. For example, in a table of customers, there might be a *PhoneNumber* field, of type character string, of size 20 characters. Each record in the table is a collection of fields that relate to one entity. For example, the first record in the customer table above contains the customer number, name, phone number, and address of Randall Apodaca, while the second record contains the same information for Bradley Bethke.

Modern databases can hold terabytes or more of information. Well-designed databases guarantee quick access to records with just a handful of disk accesses, no matter how large they are. This is made possible through the use of *indexes* or *keys*. An index is stored information that allows a particular field value to be quickly searched. For example, in a customer table you might have an index for a customer ID so that a customer can be located quickly by their customer number. You might have several indexes, such as the ability to look up by company name, contact name, or phone number. When defining a database, specifying some fields to be *key fields* causes the database software to construct indexes for those fields. It is possible to have indexes that combine several fields.

Structured Query Language

Structured Query Language is a language for querying and modifying databases, a powerful and easy way to access databases that is far simpler than doing discrete database programming. Most databases support the use of SQL, whether or not they support other means of being programmed; SQL has been around since the 1970s and is still prevalent because of its elegance.

The exact grammar of SQL that you may use depends on the database you are accessing. Some databases support a complete ANSI standard for SQL with hundreds of verbs, while others support just the common SELECT, INSERT, UPDATE, and DELETE verbs, and yet others are somewhere between these two extremes.

The SELECT Statement

The workhorse of SQL is the SELECT statement, which selects records from a database matching certain criteria. An example of a SELECT query is shown below. This simple query extracts all fields from the database table named *Customers*.

```
SELECT * FROM Customers
```

A more complex query is the one shown below. This particular query returns only some customers: Those who are not in California and have a positive balance. Instead of extracting all fields of each record, the query requests only the customer number, last name, first name, and phone number fields. The selected information is ordered by state and by phone number. In this case, a WHERE clause specifies conditions for the query, and an ORDER BY clause sorts the results in a certain order.

```
SELECT CustomerNumber, LastName, FirstName, PhoneNumber
FROM Customers
WHERE State <> 'CA' AND Balance > 0.00
ORDER BY State, PhoneNumber
```

As you can see, using SQL is a lot easier than direct programming in a traditional language. That's why SQL is so popular. However, complex SQL statements can be a great deal lengthier than the prior example. It is possible to combine information from several tables (known as a *join*), some functions allow column values to be counted or summed, and records can be grouped in various ways.

For the precise rules for formulating SELECT queries, consult the documentation for the database you are targeting.

The INSERT Statement

In addition to querying databases, modern SQL statements can also modify the information in databases. Database modification statements are known as *action queries* to distinguish them from *select queries*. The INSERT statement is an action query that adds a new record to a database table. The statement specifies both the field names and the field values that constitute a record. The following query adds a new customer record.

```
INSERT Customers
(CustomerNumber,LastName,FirstName,Phone,Address,City,State,Zip)
VALUES (00025,'Jeffries','Jason','631-555-4223','1 Sixth St',
'Islip','NY',11542)
```

For the precise rules for formulating INSERT queries, consult the documentation for the database you are targeting.

The UPDATE Statement

To modify records in a database, the UPDATE statement is employed. UPDATE is similar to SELECT in that it contains a WHERE clause for selecting records, but its purpose is to change one or more fields in the records that match the selection criteria. The following query updates existing customer records to set the Active field to 'N' and the Balance field to 0.00 for customer records that have a balance over a year old.

```
UPDATE Orders SET Active = 'N', Balance = 0.00 WHERE BalanceAge > 365.
```

For the precise rules for formulating UPDATE queries, consult the documentation for the database you are targeting.

The DELETE Statement

To remove records from a database, the DELETE statement is used. DELETE has a WHERE clause like SELECT, but erases records matching the criteria. The statement below deletes all records from the Products file that have a product code of 3.

```
DELETE Products WHERE ProductCode = 3
```

For the precise rules for formulating DELETE queries, consult the documentation for the database you are targeting.

Other SQL Verbs

The databases you access may support only the preceding common SQL verbs, or they could be more extensive. Some databases allow you to perform sophisticated operations, including creating and changing the definitions of tables, backing up databases, and commiting or rolling back transactions.

Powerful databases allow elaborate and sophisticated selection queries. Here's an example of a SELECT query that accesses information from multiple tables.

```
SELECT CategoryName, ProductName
FROM Categories INNER JOIN Products
ON Categories.CategoryID = Products.CategoryID;
```

Deluxe databases may support hundreds of other SQL statements. For example, the following query creates a new database table.

```
create table allfields (f1 varchar, f2 varchar(30), f3 text, f4
varbinary f5, varbinary (30), f6 long varbinary, f7 int, f8 smallint, f9
float, f10 datetime, f11 bit)
```

For the specific SQL queries available to you, you'll need to refer to the documentation for the database you are targeting.

Database Protocols

Although SQL is a useful way to express database operations, how exactly are database operations communicated to a database? This question has received a lot of attention over the years, especially as personal computers and networks have grown. The database you wish to access may not be on your computer at all.

The most successful development in database protocols is the Open Data Base Connection (ODBC). NQL allows you access to databases through ODBC, but also supports three newer protocols: Object Linking and Embedding Database (OLE DB), ActiveX Data Objects (ADO), and Java Database Connectivity (JDBC).

NQL supports ODBC, OLE DB, and ADO in the Windows edition, and it supports JDBC in the Java edition. JDBC supports a number of database interfaces related to ODBC.

ODBC

The ODBC protocol uses TCP/IP to communicate SQL queries to a database and receive the resulting data or acknowledgement. The idea is to use an intermediary program to go between a standard ODBC interface and the specifics of any one database. Each vendor supplies an ODBC driver that faithfully adheres to the standard ODBC database interface and translates those functions into the communication language their database expects. Thus, there is an ODBC driver for Oracle, one for Microsoft SQL Server, one for Microsoft Access, and so on. Figure 8.2 illustrates the ODBC architecture.

Under Windows, ODBC drivers are DLLs. In order to access a database, a Data Source Name (DSN) must be set up in the Control Panel's ODBC applet.

OLE DB

OLE DB is Microsoft's newer approach to universal data access. It is intended to build on the success of ODBC but add additional capabilities to access data sources above and beyond relational databases. Rather than using an ODBC driver, OLE DB relies on the notion of a Provider (a COM object), which provides the underlying access to the data source. Providers can access almost any data source, whether it is a true database or not. Data is returned in columnar format. Providers may or may not support SQL in their implementations. You can access ODBC databases through OLE DB because it is meant to eventually supersede ODBC. OLE DB providers make their presence known through the Windows registry.

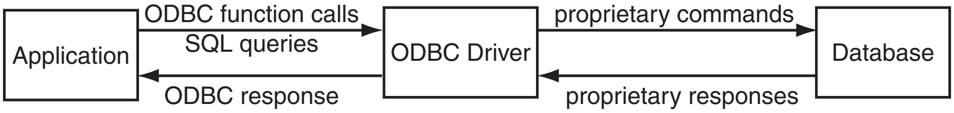


Figure 8.2 ODBC architecture.

ADO

ADO is an easy-to-use encapsulation of OLE DB for specific data sources. Several current popular products use ADO to perform their database connections, including Microsoft's Active Server Pages (ASP). NQL fully supports ADO, and you can specify the same kind of connection strings as you would in ASP.

JDBC

JDBC is a collection of database access methods for platform-independent Java-based applications. There are four database-access methods available in JDBC:

- JDBC-ODBC Bridge (ODBC through a JDBC interface)
- Native API (partly-Java driver)
- JDBC-Net (pure Java driver)
- Native-protocol (pure Java driver)

To access a database, you must specify a complete URL specification that details the driver connection string in this format:

```
jdbc:mySubprotocol:mySubName
```

The *mySubprotocol* and *mySubName* parameters specify the driver you are using. The actual values you will need to specify come from your driver supplier.

NQL's Database Support

NQL allows you to open databases and query them or modify their contents. You may open multiple databases at the same time.

Table 8.1 lists the NQL database statements.

There are function editions of the database statements. The database functions are shown in Table 8.2.

The database statements and functions are described individually below.

Database Sessions

The NQL database operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

1. Optionally, the database protocol is declared with dbprotocol.
2. A database session is opened with opendb.
3. One or more record selections or actions may be issued. For record selections, a select statement is issued, followed by multiple nextrecord statements. For actions, execute is used.
4. The database session is closed with *closedb*.

Table 8.1 Database Statements

STATEMENT	PARAMETERS	DESCRIPTION
addrecord	[*#ID*,] *#Record*	Adds a record to a database
closedb	[*#ID*]	Closes a database
dbfields	[*#ID*,] *dsn*, *table-name* [, *user-ID*, *password*]	Returns the fields in a database table (ODBC)
dbfields	[*#ID*,] *provider*, *table* [, *DSN* [, *user-ID*, *password*]]	Returns the fields in a database table (OLE DB)
dbfields	[*#ID*,] *connection-string*, *table* [, *user-ID*, *password*]	Returns the fields in a database table (ADO)
dbnames		Returns the names of available databases
dbprotocol	*protocol* [, *database-type*]	Specifies database protocol and database type
dbtables	[*#ID*,] *dsn* [, *user-ID*, *password*]	Returns the tables in a database (ODBC)
dbtables	[*#ID*,] *provider* [, *DSN* [, *user-ID, password*]]	Returns the tables in a database (OLE DB)
dbtables	[*#ID*,] *connection-string* [, *user-ID, password*]	Returns the tables in a database (ADO)
execute	[*#ID*,] *query*	Executes an action query
firstrecord	[*#ID*]	Accesses the first record in the current database table
nextrecord	[*#ID*]	Accesses the next record in the current table or query results
opendb	[*#ID*,] *dsn* [, *user-ID, password*]	Opens a database (ODBC)
opendb	[*#ID*,] *provider* [, *dsn|server* [,*user-ID, password*]]	Opens a database (OLE DB)
opendb	[*#ID*,] *connection-string* [, *user-ID, password*]	Opens a database (ADO)
select	[*#ID*,] *query*	Issues a select query against a database
table	[*#ID*,] *table*	Declares a table to be the current table

Table 8.2 Database Functions

FUNCTION	DESCRIPTION
boolean addrecord([#*ID*,] #*Record*)	Adds a record to a database
boolean closedb([#*ID*])	Closes a database
boolean dbfields([#*ID*,] *dsn*, *table-name* [,*user-ID*, *password*])	Returns the fields in a database table (ODBC)
boolean dbfields([#*ID*,] *provider*, *table* [, *DSN* [, *user-ID*, *password*]])	Returns the fields in a database table (OLE DB)
boolean dbfields([#*ID*,] *connection-string*, *table* [, *user-ID*, *password*])	Returns the fields in a database table (ADO)
boolean dbnames( )	Returns the names of available databases
boolean dbprotocol(*protocol* [, *database-type*])	Specifies database protocol and database type
boolean dbtables([#*ID*,] *dsn*, *[userid*,] *[Password*])	Returns the tables in a database (ODBC)
boolean dbtables([#*ID*,] *provider* [, *dsn*|*server* [, *user-ID, password*]])	Returns the tables in a database (OLE DB)
boolean dbtables([#*ID*,] *connection-string* [, *user-ID, password*])	Returns the tables in a database (ADO)
boolean execute([#*ID*,] *query*)	Executes an action query
boolean firstrecord([#*ID*])	Accesses the first record in the current database table
boolean nextrecord([#*ID*])	Accesses the next record in the current table or query results
boolean opendb([#*ID*,] *dsn* [,*user-ID, password*])	Opens a database (ODBC)
boolean opendb([#*ID*,] *Provider* [, *dsn* [, *user-ID, password*]])	Opens a database (OLE DB)
boolean opendb([#*ID*,] *Connection-string* [, *user-ID, password*])	Opens a database (ADO)
boolean select([#*ID*,] *query*)	Issues a select query against a database
boolean table([#*ID*,] *table*)	Declares a table to be the current table

Multiple Database Sessions

NQL uses a very simple syntax for database operations, such as the following line.

```
opendb "customers"
```

When multiple databases need to be opened simultaneously, identifiers are needed to distinguish one database from another. In NQL, these identifiers are known as database session IDs. A database session ID is a name preceded by a pound sign, such as *#cust* or *#Product*. When a session ID is specified, it is the first parameter of a statement or function. For example,

```
opendb #ord, "orders"
opendb #inv, "invoices"
```

Opening a Database

To open a database, the *opendb* statement or function is used. The database is opened using the current database protocol, which defaults to ODBC but can be changed with the *dbprotocol* statement. The *dbprotocol* statement takes one parameter, a string that evaluates to the word *odbc*, *oledb*, *ado* (Windows edition), or *jdbc* (Java edition).

```
dbprotocol protocol
```

The parameters for *opendb* vary depending on the protocol in use.

ODBC

For the default ODBC protocol, the parameters are a DSN and an optional user ID and password.

```
opendb dsn [, user-ID, password]
boolean opendb(dsn [, user-ID, password])
```

The DSN is the ODBC name associated with the database. Under Windows, DSNs are defined using the Control Panel ODBC applet. Some databases require identification, and in these cases the user-ID and password parameters should be specified. The following code opens an ODBC database with the DSN *Orders*.

```
opendb "Orders"
```

If the database cannot be opened, a failure condition results. A success condition indicates that that the database has been successfully opened and database operations may now be performed.

When working with multiple databases, specify a database ID as the first parameter.

```
opendb #contacts, "Contacts"
```

When a script ends, any database sessions still open are automatically closed.

OLE DB

For the OLE DB protocol, the one required parameter is a provider name. If the database resides on a different machine, a second parameter specifies the name of the server. In addition, user-ID and password parameters may be necessary if the database requires authentication.

```
opendb provider [, dsn|server [, user-ID, password]]
boolean opendb(provider [, dsn|server [, user-ID, password]])
```

The *provider* parameter identifies the database type. It may need to be followed with a server parameter if the database resides on a different machine. The following code connects to a SQL Server provider on a remote machine named *Server2*. Provider names are listed in the registry.

```
opendb "SQLOLEDB", "Server2"
```

Since OLE DB can incorporate ODBC, it is possible to open a database with OLE DB that in turn calls upon ODBC to access the database. In these cases, the second parameter is a DSN rather than a server name.

If the target database requires authentication, then a *user-ID* and *password* will also be required. The following code opens a SQL Server provider on a remote system, specifying a user ID and password.

```
opendb "SQLOLEDB", "dbserver", "admin", "pass*7682"
```

If the database cannot be opened, a failure condition results. A success condition indicates that that the database has been successfully opened, and database operations may now be performed.

When working with multiple databases, specify a database ID as the first parameter.

```
opendb #Orders, "SQLOLEDB", "Ordsys
```

When a script ends, any database sessions still open are automatically closed.

ADO

For the ADO protocol, the parameters are a connection string and an optional user-ID and password.

```
opendb connection-string [, user-ID, password]
boolean opendb(connection-string [, user-ID, password])
```

Table 8.3 Sample ADO Connection String Formats

DATABASE	CONNECTION
MS Access	Provider=Microsoft.Jet.OLEDB.4.0;Data Source=physical path to .mdb file
Oracle	Provider=MSDAORA.1;Data Source=path to database on server
MS SQL Server	Provider=SQLOLEDB.1;Data Source=path to database on server
ODBC (local)	Driver={*driver-name*}DBQ=physical path to file
ODBC (remote)	Driver={*driver-name*}SERVER=server

The *connection-string* parameter identifies the database type. A provider name is specified, along with additional information that is specific to the database in question. Some connection string formats are shown in Table 8.3.

If the target database require authentication, then a user ID and password will also be required. The following code opens two ADO databases, the second of which requires user authentication.

```
opendb "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=products.mdb"
opendb "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=prices.mdb",
    "administrator", "charlie"
```

If the database cannot be opened, a failure condition results. A success condition indicates that that the database has been successfully opened, and database operations may now be performed.

When working with multiple databases, specify a database ID as the first parameter.

```
opendb #products, "Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=products.mdb"
```

When a script ends, any database sessions still open are automatically closed.

Closing a Database

Once there are no more operations to perform on a database, it should be closed. The *dbclose* statement closes a database.

```
closedb
```

If a database was opened with a database session ID, specify that ID when closing it.

```
closedb #ID
```

If you forget to close a database, NQL will close all open databases when a script ends.

Querying Databases

Databases are queried with the *select* statement. The parameter is a SQL SELECT query.

```
select select-query
```

The select query can be any select query that conforms to the SQL grammar the database in question supports. The following code issues a select query to return all customer records.

```
select "SELECT * FROM CUSTOMERS"
```

The select-query is issued to the open database. If the query is successful and there is at least one record of data to be returned, a success condition results. If no data is returned, or if there is an error in the query, a failure condition results.

When a select query is successful, variables are automatically set to the field values of the first record. This is called automatic variable mapping. The variables have the same names as the database fields. This makes the database record information immediately available for use. Thus, a query such as *select "SELECT * FROM CUSTOMERS"* might end up creating variables named *CustID*, *Company*, *Street*, *Address*, *City*, *State*, *Zip*, and *Phone*.

To advance to the next record that matches the query, the *nextrecord* statement is used.

```
nextrecord
```

If there is another matching record, its fields are set into variables and a success condition results. If there are no more matching records, a failure condition results. The following code shows the typical sequence for issuing a database select query and iterating through the matching records.

```
select select-query
while
{
    ...do something with variables...
    nextrecord
}
```

To see this in practice, the following code illustrates a select query and record iteration sequence in action. The application here is to scan all customers who have balances greater than $1000.00, then send an email message to each. Note the use of *select*, *while*, and *nextrecord*.

```
opendb "customers"
openmail
select "SELECT CustEmail, CustName FROM Customers WHERE Balance >
1000.00"
while
{
    push "Dear " CustName ",\n\nYour balance is unusually large.
    Please consider catching up your account."
    sendmessage CustEmail, "Account Advice"
    nextrecord
}
closemail
closedb
```

If a database was opened with a database session ID, specify that ID when issuing a *select* statement.

```
select #ID, select-query
```

When you omit the database session ID, NQL operates on the last database referenced in your script.

Accessing a Table

For directly accessing all of the information in a database table, there's an alternative to using a *select* statement. The *table* statement allows you to specify a current table, after which *firstrecord* and *nextrecord* statements may be used to iterate through all of the records in the table.

The *table* statement requires a table name as a parameter. It has no effect other than to identify a specific table as the current one.

```
table table-name
```

Once a current table has been defined, the first record in it can be accessed using the *firstrecord* statement, which does not require any parameters.

```
firstrecord
```

The effect of *firstrecord* is just like *select*: Unless an error occurs or the table is empty, the first record is loaded into variables and a success condition results. You may then employ the *nextrecord* statement to advance to the next record in the table.

```
nextrecord
```

Putting these three statements together, the general sequence for iterating through a complete database table is to declare a current table, access the first record with *firstrecord*, then use a *while* loop to process the record. At the bottom of the *while* loop is a *nextrecord* statement. The following code is an example of this.

```
opendb "products"
table "books"
firstrecord
while
```

```
{
    output Title, ISDN, Author, Publisher
    nextrecord
}
closedb
```

The primary difference between using *table-firstrecord-nextrecord* instead of *select-nextrecord* is that you don't have to formulate a SQL SELECT query. If you don't want to retrieve all fields of all records, using *select* will be more efficient.

If a database was opened with a database session ID, specify that ID when issuing a *table*, *firstrecord* or *nextrecord* statements.

```
opendb #bks, "products"
table #bks, "books"
firstrecord #bks
while
{
    output Title, ISDN, Author, Publisher
    nextrecord #bks
}
closedb #bks
```

When you omit the database session ID, NQL operates on the last database referenced in your script.

Modifying Databases

Databases are modified with the *execute* statement. The parameter is a SQL *action-query*.

```
execute action-query
```

The action query can be any action query that conforms to the SQL grammar the database in question supports. The following code issues an action query to delete old customer records.

```
execute "DELETE * FROM CUSTOMERS WHERE LastYear <= 1995"
```

The action query is issued to the open database. If the query is successful, a success condition results. If there is an error in the query, a failure condition results.

If a database was opened with a database session ID, specify that ID when issuing an *execute* statement.

```
execute #ID, query
```

When you omit the database session ID, NQL operates on the last database referenced in your script.

Adding Records

For adding a new record to a database, there is an alternative to the *execute* statement. The *table* statement declares a current table, the *record* statement declares a record definition, and the *addrecord* statement adds database records.

The *table* statement requires a table name as a parameter. It has no effect other than to identify a specific table as the current one.

```
table table-name
```

The *record* statement list a series of field names and associates them with a record ID, which must begin with a pound sign. A record ID is specified first, followed by an equals sign and a list of field names.

```
record #ID = field1, field2, ... fieldN
```

The *addrecord* statement adds a new record to a database. The parameter is a record ID. The effect of the statement is to construct a record based on the record definition and variable values, then add the record to the database. The following code shows the use of *table*, *record* and *addrecord* to add a new database record.

```
opendb "contacts"
table "prospects"
record #prospect = company, contact, phone, email
company = "ACME Trucking"
contact = "John O'Doule"
phone = "800-555-5652"
email = "jodoule@trucking-site.com"
addrecord #prospect
closedb
```

The primary advantage to using *addrecord* instead of *execute* is that you don't have to compute a SQL INSERT query.

If a database was opened with a database session ID, specify that ID when issuing *table* and *addrecord* statements.

```
opendb #pros, "contacts"
table #pros, "prospects"
record #prospect = company, contact, phone, email
company = "ACME Trucking"
contact = "John O'Doule"
phone = "800-555-5652"
email = "jodoule@trucking-site.com"
addrecord #pros, #prospect
closedb #pros
```

When you omit the database session ID, NQL operates on the last database referenced in your script.

Determining Database Definitions

In addition to querying and modifying databases, it is often useful to discover information about them, such as the names of their tables and fields. NQL has statements and functions that allow your programs to discover the names of known databases, the tables in a database, and the fields in a table. These operations do not require that a database be open.

Discovering Named Databases

To determine the names of known database, the *dbnames* statement is used. No parameters are required.

```
dbnames
```

The effect of *dbnames* is to push a string value onto the stack that contains the name of all defined databases based on the current protocol, each separated by a newline. The *nextline* statement can be used to iterate through the database names.

```
string name
dbnames
while nextline(name)
{
    ..do something with name...
}
```

When using ODBC, the names returned by *dbnames* match the names defined in the Windows Control Panel's ODBC applet.

Discovering the Tables in a Database

To determine the tables that are contained in a database, the *dbtables* statement is used. Parameters are required to identify the target database and are dependent on the database protocol in use. They follow the same parameter rules for *opendb*.

When using the ODBC protocol, specify a DSN and, if necessary, a user ID and password.

```
dbtables dsn [, user-ID, password]
```

When using the OLE DB protocol, specify a provider name. If using ODBC through OLE DB, specify a DSN and, if necessary, a user ID and password.

```
dbtables provider [, DSN [, user-ID, password] ]
```

When using the ADO protocol, specify a connection string and, if necessary, a user ID and password.

```
dbtables connection-string [, user-ID, password]
```

The effect of *dbtables* is to push a string value onto the stack that contains the name of all tables in the specified database, each separated by a newline. The *nextline* statement can be used to iterate through the table names. The following code finds all of the table names in the database named *products*.

```
string table_name
dbtables "products"
while nextline(table_name)
{
    ..do something with table_name...
}
```

Not all database drivers supply table names on request. The ability of *dbtables* to perform is directly based on the support provided by the database driver being accessed.

Discovering the Fields in a Database Table

To determine the fields that are contained in a database table, the *dbfields* statement is used. Parameters are required to identify the target database and table and are dependent on the database protocol in use. They follow the same parameter rules for *opendb* but also specify a table name.

When using the ODBC protocol, specify a DSN, a table name, and, if necessary, a user ID and password.

```
dbtables dsn, table [, user-ID, password]
```

When using the OLE DB protocol, specify a provider name and a table name. If using ODBC through OLE DB, specify a DSN and, if necessary, a user ID and password.

```
dbtables provider, table [, DSN [, user-ID, password] ]
```

When using the ADO protocol, specify a connection string, a table name and, if necessary, a user ID and password.

```
dbtables connection-string, table, [, user-ID, password]
```

The effect of *dbfields* is to push a string value onto the stack that contains the name of all fields in the specified database table, each separated by a newline. The *nextline* statement can be used to iterate through the field names. The following code finds all of the field names for the table named *books* in the database named *products*.

```
string field_name
dbtables "products", "books"
while nextline(fields_name)
{
    ..do something with field_name...
}
```

Not all database drivers supply field names on request. The ability of *dbfields* to perform is directly based on the support provided by the database driver being accessed.

Tutorial: Movie-Reviews

Now that you know the ins and outs of accessing databases in NQL, let's put this knowledge to work in a script. In this tutorial, you will create a script that reads records from a database of movie reviews.

There are five steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the movie-reviews script.
3. Create the movie database.
4. Run the movie-reviews script.
5. Understand the movie-reviews script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Open a database.
- Issue a SQL select query.
- Process a set of records that match a query.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

```
//movie-reviews - reads movie reviews out of an Access database and
displays results
//
//Before running, insure that the dbpath below reflects the location
//of the movies.mdb sample database.

dbpath = "movies.mdb"

dbprotocol "ado"
if opendb("Driver=Microsoft Access Driver (*.mdb); DBQ=" dbpath)
{
     boolean bHaveData = select("select * from movies")
     while bHaveData
     {
          dumpvars
          bHaveData = nextrecord()
     }
}
else
{
     show "Unable to open the database " dbpath
}
closedb
```

Listing 8.1 Movie-Reviews script.

Step 2: Enter the Movie-Reviews Script

In the NQL development environment, enter the script shown in Listing 8.1 and save it under the name movie-reviews.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy movie-reviews.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select movie-reviews.nql from the Tutorials\ch08 folder.

At this point, the script in Listing 8.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Step 3: Create the Movie Database

The companion CD includes a Microsoft Access database for this tutorial, named movies.mdb. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and ensure that moveis.mdb is in the Tutorials\ch08 folder.

In order for the tutorial to be a success, your script and the movies.mdb file need to be in the same disk directory. At this point, you should have a script entered and a database to go with it.

Step 4: Run the Movie-Reviews Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Movie titles and reviews should start to display. If this does not happen, check the following:

- Make sure you have the script and the movies.mdb file in the same directory.
- Check that you have the necessary files for accessing Microsoft Access via ADO on your computer.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the movie-reviews script run, reading and displaying database information. Although we haven't discussed how it works yet, you've seen first-hand the results of an NQL database script that issues select queries and processes the results. In the next step, we'll dissect the script and explain exactly how it works.

Step 5: Understand the Movie-Reviews Script

We now want to make sure we understand every part of the movie-reviews script. The first few lines are comment lines.

```
//movie-reviews - reads movie reviews out of an Access database and
displays results
//
//Before running, insure that the dbpath below reflects the location
//of the movies.mdb sample database.
```

A variable is set named *dbpath*, which will be used shortly to open the database. This can be a relative path, as shown, or a complete file specification.

```
dbpath = "movies.mdb"
```

Next, the database protocol is specified with *dbprotocol*. In this case, we elect to use ADO as the database protocol.

```
dbprotocol "ado"
```

Now it is time to open the database. Since we are using ADO, we specify an ADO connection string, which includes the *dbpath* variable we set earlier. If the database is successfully opened, a code block that queries the database is executed.

```
if opendb("Driver=Microsoft Access Driver (*.mdb); DBQ=" dbpath)
{
     ...
}
else
{
     show "Unable to open the database " dbpath
}
```

Within the code block, the database is queried with a *select*() function. A SQL query is specified that selects all fields of all records of the movies database: *select * from movies*. The result of the select is stored in the *boolean* variable *bHaveData*.

```
boolean bHaveData = select("select * from movies")
```

If the selection is successful and there is at least one record returned, *select*() will have returned true. The fields from the first record are automatically stored in variables. A *while* block is entered to display the matching records.

```
boolean bHaveData = select("select * from movies")
while bHaveData
{
    ...
}
```

Within the while block, a *dumpvars* statement displays all variable values on the screen, a convenient way to display the names and values of the database record just accessed. Then, *bHaveData* is set through a call to *nextrecord*(), which will access the next database record that matches the query. Like *select*(), *nextrecord*() sets variables to reflect the values in the database record. The *while* loop continues until there are no more database records.

```
dumpvars
bHaveData = nextrecord()
```

After the *while* loop exits, *closedb* closes the database.

```
closedb
```

At this point, you've seen NQL's database mechanics at work first-hand.

Further Exercises

You could amend this example in a number of ways. Once you've learned more about NQL from the chapters that follow, you may want to come back to this example and try your hand at extending it. Here are some interesting modifications that can be made:

- Instead of selecting all movie records, select only those that match certain criteria.
- Prompt the user for a movie title and display reviews for that title only.
- Allow the user to enter new information and add records to the movie database.

Chapter Summary

Databases consist of tables, which are composed of rows (records) and columns (fields).

Structured Query Language (SQL) is the de facto standard for accessing databases. The level of SQL supported varies from one database to another, but some common workhorse statements of SQL are always supported. There are two general categories of query: *select* queries and *action* queries.

- The SELECT query selects records from a database table that meet specified criteria.
- The INSERT query adds new records.
- The UPDATE query modifies records that meet specified criteria.
- The DELETE query removes records that meet specified criteria.

NQL allows you to open, query, and update databases of any kind. There are four access methods supported, through which nearly all databases are accessible:

- ODBC, Open Database Connectivity (Windows)
- ADO, ActiveX Data Objects (Windows)
- OLE DB, Object Linking and Embedding Database (Windows)
- JDBC, Java Database Connectivity (Java), which in turn supports four methods.

Database sessions in NQL begin with a declaration of protocol with *dbprotocol*. A database is opened with *opendb*, and closed with *closedb*.

Select queries can be issued to retrieve records that meet specified criteria.

- The *select* statement issues a SQL select query and loads the first matching record.
- The *nextrecord* statement accesses the next matching record.

Action queries can be issued to add, update, and delete records, or to perform other database operations.

- The *execute* statement issues an action query.
- An alternate way to add new records is with the *addrecord* statement.

Lists of available databases, tables, and fields may be retrieved.

- The *dbnames* statement returns the defined ODBC drivers or OLE DB providers.
- The *dbtables* statement returns the tables that are present in a specific database.
- The *dbfields* statement returns the fields that are present in a specific table.

Armed with database access, you are ready to do some powerful work with NQL. You'll find NQL's database support is handy for database maintenance of all kinds. NQL is also useful for converting between databases and other forms of storage, such as Web pages and XML documents.

CHAPTER 9

Working with XML

The Extensible Markup Language (XML) specification has become the standard way to represent content and is widely embraced throughout the computer industry. Connected applications such as bots or agents frequently need to work with XML and convert between it and other formats. Network Query Language includes direct XML capabilities, which include the ability to parse, validate, and generate XML. This chapter begins with background information on XML itself, followed by NQL's XML operations. The tutorial at the end of the chapter reads property information from an XML document and creates a Web page of real estate listings.

XML Fundamentals

The XML specification came about from a desire to standardize the way information is stored throughout the world. Although called a language, XML is not a language in the sense of a programming language like Java or Visual Basic. Rather, XML is a *data language*; that is, a format for representing information. XML can be used to represent just about anything, such as an invoice, a novel, or a musical composition.

XML is related to two other data languages, SGML and HTML, and was heavily influenced by them. Structured Generalized Markup Language (SGML) was an early attempt at a universal language for encoding information. Although SGML included some powerful ideas, such as the ability to define your own tags, it did not catch on worldwide, partly because of its complexity. Nevertheless, SGML is used heavily in certain circles, notably by departments of the U.S. Government (such as the SEC) and the aerospace industry. Even though SGML was never widely embraced, it had many of the right ideas for a data language, and one derivative of SGML did catch on: HyperText Markup Language (HTML), the de facto method for expressing Web pages. Due to the explosive growth of the World Wide Web, there are millions of people who are familiar with HTML. HTML is actually just an application of SGML, where a particular set of tags for Web pages has been defined. One factor behind HTML's success has been that it is fairly easy to pick up and put to use because it is not overly complex.

And this brings us to XML, a data language meant to bring the benefits of SGML without its complexity, in a way that will be familiar to the millions of HTML developers. If you know HTML, XML should be a breeze. What can XML do for you, and why should you use it? Very simply, XML is like HTML except that you can come up with your own tags. Rather than create pages to be viewed, you create data documents that are primarily exchanged between applications and organizations automatically. Listing 9.1 shows an XML representation of a purchase order.

Even if you've never seen XML before, some things should strike you about the preceding code. First of all, it resembles HTML in that there are tags in angle brackets intermingled with text. Secondly, the tags are named after the data they represent. The document is easily read by people but is also easily processed by computer programs.

Because XML is meant to be used internationally, XML documents are expressed in Unicode, a two-byte character set capable of encoding all the world's written symbols. However, XML documents that use American and Western European character sets can be expressed in single-byte characters, through an allowed subset of Unicode known as UTF-8.

XML is sometimes described as a language where you can invent your own tags at will. This is true in a sense, but needs to be tempered with the necessity of cooperating with other parties. It is indeed true that you can create your own schemas for XML documents, and this is one of the powers of the language. It is also true that if you exchange XML documents with another party, you will need to agree on the structure of documents. If your software receives XML documents in an unknown schema, it will likely not be able to do much with them.

XML is evaluated according to whether it is *well-formed* and *valid*. An XML document is considered to be *well-formed* if it properly complies with the syntax rules for XML. For example, a document where a closing angle bracket is missing from a tag is not well-formed. A *valid* XML document is well-formed but is also known to adhere to a specific document structure. For example, if a well-formed XML document is known to adhere to a company standard for XML invoices, it is valid XML. For a document to be valid, it must contain a reference to a Data Type Document (DTD), or template. The DTD is a set of definitions that indicate the elements that can appear in the document, along with rules for how they may be ordered and in what quantities.

Even though XML is less complex than SGML, there is still a fair amount of work involved in processing an XML document. It is easy to generate XML in software but not so easy to read it when all of the rules are taken into account. This is especially so when you consider the work involved in validating an XML document against a DTD

```
<?xml version="1.0"?>
<purchase-order>
  <number>10545</number>
  <company>Demand Billing</company>
  <address>
    <street>367 Wildwood Road</street>
    <city>Ronkonkoma</city>
    <state>NY</state>
    <zip>11779</zip>
  </address>
  <order>
  <items>2</items>
  <item>
    <part-number>40331</part-number>
    <description>envelopes, business size, white</description>
    <price>$14.50</price>
    <qty>100</qty>
    <ext-price>$1450.00</ext-price>
  </item>
  <item>
    <part-number>19244</part-number>
    <description>envelopes, legal size, white</description>
    <price>$19.50</price>
    <qty>10</qty>
    <ext-price>$195.00</ext-price>
  </item>
  <subtotal>$1645.00</subtotal>
  </order>
</purchase-order>
```

Listing 9.1 XML purchase order.

template. To make the job easier, various large players in the computer industry make available XML parsing engines, free of charge.

XPath

XPath is a way of navigating through an XML document through the use of queries. There isn't room in this book to explain the XPath specification, but for a quick orientation, be aware that an XPath query takes this form:

```
axis-name::node-test[predicates]
```

To appreciate the use of XPath, imagine you were dealing with a large XML document that contained numerous entries like the following:

```
<contact category="civilian">
  <fullname>Walters</fullname>
```

```
    <numbers>
      <home>714-555-1698</home>
      <office>714-555-1700</office>
      <fax>714-555-1191</fax>
    </numbers>
  </contact>
```

Now, let's say you are looking for all of Walter's phone numbers. With XPath, you could find this information with a query like the following:

```
/descendant::contact[fullname="Walters"]/child::numbers/child::*
```

You can find the XPath specification online at www.w3.org/TR/xpath.

XML Operations

NQL provides several ways of working with XML. The XML operations can be grouped into two main areas: Simple XML operations and formal XML parsing.

Simple XML Operations

Simple XML operations treat XML as single-byte character strings and don't use an official parsing engine. When working with small or simple XML documents and fragments, the simple XML operations are easy and convenient.

The NQL statements for simple XML operations are listed in Table 9.1.

The function forms of some of the simple XML statements are shown in Table 9.2.

The simple XML statements and functions are described individually in the following sections.

Working with the Input Stream

Depending on how it is called, your scripts may be provided with an input stream. Although the input stream may be in any format, XML is the most common format, and NQL has features to specifically take advantage of XML-formatted input.

There are a number of ways in which an NQL script can inherit an input stream. If you run an NQL script interactively out of the NQL development environment, you may specify an input stream in advance by selecting *View, Inputs* from the menu or by entering the keyboard shortcut Ctrl+I. If an NQL script is being run as a Web application, URL parameters are converted to an XML input stream automatically. If NQL is being called as a component from another language, an input stream may be specified as a property prior to running a script. In addition to these methods, any NQL script is free to create its own input stream.

Retrieving Values from an XML Input Stream

If the input stream is in XML format, values can be extracted from it using the *input* statement. The parameters are one or more variable names.

Table 9.1 Simple XML Statements

STATEMENT	PARAMETERS	DESCRIPTION
getinput		Retrieves the current input stream and places it on the stack
getoutput		Retrieves the current output stream and places it on the stack
input	*var-list*	Extracts XML-encoded values from the input stream
inputrecord	*tag*	Advances to the next instance of the specified tag in the input stream
notify	[*value*]	Passes output stream immediate notification
output	*var-list*	Adds values to the output stream encoded as XML
output	*expr*	Adds to the output stream
outputformat	*type*	Controls the behavior of the output statement
outputrecord	*tag*	Begins another instance of the specified tag in the output stream
setinput		Sets the input stream to the value on the stack
setoutput		Sets the output stream to the value on the stack
unwrap		Converts XML values to variables
wrap	*var-list*	Converts variables to XML values

Table 9.2 Simple XML Functions

FUNCTION	DESCRIPTION
string getinput( )	Retrieves the current input stream and places it on the stack
string getoutput( )	Retrieves the current output stream and places it on the stack
boolean input(*var-list*)	Extracts XML-encoded values from the input stream
boolean inputrecord(*tag*)	Advances to the next instance of the specified tag in the input stream
string unwrap(*value*)	Converts XML values to variables
string wrap(*var*-list)	Converts variables to XML values

```
input var-list
```

For each variable specified, the XML input stream is searched for tags of the same name. If a tag set is found, the data is stored in a variable. Thus, consider the statement below. If the input stream contained *<item>widget</item><amount>$50.00</amount>*, then the variable *item* would be set to *widget* and *amount* to *$50.00*.

```
input item, amount
```

When *input* finds a tag set, the tag set is removed from the input. That way, if there are more sets of data in the input stream with the same tag values, they can be retrieved as well. If the requested information is not found in the input stream, a failure condition results. Suppose the following XML code represents an input stream:

```
<item>
  <name>Apple</name>
  <category>fruit</category>
</item>
<item>
  <name>Banana</name>
  <category>Fruit</category>
</item>
<item>
  <name>Potato</name>
  <category>Vegetable</category>
</item>
```

To easily retrieve the values from this input stream, one could use the following NQL script:

```
while input(name, category)
{
  ...do something with name and category...
}
```

Retrieving Records from an XML Input Stream

When working with more complex XML, the *input* statement may not be sufficient on its own. What if there are varying amounts of elements within other elements? For example, consider the XML document shown in Listing 9.2 which contains multiple recipes, where each recipe might contain a varying number of ingredients.

There will be difficulty in extracting the ingredients from this document if we only have the *input* statement to work with, since we want to be aware of which recipe each ingredient pertains to. For this reason, NQL provides the *inputrecord* statement, which permits two levels of hierarchy. The parameter to *inputrecord* is a string expression that contains a tag name.

```
inputrecord tag
```

```
<?xml version="1.0"?>
<recipes>
  <recipe>
    <recipename>Baked Potato Supreme</recipename>
    <ingredients>
      <ingredient><name>Potatoes</name><qty>4</qty></ingredient>
      <ingredient><name>Parsley</name><qty>pinch</qty></ingredient>
      <ingredient><name>Sour Cream</name><qty>1 cup</qty></ingredient>
      <ingredient><name>Onion Flakes</name><qty>1 oz</qty>
</ingredient>
      <ingredient><name>Bacon Bits</name><qty>2 oz</qty></ingredient>
    </ingredients>
  </recipe>
  <recipe>
    <recipename>Tomato Consumme</recipename>
    <ingredients>
      <ingredient><name>Tomatoes</name><qty>28 oz</qty></ingredient>
      <ingredient><name>Milk</name><qty>2 cups</qty></ingredient>
      <ingredient><name>Basil</name><qty>1 oz</qty></ingredient>
    </ingredients>
  </recipe>
  <recipe>
    <recipename>California Omelette</recipename>
    <ingredients>
      <ingredient><name>Eggs</name><qty>3</qty></ingredient>
      <ingredient><name>Avocado</name><qty>1</qty></ingredient>
      <ingredient><name>Bacon</name><qty>3 strips</qty></ingredient>
      <ingredient><name>Peppers</name><qty>1 oz</qty></ingredient>
      <ingredient><name>Black Olives</name><qty>1/2 cup</qty>
</ingredient>
      <ingredient><name>Tomatoes</name><qty>1/2 cup</qty></ingredient>
      <ingredient><name>Cheddar Cheese</name><qty>1 cup</qty>
</ingredient>
    </ingredients>
  </recipe>
</recipes>
```

Listing 9.2 XML recipes document.

By combining *inputrecord* with *input*, we can now write a script to locate the ingredients for each recipe of the XML input stream shown in Listing 9.2. Here's the code:

```
while inputrecord("recipe")
{
    input recipename
    while input name, qty
    {
        ...do something with name and qty...
    }
}
```

The use of *inputrecord* and *input* will not help you when you have more than two levels of hierarchy in the input stream. In such cases, it is necessary to use formal XML parsing operations to extract data meaningfully.

Copying the Input Stream

It is possible to copy the input stream. You might want to do this for secondary analysis of the input data, or perhaps to store the information in a file or some other medium for record-keeping purposes. The *getinput* statement copies the input stream into a new value pushed onto the stack.

```
getinput
```

Let's consider an example of *getinput*. The following code saves a copy of the input stream to a disk file.

```
getinput
save "input.xml"
```

Setting the Input Stream

It is possible to set the input stream from within a script. You might want to do this after retrieving XML data from someplace, such as a Web site or a database, so that you can use the *input* and *inputrecord* statements. The *setinput* statement sets the input stream to the top value on the stack.

```
setinput
```

Let's consider an example of *setinput*. The following code retrieves XML data from a Web site, sets the script's input stream to be that data, and then extracts information from the input stream.

```
get "http://www.my-web-site.com/orders/new_orders.xml"
setinput
while inputrecord("order")
{
    input CustNo, Customer, Phone
    ...process order...
}
```

Working with the Output Stream

Your scripts can generate an output stream. Although the output stream can be in any format, XML is one of the more common formats, and NQL has features to specifically help generate XML-formatted output.

There are a number of ways in which an NQL script's output can be used. If you run an NQL script interactively out of the NQL development environment, any output generated will appear in an *Output* tab window after the script finishes executing. If an NQL script is being run as a Web application, the output stream is the response sent to a browser. If NQL is being called as a component from another language, the output stream may be retrieved as a property after running a script.

Writing Values to an XML Output Stream

If the output stream is in XML format, values can be added to it using the *output* statement. The parameters are one or more variable names.

```
output var-list
```

For each variable specified, a value is added to the XML output stream reflecting the name and contents of the variable. Thus, consider the statement below. If the variables *item* and *amount* contained *widget* and *$50.00*, respectively, the data *<item>widget </item><amount>$50.00</amount>* would be added to the output stream.

```
output item, amount
```

There's another way to use the *output* statement that gives you direct control over the output: When the argument to output is a string expression rather than a variable list, the data is written to the output stream exactly as specified.

```
output value
```

This second form of *output* is useful for several reasons. First of all, it can be used to generate more elaborate sections of XML beyond the simple variable-at-a-time methodology just discussed. Secondly, it allows *output* to generate non-XML forms of output, such as HTML. The following code shows the use of the second form of *output* to generate HTML.

```
output "<TD>" price[orders] "</TD><TD>" tax[orders] </TD>"
```

Both forms of *output* are cumulative, meaning you call them repeatedly as you build your output stream.

Writing Records to an XML Output Stream

When working with more complex XML, it is sometimes desirable to nest one set of XML tags within an outer set of tags to form records. For this reason, NQL includes the *outputrecord* statement, which permits two levels of hierarchy. The parameter to *outputrecord* is a string expression that contains a tag name.

```
outputrecord tag
```

By combining *outputrecord* with *output*, we can now write a script to generate records of XML tags. The following code shows the use of *outputrecord*.

```
outputrecord "store"
location = "Montauk, NY"
phone = "631-555-7566"
output location, phone

outputrecord "store"
location = "Riverhead, NY"
phone = "631-555-7187"
output location, phone

outputrecord "store"
location = "Sag Harbor, NY"
phone = "631-555-3300"
output location, phone
```

The above script will generate the XML shown below. Notice that *outputrecord* knows to put the starting and ending tags in the right places in the XML output stream, even though you only place a single *outputrecord* before your *output* statements. It's also perfectly legal to have multiple *output* statements between the *outputrecord* statements.

```
<store>
  <location>Montauk, NY<location>
  <phone>631-555-7566</phone>
</store>
<store>
  <location>Riverhead, NY</location
  <phone>631-555-7187</phone>
</store>
<store>
  <location>Sag Harbor, NY</location>
  <phone>631-555-3300</phone>
</store>
```

The use of *outputrecord* and *output* will not help you when you have more than two levels of hierarchy in the output stream. In such cases, it is necessary to do more of the work directly in your program.

Copying the Output Stream

It is possible to copy the output stream. You might want to do this for secondary analysis of the output data, or perhaps to store the information in a file or some other medium for record-keeping purposes. The *getoutput* statement copies the output stream into a new value pushed onto the stack.

```
getoutput
```

Let's consider an example of *getoutput*. The following code saves a copy of the output stream to a disk file.

```
...
getoutput
save "output.xml"
```

Setting the Output Stream

It is possible to set the output stream from within a script. You might want to do this after retrieving XML data from someplace, such as a Web site or a database. The *setoutput* statement sets the output stream to the top value on the stack.

```
setoutput
```

Let's consider an example of *setoutput*. The following code retrieves XML data from a Web site and sets the script's output stream to be that data.

```
get "http://www.my-web.site.com/orders/new_orders.xml"
setoutput
```

Notification and Real-time Output

When your scripts create output streams, the output stream is normally generated piecemeal fashion as your script executes *output* and *outputrecord* statements. When the script finishes execution, the output stream is complete and can then be accessed by some other application. For real-time applications, this batch behavior of the output stream is unsatisfactory. NQL provides an alternative method of generating output streams in real-time with the *notify* statement.

The purpose of *notify* is to notify an external process and send it some data. For an example of where you might do this, imagine you are creating a shopping bot in which a master program calls one or more NQL scripts to search for specific items, such as books from an online book site. One way to implement such a solution would be to have the NQL script find all of the matching books and then pass its output stream to the master program when finished. However, this might prove unsatisfactory to the user because no results would be seen until all of the books had been found. With the use of *notify*, each book result could be passed to the master program as soon as it was found. This would allow results to be shown to the user without delay.

The *notify* statement takes one parameter, which is optional: Data to include with the notification. If no parameter is specified, the current output stream is passed as the notification data.

```
notify [data]
```

How does *notify* interact with another application? The application must respond to an event or set up what is known as a callback function. If you are calling NQL as an ActiveX component, from a language like Visual Basic, there is a *Notify* event that is fired whenever the running NQL script executes a *notify* statement. If you are calling NQL as a DLL from a lower-level language such as C++, you can specify the address of a callback function as a property prior to executing a script.

Wrapping and Unwrapping Values

So far, most of the simple XML statements and functions have related to input and output streams. Network Query Language also contains two simple XML statements that convert between variables and XML. The conversion of variables to XML is known as *wrapping* in NQL parlance, and the conversion of XML to variables is known as *unwrapping*.

To convert one or more variables to XML, the *wrap* statement is used. The parameters are a list of variables.

```
wrap var-list
```

The result of *wrap* is that an XML string is pushed onto the stack which contains the specified variable values, formatted as tags. Like the *output* statement, *wrap* creates tag names based on the variable name and data based on the variable's contents. Thus, the statement *wrap x* would generate *<x>5</x>* if the variable *x* happened to contain *5*. The following code shows the use of *wrap*.

```
typeface = "Times Roman"
typesize = "10"
leading = "12"
kerning = "track"
wrap typeface, typesize, leading, kerning
pop XML
```

wrap's alter-ego is *unwrap*, which takes an XML value off of the stack and creates variables from it. Each tag in the XML code becomes a variable of that name. The following code shows the use of *unwrap*.

```
push "<phone>800-555-4141</phone><email>jsmith@anytown.com</email>"
unwrap
show phone, email
```

XML Parsing Operations

For formal work with XML documents, NQL provides powerful statements for parsing, validating, and querying XML. NQL uses the Microsoft XML parser to perform this work in the Windows edition of NQL, and uses the Sun or Apache XML parser to perform this work in the Java edition of NQL. Table 9.3 shows the XML parsing statements.

There are function versions of the XML parsing statements. The XML parsing functions are shown in Table 9.4.

The XML parsing statements and functions are described individually in the following tables.

XML Sessions

The NQL XML parsing operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

Table 9.3 XML Parsing Statements

STATEMENT	PARAMETERS	DESCRIPTION
closenode	[#*ID*]	Releases a level of focus in the XML document
closexml	[#*ID*]	Terminates an XML parser session
firstnode	[#*ID*,] *node-name*	Accesses an XML node by name
getnode	[#*ID*,] *var-list*	Retrieves element data from a node using dotted notation
getnodexpath	[#*ID*,] *xpath*, *var-list*	Retrieves element data that matches an XPath query
nextnode	[#*ID*,] *node-name*	Adds values to the output stream encoded as XML
openxml	[[#*ID*,] *url*]	Opens an XML parser session

Table 9.4 XML Parsing Functions

FUNCTION	DESCRIPTION
bool closenode([#*ID*])	Releases a level of focus in the XML document
bool closexml([#*ID*])	Terminates an XML parser session
bool firstnode([#*ID*,] *node-name*)	Accesses an XML node by name
bool getnode([#*ID*,] *var-list*)	Retrieves element data from a node using dotted notation
bool getnodexpath([#*ID*,] *xpath*, *var-list*)	Retrieves element data that matches an XPath query
bool nextnode([#*ID*,] *node-name*)	Adds values to the output stream encoded as XML
bool openxml([[#*ID*,] *url*])	Opens an XML parser session

1. An XML parsing session is opened with openxml.
2. The XML may be traversed, searched, and extracted using any combinations desired of firstnode, getnode, getnodexpath, nextnode, and closenode.
3. The XML parsing session is closed with *closexml*.

Multiple XML Parser Sessions

NQL uses a very simple syntax for XML parsing operations, such as the following line:

```
openxml URL
```

When it is necessary to open multiple XML documents simultaneously, identifiers are needed to distinguish one XML document from another. In NQL, these identifiers are known as XML session IDs. An XML session ID is a name preceded by a pound sign, such as #*invoice* or #*Product*. When a session ID is specified, it is the first parameter of a statement or function. For example:

```
openxml #doc1, URL
```

Opening an XML Parser Session

An XML parser session is initiated with the *openxml* statement. The parameter is the URL of an XML document. If this parameter is omitted, the XML document is taken from the top value on the stack.

```
openxml [url]
```

NQL launches the NQL parser for your operating system, retrieves the specified document, and opens it in the parser. If the XML document makes reference to a Data Type Document (DTD) template, it is also validated. If all of this occurs successfully, a success condition results. The following code opens an XML parsing session and checks for error.

```
openxml "http://www.my-docs.com/manifest.xml"
else
{
    show "Unable to open and validate XML document"
    end
}
```

When working with multiple XML documents, specify an XML session ID as the first parameter.

```
openxml #manifest, "http://www.my-docs.com/manifest.xml"
```

Closing an XML Parser Session

An XML parser session is terminated with the *closexml* statement. Any resources consumed by the parser session are freed.

```
closexml
```

If a parser session was opened with an XML session ID, specify that ID when closing it.

```
closexml #manifest
```

If you forget to close a parser session, NQL will close all open parser sessions when a script ends.

Finding a Node

The elements in an XML document are known as nodes. When an XML document is opened, the *focus* is the entire document. The *firstnode* statement locates a node and restricts the focus to that node. The parameter to *firstnode* is a value that specifies one or more element names, separated by dots, such as *price* or *purchase-order.items.item.qty*.

```
firstnode node-name
```

If the specified node is located, the focus in the XML document is set to that node and a success condition results. You can then use *getnode* or *getnodexpath* to retrieve element values from the node with focus. If the node can't be located, a failure condition results. The following code shows the use of *firstnode* to access an item in a purchase order document.

```
openxml URL
firstnode "item"
getnode description, price
...
closexml
```

If a parser session was opened with an XML session ID, specify the same ID when using *firstnode*.

```
firstnode #po, "item"
```

Finding the Next Node

To advance to the next node in a document, *nextnode* is used. It works like *firstnode*, but advances to the next matching node rather than starting at the beginning of the document. The parameter to *nextnode* is a value that specifies one or more element names, separated by dots, such as *price* or *purchase-order.items.item.qty*.

```
nextnode node-name
```

If the specified node is located, the focus in the XML document is set to that node and a success condition results. You can then use *getnode* or *getnodexpath* to retrieve element values from the node with focus. If the node can't be located, a failure condition results. The following code shows the use of *nextnode* to access each item in a purchase order document.

```
openxml URL
firstnode "item"
while
{
    getnode description, price
    ...do something with description and price....
    nextnode "item"
}
closexml
```

If a parser session was opened with an XML session ID, specify the same ID when using *nextnode*.

```
nextnode #po, "item"
```

Closing a Node

When you set a restricted focus in an XML document (with *firstnode* or *nextnode*), you may want to eventually release that focus. The *closenode* statement releases the current focus in an XML document and reverts to the previous focus.

```
closenode
```

The XML parsing operations allow you to have nested levels of focus, which means you can combine *firstnode*, *nextnode*, and *closenode* to work with various parts of an XML document in a hierarchy. For example, the following code shows the use of these statements to walk through all item nodes within all *purchase-order.item* nodes. Note the use of *closenode* to move from the inner level of focus to the outer level of focus.

```
openxml "http://www.my-comp.com/biz/neworders.xml"
firstnode "purchase-order.items"
while
{
    firstnode "item"
    while
    {
        getnode qty, partnum, unitprice, extprice
        nextnode "item"
    }
    closenode
    nextnode "purchase-order.items"
}
closenode
closexml
```

If a parser session was opened with an XML session ID, specify the same ID when using *closenode*.

```
closenode #po
```

Retrieving Node Data

To retrieve data from an XML document, the *getnode* statement is used. The parameters are one or more variable names.

```
getnode var-list
```

For each variable specified, the current focus of the XML document searches for tags of the same name. If found, the variable is set to the data value. If any of the values are not found in the XML, a failure condition results. The following code shows the use of *getnode* to retrieve description and price values from the item node of an XML document.

```
openxml URL
firstnode "item"
getnode description, price
...
closexml
```

If a parser session was opened with an XML session ID, specify that ID when using *getnode*.

```
getnode #po, description, price
```

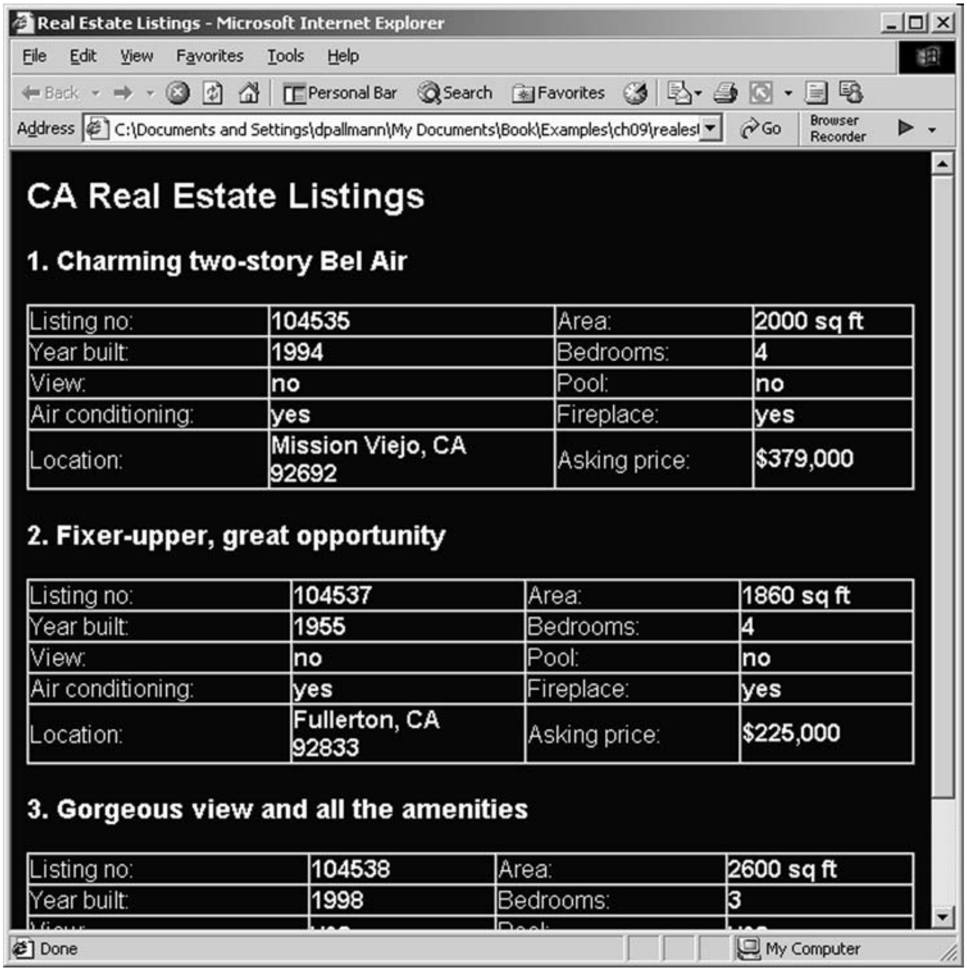
Retrieving Node Data Using an XPath Query

Another way to retrieve data from an XML document is through an XPath query, for which the *getnodexpath* statement is used. The parameters are an XPath query and one or more variable names.

```
getnode xpath [, var-list]
```

If no variable list is specified, the data extracted from the query is pushed onto the stack as a single string. If a variable list is specified, the data resulting from the query are stored in the specified variables. A success condition results if the query is successful. The following code shows the use of *getnodexpath* to retrieve description and price values from the item node of an XML document.

```
openxml URL
getnodexpath '/descendant::name/child::*', sLast, sFirst
...
closexml
```


Real Estate Listings - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address C:\Documents and Settings\dpallmann\My Documents\Book\Examples\ch09\realest
Go
Browser Recorder
Done
My Computer

CA Real Estate Listings

1. Charming two-story Bel Air

Listing no:	104535	Area:	2000 sq ft
Year built:	1994	Bedrooms:	4
View:	no	Pool:	no
Air conditioning:	yes	Fireplace:	yes
Location:	Mission Viejo, CA 92692	Asking price:	$379,000

2. Fixer-upper, great opportunity

Listing no:	104537	Area:	1860 sq ft
Year built:	1955	Bedrooms:	4
View:	no	Pool:	no
Air conditioning:	yes	Fireplace:	yes
Location:	Fullerton, CA 92833	Asking price:	$225,000

3. Gorgeous view and all the amenities

Listing no:	104538	Area:	2600 sq ft
Year built:	1998	Bedrooms:	3

Figure 9.1 Real Estate Web Page.

If a parser session was opened with an XML session ID, specify that ID when using *getnode*.

```
getnodexpath #license, '/descendant::name/child::*', sLast, sFirst
```

Tutorial: RealEstate

Now that you know how to work with XML, let's create an XML-aware NQL script. In this tutorial, you will create a script that reads an XML file of real estate listings. A real estate Web page will be created from the XML data, shown in Figure 9.1.

There are five steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the RealEstate script.
3. Set up the XML data.
4. Run the RealEstate script.
5. Understand the RealEstate script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Open, parse, and close an XML document.
- Locate and extract specific objects in an XML document.
- Generate a Web page.

Let’s begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you’re ready to enter the tutorial script.

Step 2: Enter the RealEstate Script

In the NQL development environment, enter the script shown in Listing 9.3 and save it under the name RealEstate.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy RealEstate.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book’s tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select RealEstate.nql from the Tutorials\ch09 folder.

At this point, the script in Listing 9.3 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD.

Step 3: Set Up the XML Data

The companion CD includes an XML document for this tutorial, named realestate.xml. If you have installed the companion CD on your system, you will have all of the book’s tutorial materials in a \Tutorials directory on your hard drive. Using Windows Explorer, ensure that realestate.xml is in the Tutorials\ch09 folder.

```
//realestate - reads real estate listings from an XML document
//              and generates a web page

string StateCode = "CA"

int no = 0

if !openxml("realestate.xml")
{
    show "Could not open XML document"
    end
}

create "realestate.htm"
write "<HTML><HEAD><TITLE>Real Estate Listings</TITLE>"
write "</HEAD>\r\n"
write "<BODY BGCOLOR=#000033 TEXT=#FFFFFF>\r\n"
write "<FONT FACE=Arial>\r\n”
write "<H2>{StateCode} Real Estate Listings</H2>\r\n"

boolean bHaveProperty = firstnode("property")
while bHaveProperty
{
    getnode listing, desc, area, year, bedrooms, view, pool,
        airconditioning, fireplace, city, state, zip, price
    if state==StateCode
    {
        no += 1
        write "<P><H3>{no}. {desc}</H3>\r\n"
        write "<FONT FACE=Arial>\r\n"
        write "<TABLE WIDTH=600 BORDER=1 CELLSPACING=0 CELLPADDING=
0>\r\n"
        write "<TR><TD>Listing no:</TD><TD><B>{listing}</B></TD>"
        write "<TD>Area:</TD><TD><B>{area}</B></TD></TR>\r\n"
        write "<TR><TD>Year built:</TD><TD><B>{year}</B></TD>"
        write "<TD>Bedrooms:</TD><TD><B>{bedrooms}</B></TD></TR>\r\n"
        write "<TR><TD>View:</TD><TD><B>{view}</B></TD>"
        write "<TD>Pool:</TD><TD><B>{pool}</B></TD></TR>\r\n"
        write "<TR><TD>Air conditioning:</TD><TD><B>{airconditioning}
</B></TD>"
        write "<TD>Fireplace:</TD><TD><B>{fireplace}</B></TD></TR>
\r\n"
        write "<TR><TD>Location:</TD><TD><B>{city}, {state}<BR>{zip}
</B></TD>"
        write "<TD>Asking price:</TD><TD><B>{price}</B></TD></TR>\r\n"
        write "</FONT>\r\n"
        write "</TABLE>\r\n"
```

Listing 9.3 RealEstate script.

```
    }
    bHaveProperty = nextnode("property")
}

closexml

write "</FONT>\r\n"
write "</BODY>\r\n"
write "</HTML>\r\n"
close
opendoc "realestate.htm"
end
```

Listing 9.3 RealEstate script (continued).

The XML document is shown in Listing 9.4. Feel free to view the XML document in a browser, or open it in WordPad to change it around. This particular XML document is in Unicode and was created with WordPad.

In order for the tutorial to be a success, your script and the realestate.xml file need to be in the same disk directory. At this point, you should have a script entered and an XML document to go with it.

Step 4: Run the RealEstate Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. A Web page should open up on your desktop that resembles Figure 9.1. If this does not happen, check the following:

- Check your script for typographical errors.
- Make sure you have the script and the realestate.xml file in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the RealEstate script run, reading information from an XML document and generating a Web page from it. Although we haven't discussed how it works yet, you've witnessed XML being read and transformed into another format by NQL. In the next step, we'll dissect the script and explain exactly how it works.

Step 5: Understand the RealEstate Script

We now want to make sure we understand every part of the RealEstate script. The first two lines are comment lines.

```
//realestate - reads real estate listings from an XML document
//             and generates a web page
```

```
<?xml version="1.0" ?>
<real-estate>
  <property>
    <listing>104535</listing>
    <desc>Charming two-story Bel Air</desc>
    <area>2000 sq ft</area>
    <year>1994</year>
    <bedrooms>4</bedrooms>
    <view>no</view>
    <pool>no</pool>
    <airconditioning>yes</airconditioning>
    <fireplace>yes</fireplace>
    <city>Mission Viejo</city>
    <state>CA</state>
    <zip>92692</zip>
    <price>$379,000</price>
  </property>
  <property>
    <listing>104537</listing>
    <desc>Fixer-upper, great opportunity</desc>
    <area>1860 sq ft</area>
    <year>1955</year>
    <bedrooms>4</bedrooms>
    <view>no</view>
    <pool>no</pool>
    <airconditioning>yes</airconditioning>
    <fireplace>yes</fireplace>
    <city>Fullerton</city>
    <state>CA</state>
    <zip>92833</zip>
    <price>$225,000</price>
  </property>
  <property>
    <listing>104538</listing>
    <desc>Gorgeous view and all the amenities</desc>
    <area>2600 sq ft</area>
    <year>1998</year>
    <bedrooms>3</bedrooms>
    <view>yes</view>
    <pool>yes</pool>
    <airconditioning>yes</airconditioning>
    <fireplace>yes</fireplace>
    <city>Irvine</city>
    <state>CA</state>
    <zip>92720</zip>
    <price>$450,000</price>
  </property>
```

Listing 9.4 Real Estate XML document.

```
  <property>
    <listing>104539</listing>
    <desc>Near-lakeside property</desc>
    <area>2200 sq ft</area>
    <year>1965</year>
    <bedrooms>4</bedrooms>
    <view>no</view>
    <pool>no</pool>
    <airconditioning>no</airconditioning>
    <fireplace>yes</fireplace>
    <city>Ronkonkoma</city>
    <state>NY</state>
    <zip>11779</zip>
    <price>$325,000</price>
  </property>
  <property>
    <listing>104541</listing>
    <desc>Country surroundings</desc>
    <area>1900 sq ft</area>
    <year>1992</year>
    <bedrooms>3</bedrooms>
    <view>no</view>
    <pool>no</pool>
    <airconditioning>yes</airconditioning>
    <fireplace>yes</fireplace>
    <city>Patchogue</city>
    <state>NY</state>
    <zip>11772</zip>
    <price>$340,000</price>
  </property>
  <property>
    <listing>104544</listing>
    <desc>A bargain home conveniently located</desc>
    <area>1700 sq ft</area>
    <year>1975</year>
    <bedrooms>3</bedrooms>
    <view>no</view>
    <pool>no</pool>
    <airconditioning>no</airconditioning>
    <fireplace>no</fireplace>
    <city>Farmingdale</city>
    <state>NY</state>
    <zip>11735</zip>
    <price>$220,000</price>
  </property>
</real-estate>
```

Listing 9.4 Real Estate XML document (continued).

A string variable, *StateCode*, is declared and assigned to a two-letter state code (CA shown here). The script will show only listings from the specified state. You can change the state code to NY or any other state in the data and a different set of properties will be listed in the resulting Web page.

```
string StateCode = "CA"
```

A second variable, *no*, is initialized to zero. The no variable will be used to count the properties as we list them so that we can number the results.

```
int no = 0
```

The XML document is now opened with *openxml*, which begins an XML parsing session. If the document is not found or does not contain valid XML, an error message is displayed and the script ends.

```
if !openxml("realestate.xml")
{
    show "Could not open XML document"
    end
}
```

With the XML document open successfully, we can begin building the Web page. The *create* statement creates a new HTML file. The *write* statements write initial HTML to the file.

```
create "realestate.htm"
write "<HTML><HEAD><TITLE>Real Estate Listings</TITLE>"
write "</HEAD>\r\n"
write "<BODY BGCOLOR=#000033 TEXT=#FFFFFF>\r\n"
write "<FONT FACE=Arial>\r\n"
write "<H2>{StateCode} Real Estate Listings</H2>\r\n"
```

The script seeks to find property records in the XML. A *firstnode* function locates the first instance of <property> and </property> tags, returning true if an instance was found. The true or false result is stored in a Boolean variable named *bHaveProperty*.

```
boolean bHaveProperty = firstnode("property")
```

A *while* loop iterates through each property record. *bHaveProperty* is set anew at the bottom of the while loop via a *nextnode* function. The loop will expire when there are no more property records.

```
while bHaveProperty
{
    ...
    bHaveProperty = nextnode("property")
}
```

Within the *while* loop, a *getnode* statement extracts elements from within the current focus (a property record). The *listing*, *desc*, *area*, *year*, *bedrooms*, *view*, *pool*, *airconditioning*, *fireplace*, *city*, *state*, *zip*, and *price* items are all subordinate parts of *property*. Variables with the same names are set with values from the current *property* item.

```
getnode listing, desc, area, year, bedrooms, view, pool,
    airconditioning, fireplace, city, state, zip, price
```

Next, the extracted state code (in the variable *state*) is compared to the target state (in the variable *StateCode*). If they are identical, an *if* block executes. In the *if* block, the listing number is incremented by one, and *write* statements generate a table of data in the HTML output file.

```
if state==StateCode
{
    no += 1
    write "<P><H3>{no}. {desc}</H3>\r\n"
    write "<FONT FACE=Arial>\r\n"
    write "<TABLE WIDTH=600 BORDER=1 CELLSPACING=0 CELLPADDING=0>\r\n"
    write "<TR><TD>Listing no:</TD><TD><B>{listing}</B></TD>"
    write "<TD>Area:</TD><TD><B>{area}</B></TD></TR>\r\n"
    write "<TR><TD>Year built:</TD><TD><B>{year}</B></TD>"
    write "<TD>Bedrooms:</TD><TD><B>{bedrooms}</B></TD></TR>\r\n"
    write "<TR><TD>View:</TD><TD><B>{view}</B></TD>"
    write "<TD>Pool:</TD><TD><B>{pool}</B></TD></TR>\r\n"
    write "<TR><TD>Air conditioning:</TD><TD><B>{airconditioning}</B>
</TD>"
    write "<TD>Fireplace:</TD><TD><B>{fireplace}</B></TD></TR>\r\n"
    write "<TR><TD>Location:</TD><TD><B>{city}, {state}<BR>{zip}</B></TD>"
    write "<TD>Asking price:</TD><TD><B>{price}</B></TD></TR>\r\n"
    write "</FONT>\r\n"
    write "</TABLE>\r\n"
```

At the bottom of the *while* loop, the next property is accessed. The *nextnode* function will return false if there are no more *property* items in the XML.

```
    bHaveProperty = nextnode("property")
}
```

Since we are finished reading through the XML, the XML parsing session can be terminated with *closexml*.

```
closexml
```

The closing HTML tags are now written to the output file, and the file is closed.

```
write "</FONT>\r\n"
write "</BODY>\r\n"
write "</HTML>\r\n"
close
```

The final task is to open the results page on the desktop, which is accomplished with *opendoc*.

```
opendoc "realestate.htm"
end
```

At this point, you've seen an NQL script parse XML using a parser session.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Add additional parameters besides *state* for the script to use in deciding which properties to list.
- Use an XPath query instead of direct program logic to find the properties matching the desired conditions.

Chapter Summary

Network Query Language has the capability to parse, validate, and generate XML.

- The *wrap* statement converts variables to XML.
- The *unwrap* statement converts XML to variables.

XML is similar to HTML but allows custom tags to be defined: Documents can reference DTD templates, which software can use to ensure that documents are properly structured. XML documents are officially Unicode, consisting of wide two-byte characters; but a single-character format, UTF-8, allows ANSI text files to serve as XML documents.

Because XML is meant to be used internationally, XML documents are expressed in Unicode, a two-byte character set capable of encoding all the world's written symbols. However, XML documents that use American and Western European character sets can be expressed in single-byte characters, through an allowed subset of Unicode known as UTF-8.

An NQL script has an input stream, which is usually in XML format. Operations for working with the input stream include the following:

- The input stream can be entered interactively from the NQL development environment.
- The *input* statement retrieves XML values from the input stream and assigns them to variables.
- The *inputrecord* statement focuses on a section of an XML stream.

- The *getinput* statement copies the input stream and pushes it onto the stack.
- The *setinput* statement sets a new input stream from the top value on the stack.

An NQL script also has an output stream. Operations for working with the output stream include the following:

- The *output* statement adds data to the output stream in XML format.
- The *outputrecord* statement creates section tags around XML data in the output stream.
- The *getoutput* statement makes a copy of the output stream on the stack.
- The *setoutput* statement sets a new output stream from the top value on the stack.
- The *notify* statement performs real-time output and notification.

In addition to primitive XML operations, NQL also supports XML parsing sessions, powered by an XML engine.

- The *openxml* statement opens a parsing session.
- The *firstnode* statement locates the first section of XML by name and sets it as the focus. You can nest the use of *firstnode*.
- The *getnode* statement extracts values from within the current section of focus.
- The *getnodexpath* statement retrieves element data that matches an XPath query.
- The *nextnode* statement advances to the next section of XML by name and sets it as the focus.
- The *closenode* statement closes the current nested area of focus in the XML.
- The *closexml* statement closes a parsing session.

XML is a universal way of representing content. You now know how to work with XML in Network Query Language. Combined with the other capabilities described in the rest of this book, you can easily create scripts that convert to and from XML and other data formats.

CHAPTER 10

Accessing Web Sites

The World Wide Web plays a prominent role in today's era of connected applications. Network Query Language provides facilities for accessing Web pages, crawling entire Web sites, and downloading files. This chapter begins with some background on how the Web works, then describes NQL's two methods for accessing Web pages, Web primitives and the controlled browser. Finally, site crawling and file downloading are covered. The tutorial at the end of the chapter submits a search to a book retail Web site and captures the results.

The World Wide Web

The World Wide Web is a network of networks, in which millions of computers participate at any given moment. A number of protocols can be used on the Web, but the primary one is HyperText Transfer Protocol (HTTP). The Web works on a simple client/server principal, illustrated by Figure 10.1.

Typically, HTTP clients are Web browsers controlled by interactive users, but other kinds of programs such as intelligent agents can also be clients. Clients issue a Web address, also known as a Uniform Resource Locator (URL). A URL includes enough information to locate a Web server as well as an indication of the specific resource (information) desired. For example, the URL *www.my-university.com/assignments/cs101*

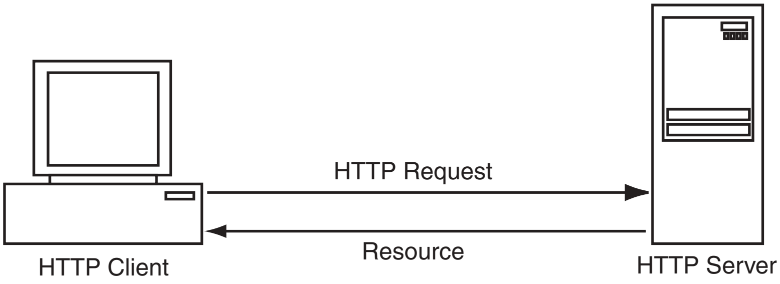


Figure 10.1 HTTP Client/Server Model.

.htm specifies both a Web server (www.my-university.com) as well as the location of a file to be retrieved (/assignments/cs101.htm).

Web servers respond to client requests by returning the requested resource or an error. A resource can be nearly any kind of information, from a Web page to an image to a video clip. The most common type of information returned, however, is a Web page. Web pages are documents composed in HyperText Markup Language (HTML). Web pages may contain references to other resources, such as images and other Web pages. These references are known as hyperlinks or links.

When you enter a Web address in a browser and hit Enter, it may seem like a single communication occurs: You enter a URL, and back comes a page. In fact, there is usually more to it than that. When a Web address is entered in a browser, it causes a request to be made from the specified Web server. The response back from the server is typically an HTML document, which the browser displays. The Web page may contain references to images, sound files, and other resources. For example, an HTML page may contain one or more image tags, such as *<IMG SRC="logo.jpg">*. In order for the image to be displayed, the browser must issue a separate request to the Web server to retrieve the actual picture. Most browsers retrieve images and other data automatically when they see the appropriate tags in the HTML. Thus, accessing a graphics-rich Web page may result in dozens of individual requests to a Web server.

Web Primitives

The most elementary operations for accessing Web sites are known as *Web primitives*. In the HTTP protocol, these are called GET, HEAD, POST, and PUT commands. Web-oriented languages usually support some or all of these operations. Table 10.1 summarizes the NQL Web primitive statements.

All of the NQL Web primitive statements are also available in function form. Table 10.2 lists the Web primitive functions. Most of them return a Boolean result indicating success or failure.

The Web statements and functions are described individually in the following sections.

Table 10.1 Web Primitive Statements

STATEMENT	PARAMETERS	DESCRIPTION
get	*url*	Retrieves a Web page or other resource from a Web server
getfile	*url*, *filespec*	Retrieves a Web page or other resource from a Web server and saves it as a file
getpage	*url*	Retrieves a Web page or other resource from a Web server (same as *get*)
head	*url*	Retrieves the header for a resource from a Web server
post	*url*, *data*	Submits a form to a Web server
putfile	*url*, *filespec*	Sends a file to a Web server
submit	[*base-url*]	Sets input fields and submits a form

Table 10.2 Web Primitive Functions

FUNCTION	DESCRIPTION
boolean get(*url*)	Retrieves a Web page or other resource from a Web server
boolean getfile(*url*, *filespec*)	Retrieves a Web page or other resource from a Web server and saves it as a file
string getpage(*url*)	Retrieves a Web page or other resource from a Web server
boolean head(*url*)	Retrieves the header for a resource from a Web server
boolean post(*url*, *data*)	Submits a form to a Web server
boolean putfile(*url*, *filespec*)	Sends a file to a Web server
boolean submit(*url*)	Sets input fields and submits a form

Retrieving a Web Page (HTTP GET)

The HTTP GET operation retrieves a Web page. In NQL, the *get* statement (or function) retrieves a Web page using this method. The parameter is a Web address (URL).

```
get url
boolean get(url)
```

The resource indicated by the URL is retrieved and pushed onto the stack. A failure condition results if the page could not be retrieved (the function returns false). Note that while a retrieved resource could be an HTML document (Web page), it could just as easily be something else, such as an image, an XML document, or a download file. For this reason, the information is pushed onto the stack in binary form. The following code retrieves a Web page, then extracts information from it using pattern matching.

```
get "http://www.my-news-site.com"
match '<li><a href="{link}">{headline}</a>'
while
{
    output link, headline
    nextmatch
}
```

An alias for *get* is *getpage*. The *getpage* statement is identical to the *get* statement, but the *getpage* function is different from the *get* function. Whereas *get*() returns a Boolean result to indicate success or failure, *getpage*() returns the Web page itself (and does not push it onto the stack).

```
getpage url
string getpage(url)
```

To retrieve a Web resource and store it as a disk file rather than on the stack, the *getfile* statement (or function) is used. The parameters are a URL and a file specification. The Web page (or other resource) is retrieved and stored on disk under the specified filename. The following code retrieves an image from a Web site and stores it as a disk file.

```
getfile "http://www.my-web-site.com/images/pic1.gif", "pic1.gif"
```

Retrieving a Web Page Header (HTTP HEAD)

The HTTP HEAD operation retrieves a Web page header. In NQL, the *head* statement (or function) retrieves a Web page's header information using this method. The parameter is a Web address (URL).

```
head url
boolean head(url)
```

The headers for the resource (not the resource itself) indicated by the URL are retrieved and pushed onto the stack. The following code retrieves the headers for a Web address.

```
head "http://www.my-site.com"
pop sHeaders
```

Posting a Form (HTTP POST)

The HTTP POST operation submits a Web form and retrieves a resulting Web page. In NQL, the *post* statement (or function) submits a form using this method. The parameter is a Web address (URL). For the *post* statement, the form data for the post must be pushed on the stack in advance. For the *post* function, the form data is a second parameter.

```
post url
boolean post(url, data)
```

The resource indicated by the URL is retrieved and pushed onto the stack. A failure condition results if the page could not be retrieved. Note that while a retrieved resource could be an HTML document (Web page), it could just as easily be something else, such as an image, an XML document, or a download file. For this reason, the information is pushed onto the stack in binary form. The following code posts to a Web server, then extracts information from the results using pattern matching.

```
push "query=watergate&page=1"
post "http://www.my-news-site.com/search",
match '<a href="{link}">{title}</a>'
while
{
    output link, title
    nextmatch
}
```

Submitting a Form

Although the *post* statement can be used to submit a form, it requires the developer to know the exact form of the URL for submitting the form. Sometimes it is simpler to mimic the operation a user would follow: Accessing a form page, filling in form fields, and then clicking a *Submit* button. The NQL *submit* statement (or function) works in this manner. *Submit* takes one optional parameter, which is the full URL of the form page. This URL is needed to convert a relative URL in the form tags into a complete one.

```
submit [base-url]
boolean submit([base-url])
```

Form submittal follows these steps: After retrieving a Web page that contains a form, variables are set that correspond to form fields. Then, a *submit* statement submits the form. The response page is retrieved and pushed onto the stack. A failure condition results if the page could not be retrieved The following code visits the main page of a commercial Web site, fills in a form field, and submits a search.

```
get "http://www.my-estore.com"
query = "jackets"
```

```
submit "http://www.my-estore.com"
pop ResponsePage
```

Compare the preceding code to the following code, which accomplishes the same thing but uses *post* rather than *submit*.

```
post "http://www.my-estore.com/scripts/search.dll"
pop ResponsePage
```

There are some trade-offs between *submit* and *post*. The work done by *submit* can be accomplished by *post*, but *submit* avoids having to compute form submittal URLs yourself; you can simply visit the main page and submit forms just as an interactive user would. When you use *post*, you have to do some research to figure out the target URL. On the other hand, retrieving a form page then submitting it adds an extra Web access that would be avoided with a direct *post*: Visiting a page with *get* followed by a *submit* is two Web accesses, whereas a *post* is just one Web access.

Sending a File

A file can be transmitted to a Web file through an HTTP PUT operation, if the server grants you permission to do so. The *putfile* statement or function puts a file to a Web server. The parameters are the Web server URL and a local file specification.

```
putfile url, filespec
boolean putfile(url, filespec)
```

If the file is transmitted without error, a success condition results (the function returns true). The following code uses *putfile* to send a file.

```
putfile "www.my-site.com/uploads", "new-orders.xml"
```

Controlled Browser

The Web primitives described above aren't satisfactory for navigation and data capture from dynamic, sophisticated Web sites. For example, some Web sites depend on client-side JavaScript code to execute in a real browser in order for the site to work. Web primitives are ineffective against these kinds of sites.

NQL includes an alternative form of Web access called a *controlled browser*. When using a controlled browser, an instance of Internet Explorer is launched and put under remote control by your script. Because a real browser is accessing Web sites, it can successfully access most Web sites with ease.

Table 10.3 shows the NQL controlled browser statements.

Table 10.3 Controlled Browser Statements

STATEMENT	PARAMETERS	DESCRIPTION
browse	[#*ID*,] *url*, *postdata*, *targetframe*, *headers*, *flags*	Navigates to a Web page
browseraction	[#*ID*,] *action*, *ItemType*, *ItemLocator*, *ItemData*, *WaitText*	Performs a browser action, such as clicking a link or button
closebrowser	[#*ID*]	Closes the controlled browser session
getbrowserdata	[#*ID*,] [*var*]	Retrieves the current document HTML
getbrowserframe	[#*ID*,] *FrameNoVar*, *FrameNameVar*	Returns the number and name of the current frame
getbrowserurl	[#*ID*] [, *var*]	Returns the current page's URL
gettotalforms	[#*ID*] [, *var*]	Returns a count of form sections on the current page
gettotalframes	[#*ID*] [, *var*]	Returns a count of frames on the current page
openbrowser	[#*ID*,] *url*, *postdata*, *targetframe*, *headers*, *flags*	Opens a controlled browser session and navigates to a starting page
setformfield	[#*ID*,] *text*, *value*	Sets the input field of a form
setbrowserform	[#*ID*,] *number*	Selects a form section of the current page
setbrowserframe	[#*ID*,] *number*	Selects a frame of the current page
submitform	[#*ID*,] [*number*]	Submits a form

All of the NQL controlled browser statements are also available in function form. Table 10.4 lists the controlled browser functions. Most of them return a Boolean result that indicates success or failure.

The controlled browser statements are described individually in the following sections.

The Browser Recorder

The NQL development environment (in the Windows edition) contains a unique browser session record-playback facility called the browser recorder. This feature is similar to the macro recorder in Microsoft Word, recording actions that can be repeated later.

Table 10.4 Controlled Browser Functions

FUNCTION	DESCRIPTION
boolean browse([#*ID*,] *url*, *postdata*, *targetframe*, *headers*, *flags*)	Navigates to a Web page
boolean browseraction([#*ID*,] *action*, *ItemType*, *ItemLocator*, *ItemData*, *WaitText*)	Performs a browser action, such as clicking a link or button
boolean closebrowser([#*ID*])	Closes the controlled browser session
boolean getbrowserdata([#*ID*,] [*var*])	Retrieves the current document HTML
boolean getbrowserframe([#*ID*,] *FrameNoVar*, *FrameNameVar*)	Returns the number and name of the current frame
boolean getbrowserurl([#*ID*] [, *var*])	Returns the current page's URL
boolean gettotalforms([#*ID*] [, *var*])	Returns a count of form sections on the current page
boolean gettotalframes([#*ID*] [, *var*])	Returns a count of frames on the current page
boolean openbrowser([#*ID*] *url*, *postdata*, *targetframe*, *headers*, *flags*)	Opens a controlled browser session and navigates to a starting page
boolean setformfield([#*ID*,] *text*, *value*)	Sets the input field of a form
boolean setbrowserform([#*ID*,] *number*)	Selects a form section of the current page
boolean setbrowserframe([#*ID*,] *number*)	Selects a frame of the current page
boolean submitform([#*ID*,] [*number*])	Submits a form

Clicking the browser recorder toolbar button launches an instance of Internet Explorer. As you visit a Web site and interact with it—entering text fields and clicking on links and buttons—each of your actions generates a controlled browser statement in the code editor.

Controlled Browser Sessions

The NQL controlled browser operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

1. A browser session is opened with *openbrowser*.
2. One or more browser actions may be commanded in any order desired, using *browse*, *browseraction*, *getbrowserdata*, *getbrowserframe*, *getbrowserurl*, *gettotalforms*, *gettotalframes*, *setformfield*, *setbrowserform*, *setbrowserframe*, and *submitform*.
3. The transaction session is closed with *closebrowser*.

Multiple Controlled Browser Sessions

NQL uses a very simple syntax for controlled browser operations, such as the following line:

```
openbrowser sURL
```

When there is a need to open multiple controlled browsers simultaneously, identifiers are needed to distinguish one browser session from another. In NQL, these identifiers are known as browser session IDs. A browser session ID is a name preceded by a pound sign, such as *#news* or *#SearchEngine1*. When a session ID is specified, it is the first parameter of a statement or function. For example,

```
openbrowser #doc1, URL
```

Opening a Controlled Browser Session

The *openbrowser* statement (or function) opens a Web session and launches an instance of Internet Explorer. There are six parameters, any of which may be blank or omitted: A starting URL, form data, a target frame, custom HTTP headers, flags, and a visibility flag. For simple access to a Web page only the URL needs to be specified.

```
openbrowser [url[,postdata[,targetframe[,headers[,flags[,visible]]]]]]
```

The *URL* parameter is the Web address to navigate to initially. If *URL* is not specified, the browser session does not navigate to a page; however, you can perform navigation in a separate *browse* statement.

The *postdata* parameter is the data required when submitting a form. If the URL parameter references a page that is normally a form submission, form data may be specified here.

The *targetframe* parameter is the name or number of a frame. This parameter is only needed when navigating to a page that contains multiple frames. It sets focus to the specified frame after navigating to the page.

The *headers* parameter allows you to specify additional HTTP headers to issue with when navigating to the starting page. Some Web sites require custom headers in order to respond correctly. Most of the time this parameter can be omitted.

The *flags* parameter customizes the browser session's behavior. For example, you might want to turn off display of images in order to retrieve Web pages faster. The *flags* value is a string created by combining any number of the following *flag* key words. The *flags* parameter defaults to empty when omitted.

- navOpenInNewWindow
- navNoHistory
- navNoReadFromCache

- navNoWriteToCache
- navAllowAutoSearch
- navBrowserBar
- NoImages
- NoVideos
- NoBgSounds
- NoScripts
- NoJavaApp
- NoRunActiveX
- NoDownloadActiveX
- AutomateIE

The *visible* parameter determines whether or not the launched instance of Internet Explorer is visible. Making the browser visible is useful during script creation and debugging, but visibility is typically turned off in production code. The *visible* parameter defaults to false if omitted.

The following code opens a controlled browser session and navigates to a Web page.

```
openbrowser "http://www.nqli.com"
```

When working with multiple browser sessions, specify a session ID as the first parameter.

```
openbrowser #yahoo, "http://www.yahoo.com"
openbrowser #lycos, "http://www.lycos.com"
```

Closing a Browser Session

The *closebrowser* statement ends a controlled browser session.

```
closebrowser
```

If a browser session was opened with a browser session ID, specify that ID when closing it.

```
closebrowser #ID
```

If you forget to close a browser session, NQL will close all open sessions when a script ends.

Navigating

The *browse* statement navigates to a Web address. The parameter is the URL you wish to navigate to.

```
browse url, postdata, targetframe, headers, flags
```

The parameters are identical to the *openbrowser* statement. Only the URL parameter is required.

The *URL* parameter is the Web address to navigate to initially.

The *postdata* parameter is the data required when submitting a form. If the URL parameter references a page that is normally a form submission, form data may be specified here.

The *targetframe* parameter is the name or number of a frame. This parameter is only needed when navigating to a page that contains multiple frames. It sets focus to the specified frame after navigating to the page.

The *headers* parameter allows you to specify additional HTTP headers to issue with when navigating to the starting page. Some Web sites require custom headers in order to respond correctly. Most of the time this parameter can be omitted.

The *flags* parameter customizes the browser session's behavior. For example, you might want to turn off display of images in order to retrieve Web pages faster. The *flags* value is a string created by combining any number of *flag* key words (listed in the description of *openbrowser*). The *flags* parameter defaults to empty if omitted.

The following code opens a browser session, then uses the browser to visit several Web sites.

```
openbrowser
browse "http://www.my-news-site-1.com"
getbrowserdata news1
browse "http://www.my-news-site-2.com"
getbrowserdata news2
browse "http://www.my-news-site-3.com"
getbrowserdata news3
closebrowser
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *browse* statement.

```
browse #ID, URL
```

Performing Browser Actions

The *browseraction* statement allows you to apply actions to the current Web page the browser is viewing, such as clicking on a link or button. The *browseraction* statement takes up to five parameters.

```
browseraction action, ItemType, ItemLocator, ItemData, WaitText
```

The *action* parameter indicates the type of action desired. The meaning and necessity of the other parameters depends on which action is being performed. Table 10.5 lists the browser actions and the meaning of parameters.

If the browser action is successful, a success condition results.

Table 10.5 Browser Actions

ACTION	ITEMTYPE	ITEMLOCATOR	ITEMDATA	WAITTEXT	DESCRIPTION
click	-	Item descriptor	-	Text on the next page to wait for	Clicks an anchor link
click	map	Text in the map area	-	Text on the next page to wait for	Clicks an area of an image map
click	link	Text in the anchor	-	Text on the next page to wait for	Clicks an anchor link
get	height	-	Variable name	-	Returns browser window height
get	left	-	Variable name	-	Returns browser window left position
get	statustext	-	Variable name	-	Returns browser status bar text
get	top	-	Variable name	-	Returns browser window top position
get	visible	-	Variable name	-	Returns browser visibility (true/false)
get	width	-	Variable name	-	Returns browser window width
go	back	-	-	-	Navigates to the previous page
go	forward	-	-	-	Navigates to the next page

Table 10.5 Browser Actions (continued).

ACTION	ITEMTYPE	ITEMLOCATOR	ITEMDATA	WAITTEXT	DESCRIPTION
go	home	-	-	-	Navigates to the home page
refresh	-	-	-	-	Refreshes the browser window
select	multi-option	Text in the item	-	-	Selects multiple options from a select list
select	option	Text in the item	-	-	Selects an option from a select list
set	cookie	-	Text data	-	Sets a cookie for the current page
set	debug	-	0 or 1	-	Turns debug messages on or off
set	formfield	Text in the field	Text data	-	Sets the value of a form field
set	height	-	Integer	-	Sets the browser window height
set	left	-	Integer	-	Sets the browser window left position
set	pausebeforeclicks	-	Number of seconds	-	Pauses n seconds before each click action
set	pausebeforeall	-	Number of seconds	-	Pauses n seconds before each action

continues

Table 10.5 Browser Actions (continued).

ACTION	ITEMTYPE	ITEMLOCATOR	ITEMDATA	WAITTEXT	DESCRIPTION
set	timeout	-	Seconds	-	Changes the page wait timeout
set	top	-	Integer	-	Sets the browser window top position
set	Use NewBrowser Windows	-	0 or 1	-	Sets whether or not new browser windows will be followed
set	visible	-	0 or 1	—	Sets the visibility of the browser window
set	width	-	Integer	-	Sets the browser window width
stop	-	-	-	-	Stops the current navigation
submit	-	Text in the form	-	-	Submits a form
wait	-	-	Seconds	-	Pauses for n seconds
wait	window	-	-	Window caption	Waits for a window with the specified caption to be displayed
waitinthread	windows	Keys to send to the window	Class of the window to locate	Caption of the window to locate	Waits in a thread for a specified window and optionally sends keystrokes to it when found

If a browser session was opened with a browser session ID, specify that ID when issuing a *browseraction* statement.

```
browseraction #ID, action, ItemType, ItemLocator, ItemData, WaitText
```

Retrieving Web Page Data

Once a browser session visits a Web page (via *openbrowser* or *browse*), the Web page may be retrieved using the *getbrowserdata* statement. The parameter is a variable to receive the data.

```
getbrowserdata var
```

The Web page data, typically an HTML document, is copied into the variable. The following code navigates to a Web page, then retrieves the Web page with *getbrowserdata*.

```
openbrowser url
getbrowserdata page
closebrowser
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *getbrowserdata* statement.

```
getbrowserdata #ID, var
```

Retrieving the Current Frame

The *getbrowserframe* statement returns the number and name of the current frame. The parameters are a variable to receive the frame number and a variable to receive the frame name.

```
getbrowserframe FrameNumberVar, FrameNameVar
```

The frame number variable receives the logical number of the frame, which will be a value greater than or equal to 1. The frame name is the name of the frame in the HTML page, if any.

If a browser session was opened with a browser session ID, specify that ID when issuing a *getbrowserframe* statement.

```
getbrowserframe #ID, FrameNumberVar, FrameNameVar
```

Retrieving the Current URL

The *getbrowserurl* statement returns the URL of the current frame. The parameter is a variable to receive the URL.

```
getbrowserurl var
```

The Web address of the current page is returned in the specified variable. If the browser is viewing a frame-based page, the Web address of the frame with current focus is returned. The following code visits a Web page, submits a form, then retrieves the URL of the results page.

```
openbrowser url1
submitform
getbrowserurl url2
closebrowser
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *getbrowserurl* statement.

```
getbrowserurl #ID, var
```

Submitting a Form

To submit a form, the *submitform* statement is used. The parameter is an optional form number that defaults to 1. There is no need to specify a form number if a page contains only one form section.

```
submitform [number]
```

The following code submits a form using the second form section of a page.

```
submitform 2
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *submitform* statement.

```
submitform #ID, [number]
```

Counting the Forms on a Page

The *gettotalforms* statement returns the number of forms on the current Web page. The parameter is a variable to receive the count.

```
gettotalforms var
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *gettotalforms* statement.

```
gettotalforms #ID, var
```

Counting the Frames on a Page

The *gettotalframes* statement returns the number of frames on the current Web page. The parameter is a variable to receive the count.

```
gettotalframes var
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *gettotalframes* statement.

```
gettotalframes #ID, var
```

Setting Form Fields

When preparing to submit a form, it is necessary to fill in form fields. The *setformfield* statement sets a form field value. The parameters are the name of the form field and a value.

```
setformfield fieldname, fieldvalue
```

The following code visits a Web page and sets two form fields in anticipation of submitting a form.

```
openbrowser
browse sURL
setbrowserform 1
setformfield 1, sQuery
setformfield "max", 10
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *setformfield* statement.

```
setformfield #ID, fieldname, fieldvalue
```

Selecting a Form

On Web pages with multiple forms, it is necessary to select a specific form before setting form fields. The *setbrowserform* statement selects a form. The parameter is a form number greater than or equal to 1.

```
setbrowserform form-number
```

The following code selects the second form on a page.

```
setbrowserform 2
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *setbrowserform* statement.

```
setbrowserform #ID, form-number
```

Selecting a Frame

When working with frame-based Web pages, it is necessary to select a specific frame before performing actions on the page. The *setbrowserframe* statement selects a frame. The parameter is a frame number greater than or equal to 1.

```
setbrowserframe frame-number
```

The following code selects the third frame on a page.

```
setbrowserframe 3
```

If a browser session was opened with a browser session ID, specify that ID when issuing a *setbrowserframe* statement.

```
setbrowserframe #ID, frame-number
```

Site Crawling and File Downloading

NQL provides statements for crawling Web sites and downloading files. Table 10.6 summarizes the NQL site crawling statements.

Many of the NQL site crawling statements are also available in function form. Table 10.7 lists the site crawling functions. Most of them return a Boolean result that indicates success or failure.

The site crawling statements and functions are described individually in the following sections.

Converting a URL to an Absolute URL

The *absurl* statement or function turns a relative (partial) Web address into an absolute one. The statement form assumes a relative URL is on the stack. The parameter is the base URL to be used to make the URL on the stack complete.

```
absurl base-url
```

The function form of *absurl* takes two parameters: A relative URL and a base URL. The string result is an absolute URL.

```
string absurl(url, base-url)
```

Table 10.6 Site Crawling Statements

STATEMENT	PARAMETERS	DESCRIPTION
absurl	*url*	Retrieves a Web page or other resource from a Web server
crawl	*url*, *depth* [, *text*[, *type*]]	Crawls a site and places URLs and HTML on the stack
crawl	[*base-url*]	Computes links for the HTML page on the stack
crawl	*url*, *depth*	Crawls a site and executes a code block for each page encountered*
download	*url*, *depth*, *file-type* [, *path* [, *overwrite* [, *text* [, *notify* [, *link-type*]]]]]	Crawls a site and downloads files
fetch	*base-url*, *pattern* [, *text*]	Crawls the HTML on the stack to one level and pushes the pages matching a pattern onto the stack

Table 10.7 Site Crawling Functions

FUNCTION	DESCRIPTION
string absurl(*url*, *base-url*)	Retrieves a Web page or other resource from a Web server
boolean crawl(*url*, *depth* [, *text*[, *type*]])	Crawls a site and places URLs and HTML on the stack
boolean crawl([*base-url*])	Computes links for the HTML page on the stack
boolean download(*url*, *depth*, *file-type* [, *path* [, *overwrite* [, *text* [, *notify* [, *link-type*]]]]])	Crawls a site and downloads files
boolean fetch(*base-url*, *pattern* [, *text*])	Crawls the HTML on the stack to one level and finds pages matching a pattern

If the URL is already absolute, it is not modified. The following code uses *absurl* to ensure that Web addresses are complete before outputting them.

```
BaseURL = "http://www.my-news-site.com"
get BaseURL
match '<li><a href="{link}">{headline}</a>'
while
{
    output absurl(link, BaesURL), headline
    nextmatch
}
```

The *absurl* statement or function is useful for making retained Web addresses meaningful, where loss of the original context of the URL would cause problems in understanding relative links.

Crawling Web Sites

NQL provides three forms of *crawl* statements for crawling Web sites, each with different parameters and different behaviors.

Crawling to the Stack

The first form of *crawl* takes a given URL, retrieves the page, and crawls its links to other pages, up to the specified number of levels. The retrieved pages are pushed onto the stack. The parameters are a URL, a crawl depth, continuation page text, and a link type ("domain", "server", or "all"). The last two parameters are optional.

```
crawl url, depth [, continuation-text [, link-type]]
boolean crawl(url, depth [,continuation-text[, link-type]])
```

The Web address is retrieved, the URL and HTML are pushed onto the stack, and links within it are recognized. Those links in turn are retrieved, repeating until the desired site depth has been satisfied. The number of pages is set into a variable named *CrawlCount*. A log of the site crawl is set into a variable named *CrawlLog*. There is one stack entry for each page: The first line is the URL, and the remainder is the page's HTML content. The *nextline* statement is useful for separating the URL from the HTML in the stack items.

If continuation page text is specified, only links containing that text are visited. This is handy when you know something about the format of desired links as compared to other links. For example, when processing search results from a retail site, following all links containing "results" might avoid unnecessary crawling of pages not related to your interests.

If a link type is specified, the extent of the site crawl is affected.

- A link type of "domain" restricts the crawl to links that are part of the same domain as the original URL (that is, the domain.type part of a system.domain.type URL is the same as the original). For example, if you were crawling www.my-site.com, search.my-site.com and store.my-site.com are considered part of the same domain and would be included in the crawl.
- A link type of "server" restricts the crawl to links that are on the same server as the original URL. For example, if you were crawling www.my-site.com, links to www.my-site.com would be followed while links to support.my-site.com would not.
- A link type of "all" follows all links, regardless of domain.

The following code uses *crawl* to crawl a Web site to three levels, then retrieves and displays the URL and HTML of each page.

```
string URL, HTML

crawl "http://www.my-site.com", 3

for pages = CrawlCount
{
    nextline URL
    pop HTML
    show URL, HTML
}
```

Crawling a Single Page

The second form of *crawl* examines the HTML page on the stack and identifies its links, but does not visit them. The optional parameter is a base URL for turning relative URLs into absolute ones.

```
crawl [base-url]
boolean crawl([base-url])
```

The Web page is analyzed. Just one stack item is created containing the URLs of all links in the page, each separated by a newline. The *nextline* function is handy for iterating through the links. The following code uses *crawl* to find the links in a page.

```
get "http://www.my-site.com"
crawl "http://www.my-site.com"
while nextline(sCurrLink)
{
    ...do something with sCurrLink...
}
```

Crawling Using Code Blocks

PLANNED FEATURE The third form of *crawl* crawls a Web site to the specified number of levels, executing a code block for each page that is found. The parameters are a URL and a crawl depth.

```
crawl url, depth
{
    ...process page...
}
```

As pages are found, the *crawl* block executes. The top stack value is the current page, and stack clean-up is automatic. The variable *CrawlURL* contains the URL of the current page, *CrawlParentURL* contains the parent page's URL, and *CrawlLevel* contains the crawl level at which the page was found.

Web Services

The growing need for automation between organizations, their networks, and the Internet has led to the idea of *Web services*. Web services are the automated counterpart to the World Wide Web as we know it today. Web services will permit two organizations' computer systems to exchange information or perform transactions automatically, in contrast to the norm today where a human being performs these tasks interactively with a Web browser.

To make Web services attractive and easy to implement, they are based on two winning technologies that have already been embraced in the computer industry: HTTP (the protocol of the Web) and XML. Web services work by submitting XML requests to a Web address and receiving XML responses. You can think of a Web service as a function or component you call, with the interesting quality that the software doing the responding may be halfway around the world. If you're familiar with the concept of a remote procedure call, you'll be right at home with Web services.

The primary protocol for Web services is Simple Object Access Protocol (SOAP). There are also related specifications for describing Web services and for discovering the available Web services on a Web server. Network Query Language has the ability to act as a SOAP client, making requests of Web services and retrieving the results for use in your programs.

Accessing a Web Service

The *getobject* statement or function accesses a Web service, acting as a SOAP client. The parameters are a Web service URL, the name of the method (Web service) to access, and an optional interface. If you omit the interface, the default interface for the Web service is used. Before invoking *getobject*, an XML payload must be pushed onto the stack.

```
getobject url, method [, interface]
boolean getobject(url, method [, interface])
```

The XML payload is popped off of the stack, and the Web service is accessed. If the Web service responds, a success condition results (the function returns true). If the Web service returned a value, it is stored in a variable named *MethodName*Return. The following code shows a Web service named *reverse_string* being accessed to reverse a character string.

```
push '<SerializedStream SerializationPattern="urn:schemas-
microsoft-com:soap:v1" headers="#ref-0" main="#ref-1">
<headers id="ref0"><InterfaceName>soap:cdl:com.develop.demos.string_test
</InterfaceName></headers><reverse_string id="ref-1"><__param0>!dlrow
olleH</__param0></reverse_string></SerializedStream>'

if getobject("http://soap.develop.com/sd2/vbasperl.soap ",
    "reverse_string", "soap:cdl:com.develop.demos.string_test")
{
        show [reverse_stringReturn]
    }
    end
```

Tutorial: BookSearch

Now that the basics of NQL Web interaction have been covered, let's write a Web-oriented script. In this tutorial, you will create a script that submits a search to a book retail Web site and captures the results. The script will need to visit the site's page, select books as a category, enter a subject, and submit the search. Book titles and prices will need to be located and captured from the response page. The captured books will output as XML.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the BookSearch script.
3. Run the BookSearch script.
4. Understand the BookSearch script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Set input fields and submit a form.
- Capture data from a Web page.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the BookSearch Script

In the NQL development environment, enter the script shown in Listing 10.1 and save it under the name BookSearch.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy BookSearch.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial

```
//booksearch - submits a search to Amazon.com, collects results from
first response page, and outputs as XML.

get "http://www.amazon.com"
index = "Books"
[field-keywords] = "artificial intelligence"
submit "http://www.amazon.com"
match '<b><a href="{link}">{title}</a></b>*<br><font
face=verdana,arial,helvetica size=-1>by {Author}</font>*Our Price:
<font *>${price}</font></b><br>'
while
{
    outputrecord "result"
    output title, author, price
    nextmatch
}
```

Listing 10.1 BookSearch script.

materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select BookSearch.nql from the Tutorials\ch10 folder.

At this point, the script in Listing 10.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Step 3: Run the BookSearch Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of book results. If this does not happen, check the following:

- Check your script for typographical errors.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the BookSearch script run, submitting a form to a Web site and capturing data from the response. Although we haven't discussed how it works yet, you've witnessed NQL interacting with a Web site. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the BookSearch Script

We now want to make sure we understand every part of the BookSearch script. The first line is a comment line.

```
//booksearch - submits a search to Amazon.com, collects results from
first response page, and outputs as XML.
```

The script retrieves the Amazon.com home page with a *get* statement. If successful, the HTML page is pushed onto the stack.

```
get "http://www.amazon.com"
```

The Amazon.com home page contains a search form with two input fields. One is named *index*, and is a selection of product types (books, music, electronics, and so on). The other is named *field-keywords*, and is a place to enter search keywords. Before submitting the search form, the input fields will need to be set.

To set the input fields, variables are assigned with the same names as the form fields. In the case of *field-keywords*, we need to put square brackets around the variable name because it is not a legal NQL variable name. The brackets tell the compiler to accept the variable name anyway.

```
index = "Books"
[field-keywords] = "artificial intelligence"
```

With the input fields set, we can now submit the form with a *submit* statement. The HTML page on the stack is examined, the form section is found, and the input fields are set from the variables we just assigned. The page on the stack is replaced by the response to the form.

```
submit "http://www.amazon.com"
```

Now that the response page is on the stack, we can extract data from it. A *match* statement searches for the first occurrence of a book title, author, and price in the format we expect. (If the Web site changes significantly, this pattern won't work and will need to be revised).

```
match '<b><a href="{link}">{title}</a></b>*<br><font
face=verdana,arial,helvetica size=-1>by {Author}</font>*Our Price: <font
*>${price}</font></b><br>'
```

If the match is successful, a *while* loop executes where the title, author, and price are output in XML. The values are output by an *output* statement, but this is preceded by an *outputrecord* statement to frame the fields in <result> tags. A *nextmatch* statement finds the next match, so that the *while* loop will continue as long as there are more matches on the page.

```
while
{
    outputrecord "result"
    output title, author, price
```

```
        nextmatch
    }
```

At this point, you've seen an NQL script interact with a Web site and capture results.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Make the interaction with the Web site more aggressive, and/or find a different Web site to interact with. If necessary, use the controlled browser instead of Web primitives for sophisticated navigation.
- Add error checking, so that a *get* or *submit* failure is detected and reported.

Chapter Summary

Network Query Language includes several methods for accessing Web sites, including basic Web primitives.

- The *get*, *getpage*, and *getfile* statements retrieve Web pages (an HTTP GET operation).
- The *head* statement retrieves a Web page header (an HTTP HEAD operation).
- The *post* and *submit* statements submit forms (an HTTP POST operation).
- The *putfile* statement transmits a file to a Web server (an HTTP PUT operation).

For deluxe interaction with dynamic Web sites, NQL can launch a browser under remote control. Controlled browser capabilities include the following:

- The NQL development environment offers a browser recorder for training Web scripts.
- The *openbrowser* statement begins a browser session.
- The *browse* statement navigates to a Web address.
- The *closebrowser* statement ends a browser session.
- The *getbrowserdata* statement retrieves the current HTML document from the browser.
- The *getbrowserurl* statement retrieves the current URL from the browser.
- The *getbrowserframe* and *getbrowserform* statements retrieve the current frame and form selection from the browser.
- The *gettotalframes* and *gettotalforms* statements retrieve the current frame and form count from the browser.
- The *setbrowserframe* and *setbrowserform* statements select a frame and form.

- The *setformfield* statement sets a form field.
- The *submitform* statement submits a form.

NQL offers facilities for crawling Web sites and downloading files.

- The *absurl* statement converts a relative Web address into an absolute one.
- The *crawl* statement crawls a Web site, with three possible behaviors.
- The *download* statement crawls a Web site and downloads files.

Retrieving data from Web sites is one of the most frequent uses for NQL. With the communications and conversion facilities described in other chapters, you can create scripts that transport (and transform) information between your local network and the World Wide Web.

CHAPTER

11

Processing HTML

Extracting information from Web pages can be a tricky business. Web pages are visual entities designed to be read by people, not processed by programs. Although XML promises to simplify the job of extracting data from Web pages, HTML is still the code primarily found on Web sites today. Network Query Language provides pattern-matching capabilities for locating objects on Web pages. For example, pattern matching can be used to locate part numbers, descriptions, and prices from an online catalog. This chapter begins with an overview of the five types of pattern matching available in NQL, then goes into detail about each one. A tutorial is supplied for each type of pattern matching. The chapter also covers HTML operations.

Types of Pattern Matching

The millions of Web pages on the Internet vary greatly in their organization. Some Web pages are simple, while others are amazingly complex. Some sites serve up static Web pages while others are dynamic, creating pages on the fly in response to input fields and user actions.

Pattern matching comes to the rescue, making it possible to express rules for locating wanted data. With properly expressed rules, the HTML falls through a sieve, where desired information is captured and undesired information is discarded. Nearly all forms of pattern matching are based on either of two suppositions: The desired data has identifiable landmarks surrounding it or the data itself contains unique characteristics.

Due to the variety of data on the Web, no one method of pattern matching meets all needs. For that reason, NQL contains a variety of pattern-matching methods: HTML pattern matching, data pattern matching, HTML table processing, regular expressions, and Web queries. Table 11.1 lists some of the attributes of the different types of pattern matching.

HTML Pattern Matching

Pattern matching in HTML is based on patterns of tags and text sequences. Specifically, a pattern match contains tags, text, and wildcards. Wildcards are used for capturing information, and are notated by placing a variable name in curly braces. Consider the pattern below, which has two wildcards, one named *link* and one named *headline*.

```
<LI><A HREF="{link}">{headine}</A>
```

This pattern can be used to capture news headlines from some news Web sites. The following fragment of an HTML document contains three matches to this pattern.

```
<LI><A HREF="/politics/story104.htm">Future of F16 in doubt</A>
<P>
<LI><A HREF="/politics/story113.htm">Filibuster Stalls Education
Bill</A>
<P>
<LI><A HREF="/politics/story114.htm">Senator Considers Changing
Parties</A>
```

The HTML pattern-matching statements are listed in Table 11.2.

The HTML pattern matching statements are also available as functions. Table 11.3 lists the HTML pattern matching functions.

Pattern matching with HTML is one of the easier forms of pattern matching to use, because you can create patterns by copying and pasting HTML from the source to any Web page. However, this form of pattern matching is very sensitive to changes in the HTML. An innocent or minor revision to a Web page, such as a cosmetic change, can render an HTML pattern useless.

Finding a Match

The *match* statement finds a pattern match against the HTML on the stack. The parameter is an HTML pattern.

```
match *pattern*
```

If an occurrence of the pattern is located, a success condition results and variables are created containing the captured data. The HTML on the stack is shortened; all that remains is the portion after the pattern match. If the pattern cannot be located, a failure

Table 11.1 Comparison of Pattern-Matching Methods

ATTRIBUTE	HTML PATTERN	DATA PATTERN	HTML TABLE	REGULAR EXPRESSIONS	WEB QUERIES
Patterns	Tags and text	Sequences of objects	Sequences of statements	Character patterns	SQL-like queries
Sensitivity to Web page layout changes	High	Low	High	High	Low
Sensitivity to changes in the order of data	High	High	High	High	Low
What can be captured	Tags and text	Text	Tags and text within tables	Tags and text	Tags and text

Table 11.2 HTML Pattern-Matching Statements

STATEMENT	PARAMETERS	DESCRIPTION
match	*pattern*	Matches the specified pattern against the HTML on the stack
matchall	*pattern*	Matches the specified pattern against the HTML on the stack repeatedly
nextmatch		Repeats the last pattern match against the HTML on the stack

Table 11.3 HTML Pattern-Matching Functions

FUNCTION	DESCRIPTION
boolean match(*pattern*)	Matches the specified pattern against the HTML on the stack
boolean matchall(*pattern*)	Matches the specified pattern against the HTML on the stack repeatedly
boolean nextmatch( )	Repeats the last pattern match against the HTML on the stack

condition results and the HTML is removed from the stack. The following code retrieves a Web page, then uses HTML pattern matching to locate all table cells on the page.

```
get URL
while match("<TD>{data}</TD>")
{
    ...do something with data...
}
```

Patterns may contain optional sections. Optional sections are useful when dealing with irregularities in HTML such as the following HTML code, where some items have additional parts not common to all items.

```
<B>$15.00 </B><P>
<B>$17.00 (floor model)</B><P>
<B>$8.00 </B><P>
<B>$12.00 (sale price this weekend only)</B><P>
```

Optional sections are enclosed in square brackets. Thus, the pattern below is a valid one for dealing with this wayward HTML.

```
<B>{price} [({note})]</B>
```

An optional section may contain {variable} sections, but not at the very end of the pattern. Thus, <TR>[<TD>{value}]</TD></TR> is not a valid pattern while <TR>[<TD>{value}</TD>]</TR> is.

Finding the Next Match

The *nextmatch* statement or function repeats the last pattern match. No parameters are required.

```
nextmatch
```

Strictly speaking, you could just issue another *match* statement rather than using *nextmatch*, but *nextmatch* is handy in that you don't have to specify the pattern all over again—it's remembered from the initial *match*. The following code uses *nextmatch* to iterate through all occurrences of a pattern on a page.

```
get url
match pattern
while
{
    ...do something with data from the match...
    nextmatch
}
```

Finding All Matches

The *matchall* statement is like *match*, but performs multiple pattern matches against the HTML on the stack until there are no more. Like *match*, the parameter is a pattern.

```
matchall pattern
```

Unlike *match*, *matchall* does not return its data in variables. Instead, the matching values are added to the script's output stream in XML format. To see this in action, try entering the following script in the NQL development environment and running it, supplying a URL to *get* that contains tables.

```
push "<td>Brown, Molly</td><td>Black, Burton</td><td>White, Willie</td>"
matchall "<td>{last}, {first}</td>"
```

The XML output contains all of the table cells.

Tutorial: Quote1

Let's see HTML pattern matching work in the real world. The script will request a page of multiple stock quotes from a financial Web site, locate the quote information using HTML pattern matching, and output the captured data as XML. All five of the tutorials in this chapter access the same quote data from the same source, but apply a different type of pattern matching in each case.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Quote1 script.
3. Run the Quote1 script.
4. Understand the Quote1 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Match Web page data using HTML pattern matching.
- Output data as XML.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Quote1 Script

In the NQL development environment, enter the script shown in Listing 11.1 and save it under the name Quote1.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Quote1.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Quote1.nql from the Tutorials\ch11 folder.

At this point, the script in Listing 11.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Since the quote site for this script is imaginary (a real quote site would have changed layout by the time you read this book), a local copy of the expected HTML is

```
//Quote1 - get a stock quote and extract data,
//         using HTML pattern matching.

//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"

//perform a pattern match to locate the quote fields in the HTML.

match "<TABLE>
<TR><TH COLSPAN=2>Micro Semantic Extensions, Inc. (MSXX)</TD></TR>
<TR><TD>Last Sale</TD><TD><TD>{LastSale}</TD></TR>
<TR><TD>Open</TD><TD>{Open}</TD></TR>
<TR><TD>Change</TD><TD>{Change}</TD></TR>
<TR><TD>Change %</TD><TD>{Percent}%</TD></TR>
<TR><TD>Volume</TD><TD>{Volume}</TD></TR>
<TR><TD>High</TD><TD>{High}</TD></TR>
<TR><TD>Low</TD><TD>{Low}</TD></TR>
</TABLE>"

//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

Listing 11.1 Quote1 script.

included in the \Tutorial\ch11 directory. The script will load this page rather than perform an actual Internet access. The HTML we will be scanning is shown in Listing 11.2.

Each of the other examples in this chapter will be against this same HTML page, but using a different form of pattern matching each time.

Step 3: Run the Quote1 Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of stock quotes.

```
<LastSale>65.80</LastSale><Open>65.65</Open><Change>0.33</Change><Perce
nt>0.50</Percent><Volume>21,098,400</Volume><High>66.88</High><Low>65.5
4</Low>
```

If this does not happen, check the following:

```
<HTML>
<HEAD>
<TITLE>Stock Quote</TITLE>
</HEAD>
<BODY>
<TABLE>
<TR><TH COLSPAN=2>Micro Semantic Extensions, Inc. (MSXX)</TD></TR>
<TR><TD>Last Sale</TD><TD><TD>65.80</TD></TR>
<TR><TD>Open</TD><TD>65.65</TD></TR>
<TR><TD>Change</TD><TD>0.33</TD></TR>
<TR><TD>Change %</TD><TD>0.50%</TD></TR>
<TR><TD>Volume</TD><TD>21,098,400</TD></TR>
<TR><TD>High</TD><TD>66.88</TD></TR>
<TR><TD>Low</TD><TD>65.54</TD></TR>
</TABLE>
```

Listing 11.2 Stock Quote HTML.

- Check your script for typographical errors.
- Check that your script and quote.htm are in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Quote1 script run, retrieving a Web page and locating stock quote data. Although we haven't discussed how it works yet, you've witnessed NQL putting HTML pattern matching to use. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Quote1 Script

We now want to make sure we understand every part of the Quote1 script. The first two lines are comment lines.

```
//Quote1 - get a stock quote and extract data,
//         using HTML pattern matching.
```

The stock quote is retrieved. In a real script, a *get* statement would be used to access an Internet quote page. Here, we comment out the *get* statement and use *load* to load the stock quote HTML page provided with the tutorial. The net effect is the same: An HTML page is on the stack.

```
//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"
```

Now for the pattern match. In HTML pattern matching, the *match* statement is used along with a pattern of HTML tags and text. As expected, the pattern is very similar to the actual page listed earlier in Listing 11.2. Areas to be captured are represented as variable names in curly braces.

```
//perform a pattern match to locate the quote fields in the HTML.

match "<TABLE>
<TR><TH COLSPAN=2>Micro Semantic Extensions, Inc. (MSXX)</TD></TR>
<TR><TD>Last Sale</TD><TD><TD>{LastSale}</TD></TR>
<TR><TD>Open</TD><TD>{Open}</TD></TR>
<TR><TD>Change</TD><TD>{Change}</TD></TR>
<TR><TD>Change %</TD><TD>{Percent}%</TD></TR>
<TR><TD>Volume</TD><TD>{Volume}</TD></TR>
<TR><TD>High</TD><TD>{High}</TD></TR>
<TR><TD>Low</TD><TD>{Low}</TD></TR>
</TABLE>"
```

If the pattern match is successful, the quote fields have been stored in variables. They can now be output in XML format with the *output* statement.

```
//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

At this point, you've seen an NQL script interact with a Web site and capture results using HTML pattern matching.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Amend the script to work with a real quote site.
- Add error checking, so that a *get* failure is detected and reported.

Data Pattern Matching

Data pattern matching finds objects on Web pages without regard to HTML tags. Patterns are series of data types, such as names, dates, or amounts. Unlike HTML pattern matching, data patterns can be created without having to look at the HTML source to a Web page. Often, you can determine a data pattern just by viewing a page in your browser.

Patterns may contain wildcards that are used for capturing information and are notated by placing a variable name in curly braces. Consider the following pattern, which has two wildcards, one named *company* and one named *industry*.

```
name{company}+text:industry+name{industry}
```

Data patterns consist of one or more *terms*. Terms may contain data types, text, and wildcards. The data type is required; the text and wildcard portions are optional.

```
data-type [:text] [{variable}]
```

The first part of a term, which is required, is a data type such as *name* or *amount*. Table 11.4 lists the available data types.

Any term may be followed by a colon and some text that must appear within the term. The term below matches text that contains the word *name*.

```
text:name
```

If you want to capture a term as data, it is followed by a variable name in curly braces. The pattern below has three terms, two of which store results in variables.

```
name+name{city}+state{state}
```

Table 11.4 Data Pattern Matching Data Types

DATA TYPE	DESCRIPTION
amount	Matches a U.S. dollar amount (with a dollar sign)
date	Matches a date
datetime	Matches a date followed by a time
float	Matches a floating point number (with a decimal point)
fraction	Matches a fraction or an integer followed by a fraction
integer	Matches an integer (whole number)
name	Matches a sequence of capitalized words
number	Matches an integer, an integer with commas, a float, or an amount
phone	Matches a phone number
ssn	Matches a U.S. social security number
state	Matches a two-letter U.S. or Canadian postal abbreviation
text	Matches any text
time	Matches a time
word	Matches a single word
words	Matches one or more words terminated by two spaces, a tab, or a newline
zipcode	Matches a U.S. five-digit or nine-digit zip code

If there are multiple terms in a pattern, each term is separated by a plus sign or a comma. A plus sign means the data types must be immediately adjacent to each other on the page. A comma means other data may occur between the fields. The pattern below requires *date* and *time* to be adjacent, but *phone* can be elsewhere on the page.

```
date+time,phone
```

There is just one NQL statement for data pattern matching, *matchdata*.

Finding a Match

The *matchdata* statement (or function) finds a pattern match against the HTML on the stack. The parameter is a data pattern.

```
matchdata pattern
```

If an occurrence of the pattern is located, a success condition results and variables are created reflecting the data. The HTML on the stack is shortened; all that remains is the portion after the pattern match. If the pattern cannot be located, a failure condition results and the HTML is removed from the stack. The following code retrieves a Web page, then uses data pattern matching to locate all addresses on the page.

```
get URL
while matchdata("name{city}+state{state}+zipcode{zip}")
{
    ...do something with city, state, and zip...
}
```

Patterns may contain optional sections. Optional sections are useful when dealing with irregularities in HTML such as the following HTML code. Some items have additional parts not common to all items.

```
<B>$15.00</B><P>
<B>$17.00 (floor model)</B><P>
<B>$8.00</B><P>
<B>$12.00 (sale price this weekend only)</B><P>
```

Optional sections are enclosed in square brackets. Thus, the following pattern is valid for dealing with this HTML.

```
amount{price}+[text:(+words{note}+text:)]
```

Tutorial: Quote2

Let's see data pattern matching work in the real world. The script will request a page of multiple stock quotes from a financial Web site, locate the quote information using data pattern matching, and output the captured data as XML. All five of the tutorials in this chapter access the same quote data from the same source, but apply a different type of pattern matching in each case.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Quote2 script.
3. Run the Quote2 script.
4. Understand the Quote2 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Match Web page data using data pattern matching.
- Output data as XML.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Quote2 Script

In the NQL development environment, enter the script shown in Listing 11.3 and save it under the name Quote2.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Quote2.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Quote2.nql from the Tutorials\ch11 folder.

At this point, the script in Listing 11.3 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Since the quote site for this script is imaginary (a real quote site would have changed layout by the time you read this book), a local copy of the expected HTML is included in the \Tutorial\ch11 directory. The script will load this page rather than perform an actual Internet access. The HTML we will be scanning is shown previously in Listing 11.2.

```
//Quote2 - get a stock quote and extract data,
//         using data pattern matching.

//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"

//perform a data pattern match to locate the quote fields in the page

matchdata "text:Last Sale+number{LastSale},
           text:Open+number{Open},
           text:Change+number{Change},
           text:Change %+number{Percent},
           text:Volume+number{Volume},
           text:High+number{High},
           text:Low+number{Low}"

//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

Listing 11.3 Quote2 script.

Step 3: Run the Quote2 Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of stock quotes.

```
<LastSale>65.80</LastSale><Open>65.65</Open><Change>0.33</Change><Perce
nt>0.50</Percent><Volume>21,098,400</Volume><High>66.88</High><Low>65.5
4</Low>
```

If this does not happen, check the following:

- Check your script for typographical errors.
- Check that your script and quote.htm are in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Quote2 script run, retrieving a Web page and locating stock quote data. Although we haven't discussed how it works yet, you've witnessed NQL putting data pattern matching to use. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Quote2 Script

We now want to make sure we understand every part of the Quote2 script. The first two lines are comment lines.

```
//Quote2 - get a stock quote and extract data,
//         using data pattern matching.
```

The stock quote is retrieved. In a real script, a *get* statement would be used to access an Internet quote page. Here, we comment out the *get* statement and use *load* to load the stock quote HTML page provided with the tutorial. The net effect is the same: An HTML page is on the stack.

```
//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"
```

Now for the pattern match. In data pattern matching, the *matchdata* statement is used along with a pattern of logical data types and hints. Areas to be captured are represented as variable names in curly braces. Since each value we want to gather is to the right of its label, we use a text:*label*+number{*variable*} construct; that is, a text label containing certain text is found, and the next value after it that is a number is the field we want to capture.

```
//perform a data pattern match to locate the quote fields in the page
matchdata "text:Last Sale+number{LastSale},
           text:Open+number{Open},
           text:Change+number{Change},
           text:Change %+number{Percent},
           text:Volume+number{Volume},
           text:High+number{High},
           text:Low+number{Low}"
```

If the pattern match is successful, the quote fields have been stored in variables. They can now be output in XML format with the *output* statement.

```
//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

At this point, you've seen an NQL script interact with a Web site and capture results using data pattern matching.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Amend the script to work with a real quote site.
- Add error checking, so that a *get* failure is detected and reported.

HTML Table Processing

Table processing in HTML parses an HTML page, then allows you to access the table data using a series of NQL statements. Rather than specifying a pattern, discrete statements are used in combination to navigate through the cells (rows and columns) of tables.

The HTML table processing statements are listed in Table 11.5.

Table 11.5 HTML Table Processing Statements

STATEMENT	PARAMETERS	DESCRIPTION
changetablepos	*table*, *row*, *value*	Changes relative table, row, and column position
closehtml		Closes an HTML table processing session
findcell	*text*	Finds the first cell that contains the specified text
findnextcell	*text*	Finds the next cell that contains the specified text
firstcell		Finds the first table cell on the page
firstcol		Moves to the first column of the current row in the current table
firstrow		Moves to the first cell of the first row of the current table
firsttable		Moves to the first cell of the first row of the first table on the page
getcell	*var*	Retrieves the cell contents of the current table cell location
getcelltext	*var*	Retrieves the cell contents of the current table cell location, text-only
getcol	*var*	Retrieves multiple cells from a table column
getrow	*var*	Retrieves multiple cells from a table row
gettablecount	*var*	Returns the number of tables on the Web page

continues

Table 11.5 HTML Table Processing Statements (continued)

STATEMENT	PARAMETERS	DESCRIPTION
gettabledata		Retrieves an entire table as delimited data pushed onto the stack
nextcell		Advances to the next cell of the current table
nextcol		Advances to the next column of the current row of the current table
nextrow		Advances to the next row of the current table
nexttable		Advances to the next table on the page
openhtml	[*url*]	Begins an HTML table processing session
setcol	*n*	Sets the current table column location
setrow	*n*	Sets the current table row location
settable	*n*	Sets the current table number
settablepos	*table*, *row*, *value*	Sets absolute table, row, and column positions

The HTML table processing statements are also available as functions. Table 11.6 lists the HTML table processing functions.

Table processing is most useful with HTML if table records are in a uniform and consistent format, or if you have a definite idea of the location of the data you want.

Opening a Table Processing Session

The *openhtml* statement (or function) retrieves a Web page and opens a table processing session. To retrieve the page from a Web site, specify a URL as a parameter. If no URL parameter is specified, the HTML is presumed to be the top value on the stack.

```
openhtml [url]
```

A success condition results if the page is acquired and parsed without error. The following code retrieves a Web page and opens it for table processing.

```
openhtml "http://www.my-site.com"
```

Once an HTML page is open for table processing, the other statements may be used to navigate through the table and extract data from cells.

Table 11.6 HTML Table Processing Functions

FUNCTION	DESCRIPTION
boolean changetablepos(*table*, *row*, *value*)	Changes relative table, row, and column position
boolean closehtml( )	Closes an HTML table processing session
boolean findcell(*text*)	Finds the first cell that contains the specified text
boolean findnextcell(*text*)	Finds the next cell that contains the specified text
boolean firstcell( )	Finds the first table cell on the page
boolean firstcol( )	Moves to the first column of the current row in the current table
boolean firstrow( )	Moves to the first cell of the first row of the current table
boolean firsttable( )	Moves to the first cell of the first row of the first table on the page
boolean getcell(*var*)	Retrieves the cell contents of the current table cell location
boolean getcelltext(*var*)	Retrieves the cell contents of the current table cell location, text-only
boolean getcol(*var*)	Retrieves multiple cells from a table column
boolean getrow(*var*)	Retrieves multiple cells from a table row
boolean gettablecount(*var*)	Returns the number of tables on the Web page
boolean gettabledata( )	Retrieves an entire table as delimited data pushed onto the stack
boolean nextcell( )	Advances to the next cell of the current table
boolean nextcol( )	Advances to the next column of the current row of the current table
boolean nextrow( )	Advances to the next row of the current table
boolean nexttable( )	Advances to the next table on the page
boolean openhtml([*url*])	Begins an HTML table processing session
boolean setcol(*n*)	Sets the current table column location
boolean setrow(*n*)	Sets the current table row location
boolean settable(*n*)	Sets the current table number
boolean settablepos(*table*, *row*, *value*)	Sets absolute table, row, and column positions

Closing a Session

The *closehtml* statement (or function) closes a table processing session.

```
closehtml
```

If you fail to close an HTML table processing session, it is closed automatically when the script ends.

Moving Through Tables

To move to the first table on a page, use the *firsttable* statement or function.

```
firsttable
```

A failure condition results if there are no tables on the page. To move on to the next table, the *nexttable* statement may be used. A failure condition results if there are no more tables on the page.

```
nexttable
```

The following code uses *firsttable* and *nexttable* to iterate through all tables on a page.

```
firsttable
while
{
    ...process table values...
    nexttable
}
```

Moving Through Rows

To move to the first row in a table, use the *firstrow* statement or function.

```
firstrow
```

A failure condition results if there are no tables on the page, or no rows in the current table. To move on to the next row, the *nextrow* statement may be used. A failure condition results if there are no more rows in the current table.

```
nextrow
```

The following code uses *firstrow* and *nextrow* to iterate through all rows of a table.

```
firstrow
while
{
    ...process row values...
    nextrow
}
```

Moving Through Columns

To move to the first column in a row, use the *firstcol* statement or function.

```
    firstcol
```

A failure condition results if there are no tables on the page, or no rows in the current table, or no columns in the current row. To move on to the next column, the *nextcol* statement may be used. A failure condition results if there are no more columns in the current row.

```
    nextcol
```

The following code uses *firstcol* and *nextcol* to iterate through all columns of a row.

```
    firstcol
    while
    {
        ...process column value...
        nextcol
    }
```

To move through table cells without stopping at row boundaries, use *firstcell* and *nextcell*.

Moving Through Cells

If you wish, you can deal with table values in terms of *cells* rather than columns. The difference is that cell statements are happy to advance past a row boundary to a new row, and past a table boundary to a new table.

To move to the first cell on a page, use the *firstcell* statement or function.

```
    firstcell
```

A failure condition results if there are no table cells on the page. To move on to the next cell, the *nextcell* statement may be used. A failure condition results if there are no more table cells in the page.

```
    nextcell
```

The following code uses *firstcell* and *nextcell* to iterate through all cells of all tables in a page.

```
firstcell
while
{
    ...process cell value...
    nextcell
}
```

Finding Cells

You can locate cells containing certain text with the *findcell* and *findnextcell* statements. The *findcell* statement locates the first table cell on a page that contains the search text, which is specified as a parameter.

```
findcell text
```

To find the next cell containing specific text, use *findnextcell*, which starts searching from your current position in the page.

```
findnextcell text
```

The following code uses *findcell* and *findnextcell* to find all table cells in a page that contain a dollar sign.

```
findcell "$"
while
{
    ...process cell value...
    findnextcell "$"
}
```

Retrieving Cell Data

Once positioned at a desired table location, the *getcell* statement may be used to retrieve the data in the cell. The parameter to *getcell* is a variable to receive the data.

```
getcell var
```

The entire cell contents are returned, which may include HTML tags as well as text. To retrieve only the text portion of a table cell, use the *getcelltext* statement.

```
getcelltext var
```

The following code iterates through all cells of all tables, retrieving each cell.

```
firstcell
while
{
    getcell data
```

```
      ...do something with data...
      nextcell
  }
```

Changing Table Position

You can change table position by specifying relative or absolute values for table, row, and column. Three statements allow you to specify table, row, or column positions: *settable*, *setrow*, and *setcol*.

```
settable table
setrow row
setcol col
```

Table, row, and column numbers begin with zero. The following code moves to the first table on the page and the fifth row of the table.

```
settable 0
setrow 4
```

To set the table, row, and column positions in a single statement, the *settablepos* statement is used. The parameters are the table, row, and column positions to move to.

```
settablepos table, row, col
```

The following code moves to the second table, first row, third column.

```
settablepos 1, 0, 2
```

To change table position relative to the current position, the *changetablepos* statement is used.

```
changetablepos table, row, col
```

The parameters are relative changes to table, row, and column position. These numbers may be negative, zero, or positive. The following code advances to the next row without changing the table or column numbers.

```
changetablepos 0, 1, 0
```

Negative numbers move earlier in the page. The following example backs up to the previous table.

```
changetablepos -1, 0, 0
```

All of the table position statements set a failure condition if they are unable to move to the specified location.

Retrieving a Table as HTML

The *gettablecount* statement (or function) counts the number of tables in the Web page and returns the count in the specified variable.

```
gettablecount var
```

The following code retrieves the count of tables on a page, then processes each table.

```
openhtml url
gettablecount nTables
for t = nTables
{
    settable t
    firstrow
    {
        ...do something with cell data...
        nextrow
    }
}
```

Tutorial: Quote3

Let's see HTML table processing work in the real world. The script will request a page of multiple stock quotes from a financial Web site, locate the quote information using HTML table processing, and output the captured data as XML. All five of the tutorials in this chapter access the same quote data from the same source, but apply a different type of pattern matching in each case.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Quote3 script.
3. Run the Quote3 script.
4. Understand the Quote3 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Match Web page data using HTML table processing.
- Output data as XML.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Quote3 Script

In the NQL development environment, enter the script shown in Listing 11.4 below and save it under the name Quote3.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Quote3.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Quote3.nql from the Tutorials\ch11 folder.

```
//Quote3 - get a stock quote and extract data,
//         using HTML table processing.

//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"

//use HTML table processing to extract the quote data.

openhtml
findcell "Last Sale" : nextcell : getcell LastSale
findcell "Open" : nextcell : getcell Open
findcell "Change" : nextcell : getcell Change
findcell "Change %" : nextcell : getcell Percent
findcell "Volume" : nextcell : getcell Volume
findcell "High" : nextcell : getcell High
findcell "Low" : nextcell : getcell Low
closehtml

//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

Listing 11.4 Quote3 script.

At this point, the script in Listing 11.4 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Since the quote site for this script is imaginary (a real quote site would have changed layout by the time you read this book), a local copy of the expected HTML is included in the \Tutorial\ch11 directory. The script will load this page rather than perform an actual Internet access. The HTML we will be scanning is shown earlier in the chapter in Listing 11.2.

Step 3: Run the Quote3 Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of stock quotes.

```
<LastSale>65.80</LastSale><Open>65.65</Open><Change>0.33</Change><Perce
nt>0.50</Percent><Volume>21,098,400</Volume><High>66.88</High><Low>65.5
4</Low>
```

If this does not happen, check the following:

- Check your script for typographical errors.
- Check that your script and quote.htm are in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Quote3 script run, retrieving a Web page and locating stock quote data. Although we haven't discussed how it works yet, you've witnessed NQL putting HTML table processing to use. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Quote3 Script

We now want to make sure we understand every part of the Quote3 script. The first two lines are comment lines.

```
//Quote3 - get a stock quote and extract data,
//         using HTML table processing.
```

The stock quote is retrieved. In a real script, a *get* statement would be used to access an Internet quote page. Here, we comment out the *get* statement and use *load* to load the stock quote HTML page provided with the tutorial. The net effect is the same: An HTML page is on the stack.

```
//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"
```

Now to extract the data. An *openhtml* statement analyzes the HTML as table data. For each field, a *findcell* statement locates the label we are interested in. A *nextcell* statement advances to the next table cell, which contains the data we are interested in. A *getcell* statement retrieves the cell contents into a variable.

```
//use HTML table processing to extract the quote data.

openhtml
findcell "Last Sale" : nextcell : getcell LastSale
findcell "Open" : nextcell : getcell Open
findcell "Change" : nextcell : getcell Change
findcell "Change %" : nextcell : getcell Percent
findcell "Volume" : nextcell : getcell Volume
findcell "High" : nextcell : getcell High
findcell "Low" : nextcell : getcell Low
closehtml
```

If the pattern match is successful, the quote fields have been stored in variables. They can now be output in XML format with the *output* statement.

```
//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

At this point, you've seen an NQL script interact with a Web site and capture results using HTML table processing.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Amend the script to work with a real quote site.
- Add error checking, so that a *get* failure is detected and reported.

Regular Expressions

If you have experience with UNIX systems or languages like Perl, you may be used to the form of pattern matching called *regular expressions*. Regular expressions are combinations of the elements listed in Table 11.7.

Regular expression syntax can take some getting used to, so let's consider some examples. The following regular expression would match this sentence.

```
The following regular expression would match this sentence.
```

Table 11.7 Regular Expression Elements

ELEMENT	MEANING
`a`	Match the given character.
`.`	Match any single character except a newline.
`[a-z]`	Match any single character in the range.
`[^a-z]`	Match any single character not in the range.
`\d`	Match any single decimal digit; same as [0-9].
`\D`	Inverse of \d; same as [^0-9].
`\s`	Match any single white-space character.
`\S`	Inverse of \s; match any non-white-space character.
`\w`	Match any character that is part of a word; same as [a-zA-Z0-9_].
`\W`	Inverse of \w; same as [^a-zA-Z0-9_].
`\b`	Match a word boundary.
`\B`	Match a non-word boundary.
`^$`	Match beginning or end of given input or line, respectively.
`\n`	Match a newline.
`\r`	Match a carriage return.
`\t`	Match a tab.
`\f`	Match a form feed.
`\0`	Match a null.
`()`	Parentheses surround the data to be captured.
`char*`	Match character zero or more times.
`char+`	Match character one or more times.
`char?`	Match character zero or one times.
`char{n}`	Match character exactly n times.
`char{n,}`	Match character at least n times.
`char{n,m}`	Match character at least n and at most m times.
`char1|char2`	Match char1 or char2.

The following expression would also match, but uses wildcard values for the center of the sentence. The period matches any character, and following it with an asterisk matches any number of characters.

```
The following regular expression .* this sentence.
```

Let's consider a third version of the pattern, in which we want to capture some of the data. In regular expressions, parentheses are used to indicate which parts of the pattern are to be extracted. This form of the pattern would capture the words "would match".

```
The following regular expression (.*) this sentence.
```

Finding a Match

The *matchregexp* statement (or function) matches a regular expression against the HTML on the stack. The parameters are a regular expression and a list of variables to receive data values.

```
matchregexp pattern, var-list
```

If an occurrence of the pattern is located, a success condition results and variables are created reflecting the data. The HTML on the stack is shortened; all that remains is the portion after the pattern match. If the pattern cannot be located, a failure condition results and the HTML is removed from the stack. The following code retrieves a Web page, then uses regular expression pattern matching to locate the three parts of a social security number. Note the use of \d to match a digit, and the use of parentheses to indicate the parts of the pattern that are to be returned in variables.

```
get URL
matchregexp "(\d\d\d)-(\d\d)-(\d\d\d\d)", ss1, ss2, ss3
```

Tutorial: Quote4

Let's see regular expressions work in the real world. The script will request a page of multiple stock quotes from a financial Web site, locate the quote information using regular expressions, and output the captured data as XML. All five of the tutorials in this chapter access the same quote data from the same source, but apply a different type of pattern matching in each case.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Quote4 script.
3. Run the Quote4 script.
4. Understand the Quote4 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Match Web page data using regular expressions.
- Output data as XML.

Let’s begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you’re ready to enter the tutorial script.

Step 2: Enter the Quote4 Script

In the NQL development environment, enter the script shown in Listing 11.5 and save it under the name Quote4.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//Quote4 - get a stock quote and extract data,
//         using regular expression pattern matching.

//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"

//perform a regular expression match to get the quote data.

matchregexp "Last Sale.*<TD>(.*)</TD>", LastSale
matchregexp "Open.*<TD>(.*)</TD>", Open
matchregexp "Change.*<TD>(.*)</TD>", Change
matchregexp "Volume.*<TD>(.*)</TD>", Volume
matchregexp "High.*<TD>(.*)</TD>", High
matchregexp "Low.*<TD>(.*)</TD>", Low

//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

Listing 11.5 Quote4 script.

If you prefer, you may copy Quote4.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Quote4.nql from the Tutorials\ch11 folder.

At this point, the script in Listing 11.5 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Since the quote site for this script is imaginary (a real quote site would have changed layout by the time you read this book), a local copy of the expected HTML is included in the \Tutorial\ch11 directory. The script will load this page rather than perform an actual Internet access. The HTML we will be scanning is shown previously in Listing 11.2.

Step 3: Run the Quote4 Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of stock quotes.

```
<LastSale>65.80</LastSale><Open>65.65</Open><Change>0.33</Change><Perce
nt>0.50</Percent><Volume>21,098,400</Volume><High>66.88</High><Low>65.5
4</Low>
```

If this does not happen, check the following:

- Check your script for typographical errors.
- Check that your script and quote.htm are in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Quote4 script run, retrieving a Web page and locating stock quote data. Although we haven't discussed how it works yet, you've witnessed NQL putting regular expressions to use. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Quote4 Script

We now want to make sure we understand every part of the Quote4 script. The first two lines are comment lines.

```
//Quote4 - get a stock quote and extract data,
//         using regular expression pattern matching.
```

The stock quote is retrieved. In a real script, a *get* statement would be used to access an Internet quote page. Here, we comment out the *get* statement and use *load* to load the stock quote HTML page provided with the tutorial. The net effect is the same: An HTML page is on the stack.

```
//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"
```

Now for the pattern match. In regular expression pattern matching, the *matchregexp* statement is used along with a regular expression pattern. Areas to be captured are enclosed in parentheses in the pattern, and variable names are supplied as parameters. The first *matchregexp* statement searches for "Last Sale", followed by any number of characters (.*), followed by a start table tag (<TD>), followed by any number of characters (captured because of the enclosing parentheses), followed by an ending table tag (</TD>). The captured data is stored in the variable *LastSale*. The remaining fields are captured in the same way.

```
//perform a regular expression match to get the quote data.

matchregexp "Last Sale.*<TD>(.*)</TD>", LastSale
matchregexp "Open.*<TD>(.*)</TD>", Open
matchregexp "Change.*<TD>(.*)</TD>", Change
matchregexp "Volume.*<TD>(.*)</TD>", Volume
matchregexp "High.*<TD>(.*)</TD>", High
matchregexp "Low.*<TD>(.*)</TD>", Low
```

If the pattern match is successful, the quote fields have been stored in variables. They can now be output in XML format with the *output* statement.

```
//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

At this point, you've seen an NQL script interact with a Web site and capture results using regular expression pattern matching.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Amend the script to work with a real quote site.
- Add error checking, so that a *get* failure is detected and reported.

Web Queries

Web queries are a form of pattern matching that resembles SQL, the standard language for querying databases. If you're familiar with SQL database queries, you'll find Web queries easy to pick up. A Web query contains several clauses that identify the attributes of the data you wish to mine from a Web page. A sample Web query is shown on the following page.

```
SELECT PartNumber, Desc, RetailPrice, DiscountPrice
WHERE
    PartNumber is text,
    Desc is text,
    RetailPrice is amount,
    DiscountPrice is amount
```

Web Query Syntax

A Web query is similar to an SQL query, as the following example shows.

```
SELECT PartNumber, Desc, RetailPrice, DiscountPrice
WHERE
    PartNumber is text,
    Desc is text,
    RetailPrice is amount,
    DiscountPrice is amount
```

A Web query contains several clauses that identify the attributes of the data you wish to mine from a Web page. Every Web query potentially contains a SELECT clause, an IN clause, and a WHERE clause.

The SELECT Clause

The first part of a Web query is a SELECT clause, which is required. It lists the names of the data to be extracted into NQL variables.

```
SELECT var-list
```

The IN Clause

The next part of a Web query, which is optional, is an IN clause. The keyword IN is followed by a phrase that reflects a relationship between the data items, such as being in the same table row.

```
IN grouping
```

The valid IN clause phrasings are listed in Table 11.8

PLANNED FEATURE **The IN clauses marked with an asterisk are planned features.**

The WHERE Clause

The next part of a Web query is its WHERE clause, which is required. It lists criteria for each variable mentioned in the SELECT clause: Every term must have a criterion, and each criterion is separated by a comma.

Table 11.8 IN Clauses

PHRASE	DESCRIPTION
in same table	All elements are within the same table.
in same row	All elements are in the same row of a table.*
in same column	All elements are in the same column of a table.*
in same cell	All elements are in the same table cell.*
in same section	All elements are in the same logical grouping.
in same listitem	All elements are part of the same list item.

```
WHERE var1 is criterion1,
      var2 is criterion2,
             ...
      varN is criterionN
```

Each criterion must specify a data type. The Web query data types are listed in Table 11.9.

In addition to a data type, each criterion may also specify attributes. Any number of attributes may be specified for a variable. Table 11.10 lists the Web query attributes.

In addition to the data type keywords and attribute keywords, the words *is*, *a*, and *an* may be freely used in a query to make it more readable.

There are many possible combinations of data type and attributes, so let's take a look at one. Imagine you wanted to retrieve news headlines from a news site. The *phrase* data type will match a headline, but it will also match plenty of other things on the page besides headlines. You decide that one distinctive factor of a headline is that it is enclosed within a hyperlink, so you decide to specify the *linked* attribute. Your query looks like this:

```
SELECT headline
WHERE headline is a linked phrase
```

Querying a Web Page

The *webquery* statement issues a Web query against an HTML Web page. There are two forms of the *webquery* statement: single-line and multi-line. The single-line form is the keyword *webquery* which takes one parameter, the query to execute.

```
webquery query
```

The HTML on the stack is scanned. If a match is found, variables are set based on the query and a success condition results. If no match is found, a failure condition results. Successive matches can be found with the *nextquery* statement. The following code uses a simple Web query to extract all of the prices from a Web page.

Table 11.9 Web Query Data Types

KEYWORD	DESCRIPTION
ad	image tag to an advertising banner
amount	currency amount
company	company name
currency	currency amount
day	day of the month
exchange	stock exchange
float	floating-point number
fraction	fractional number
graphic	image tag
image	image tag
integer	integer (whole number)
link	hyperlink
month	month of the year
name	proper name
number	general number (integer, float, fraction, currency)
paragraph	paragraph of text
percent(age)	percentage
phrase	phrase (one or more words of text)
picture	image tag
price	currency amount
sentence	sentence
shareprice	integer and/or fraction
symbol	ticker symbol
tag	HTML tag
text	text
ticker	ticker symbol
time	time of day
timezone	time zone

continues

Table 11.9 Web Query Data Types (continued)

KEYWORD	DESCRIPTION
ticker	ticker symbol
weekday	name of a day of the week
word	single word
year	calendar year

Table 11.10 Web Query Attributes

KEYWORD	DESCRIPTION
after *term*	Variable appears after the specified term.
before *term*	Variable appears before the specified term.
cell	Variable is in a table cell.
containing *text*	Variable contains the specified text.
emphasis	Variable is emphasized.
emphasized	Variable is emphasized.
heading	Variable is in a heading.
linked	Variable is within a hyperlink.
listitem	Variable is part of a list item.
lowercase	Variable is made up of lowercase characters.
matching *regular-expression*	Variable matches the specified regular expression pattern.
named *text*	Variable has the specified text as a label or table heading.
non	Variable reverses the meaning of the criterion.
not	Variable reverses the meaning of the criterion.
optional	Term does not have to be present for query to match overall.
uppercase	Variable is made up of uppercase characters.
within *term*	Variable is contained within another term.

```
get url
webquery "SELECT price WHERE price is an amount"
while
{
    ...do something with price...
    nextquery
}
endquery
```

The multi-line form of the *webquery* statement allows the query to be written over multiple lines, without the need for quotation marks.

```
webquery
{
    multi-line query
}
```

There is no functional difference between the single-line and multi-line forms of *webquery*. The single-line form has the advantage that any expression can be supplied as a Web query, whereas in the multi-line form the query is in-line. The multi-line form is easier to use for lengthy queries that would be awkward to cram into a single line. The following Web query shows the use of the multi-line form of *webquery*.

```
get url
webquery
{
    select Title, Author, Price
    in same table
    where
        Title is phrase,
        Author is name after Title,
        Price is amount after Title
}
while
{
    ...do something with query results...
    nextquery
}
endquery
```

Note that the *webquery* statement is not destructive, as are all of NQL's other forms of pattern matching. The HTML on the stack is not reduced in any way.

Finding the Next Query Match

The *matchquery* statement finds only the first match in the HTML. To find successive matches, it is necessary to use the *nextquery* statement, which repeats a Web query against the HTML on the stack, continuing on from where the last query left off.

```
nextquery
```

If a match is found, a success condition results and variables are set based on the contents of the query. If no more matches are found, a failure condition results. The following code uses *webquery* and *nextquery* to find all table cells in a page.

```
get url
webquery "SELECT data WHERE data is a cell"
while
{
    ...do something with data...
    nextquery
}
endquery
```

Ending a Query

A Web query consumes resources. Those resources are released when you perform a new Web query, or when a script ends. To release the resources sooner, the *endquery* statement may be used.

```
endquery
```

It is good form to supply *endquery* after a sequence of using *webquery* and *nextquery*. The following code shows the use of *endquery*.

```
get url
webquery
{
    select link, headline
    where
        link is a link,
        headline is a linked phrase within link
}
while
{
    ...do something with link and headline...
    nextquery
}
endquery
```

Tutorial: Quote5

Let's see Web queries work in the real world. The script will request a page of multiple stock quotes from a financial Web site, locate the quote information using a Web query, and output the captured data as XML. All five of the tutorials in this chapter access the same quote data from the same source, but apply a different type of pattern matching in each case.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Quote5 script.
3. Run the Quote5 script.
4. Understand the Quote5 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve a Web page with Web primitives.
- Match Web page data using regular expressions.
- Output data as XML.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Quote5 Script

In the NQL development environment, enter the script shown in Listing 11.6 and save it under the name Quote5.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Quote5.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Quote5.nql from the Tutorials\ch11 folder.

At this point, the script in Listing 11.6 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Since the quote site for this script is imaginary (a real quote site would have changed layout by the time you read this book), a local copy of the expected HTML is included in the \Tutorial\ch11 directory. The script will load this page rather than perform an actual Internet access. The HTML we will be scanning is shown earlier in the chapter in Listing 11.2.

```
//Quote5 - get a stock quote and extract data,
//         using web queries.

//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"

//perform a web query match to get the quote data.

webquery
{
       select LastSale, Open, Change, Percent, Volume, High, Low
       where
            LastSale is a number named "Last Sale",
            Open is a number named "Open",
            Change is a number named "Change",
            Percent is a number named "Change %",
            Volume is a number named "Volume",
            High is a number named "High",
            Low is a number named "Low"
}

//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

Listing 11.6 Quote5 script.

Step 3: Run the Quote5 Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Once the script ends, the *Output* tab of the development environment should contain an XML stream of stock quotes.

```
<LastSale>65.80</LastSale><Open>65.65</Open><Change>0.33</Change><Perce
nt>0.50</Percent><Volume>21,098,400</Volume><High>66.88</High><Low>65.5
4</Low>
```

If this does not happen, check the following:

- Check your script for typographical errors.
- Check that your script and quote.htm are in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Quote5 script run, retrieving a Web page and locating stock quote data. Although we haven't discussed how it works yet, you've witnessed NQL putting Web queries to use. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Quote5 Script

We now want to make sure we understand every part of the Quote5 script. The first two lines are comment lines.

```
//Quote5 - get a stock quote and extract data,
//         using web queries.
```

The stock quote is retrieved. In a real script, a *get* statement would be used to access an Internet quote page. Here, we comment out the *get* statement and use *load* to load the stock quote HTML page provided with the tutorial. The net effect is the same: An HTML page is on the stack.

```
//get the stock quote

//get "quote.my-site.com/quote/stocks/quotes.asp?symbols=msxx"
load "quote.htm"
```

Now for the Web query. A *webquery* statement issues the query, which is contained on multiple lines in the code block. The *select* clause of the Web query lists the terms to be extracted. For each term, there is a description of attributes in the *where* clause of the query. If the query is successful, each term is mapped to a variable of the same name.

```
//perform a web query match to get the quote data.

webquery
{
     select LastSale, Open, Change, Percent, Volume, High, Low
     where
          LastSale is a number named "Last Sale",
          Open is a number named "Open",
          Change is a number named "Change",
          Percent is a number named "Change %",
          Volume is a number named "Volume",
          High is a number named "High",
          Low is a number named "Low"
}
```

If the pattern match is successful, the quote fields have been stored in variables. They can now be output in XML format with the *output* statement.

```
//output the results as XML

output LastSale, Open, Change, Percent, Volume, High, Low
```

At this point, you've seen an NQL script interact with a Web site and capture results using a Web query.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Amend the script to work with a real quote site.
- Add error checking, so that a *get* failure is detected and reported.

HTML Operations

In addition to the pattern matching operations described earlier in the chapter, NQL can perform other operations on HTML. The HTML-related statements are listed in Table 11.11.

The HTML statements are also available as functions. Table 11.12 lists the HTML functions.

Table 11.11 HTML Statements

STATEMENT	PARAMETERS	DESCRIPTION
cliptable	[*table-number*]	Extracts a table from a page by table number
cliptable	*text*	Extracts a table from a page that contains the specified text
removetags	[*var-list*]	Removes HTML or XML tags from a string
tablerow	*var-list*	Locates the next table row in the current table and extracts cells into variables

Table 11.12 HTML Functions

FUNCTION	DESCRIPTION
boolean cliptable([*table-number*])	Extracts a table from a page by table number
boolean cliptable(*text*)	Extracts a table from a page that contains the specified text
boolean removetags([*var-list*])	Removes HTML or XML tags from a string
boolean tablerow(*var-list*)	Locates the next table row in the current table and extracts cells into variables

Each of the HTML statements and functions are described individually in the following section.

Clipping a Table

To extract one table's HTML out of a page, the *cliptable* statement may be used. The *cliptable* statement works directly with HTML on the stack and does not require or use an HTML table processing session.

The parameter is a table number or some text that the table must contain. Table numbers start with 1.

```
cliptable [number]
cliptable text
```

If the specified table is located, a success condition results and the full page HTML on the stack is replaced with just the table's HTML. The following code retrieves the HTML of a table that contains the text *price*.

```
push Page
cliptable "price"
pop PriceTable
```

Extracting Data from a Table Row

The *tablerow* statement extracts data from each column of a table row and returns the data in variables. The *tablerow* statement works directly with HTML on the stack and does not require or use an HTML table processing session. The parameters are a variable list to receive the table data.

```
tablerow var-list
```

If a table row is found, a success condition results and each variable specified receives a table cell value. The following code processes each row of a price table.

```
get url
while tablerow(PartNo, Desc, Amount)
{
    ...do something with PartNo, Desc, and Amount...
}
```

Removing Tags

The *removetags* statement removes tags from HTML or XML, leaving just text as a remainder. One or more variables may be specified to *removetags*. Each variable's contents are detagged. If no parameters are specified, the top value on the stack is detagged.

```
removetags [var-list]
```

The following code retrieves a Web page, removes its tags, and saves the result as a text file.

```
get url
removetags
save "file.txt"
```

Chapter Summary

Network Query Language supports a rich offering of tools for extracting data from Web pages. There are five types of pattern matching.

HTML pattern matching uses patterns of HTML tags and text. An intricate knowledge of a Web page's HTML is required.

- The *match* statement performs a pattern match, setting variables and shortening the HTML on the stack.
- The *nextmatch* statement repeats the last match.

Data pattern matching looks for sequences of logical data types. Often, a pattern can be deduced just from viewing a Web page in a browser without having to study the underlying HTML.

- The *matchdata* statement performs a pattern match, setting variables and shortening the HTML on the stack.

HTML table processing iterates through tables, rows, columns, and cells. Web data stored in tables can be easily located and extracted with HTML table processing.

- The *changetablepos* statement changes relative table, row, and column position.
- The *closehtml* statement closes an HTML table processing session.
- The *findcell* statement finds the first cell that contains the specified text.
- The *findnextcell* statement finds the next cell that contains the specified text.
- The *firstcell* statement finds the first table cell on the page.
- The *firstcol* statement moves to the first column of the current row in the current table.
- The *firstrow* statement moves to the first cell of the first row of the current table.
- The *firsttable* statement moves to the first cell of the first row of the first table on the page.
- The *getcell* statement retrieves the cell contents of the current table cell location.
- The *getcelltext* statement retrieves the cell contents of the current table cell location, text-only.
- The *getcol* statement retrieves multiple cells from a table column.

- The *getrow* statement retrieves multiple cells from a table row.
- The *gettablecount* statement returns the number of tables on the Web page.
- The *gettabledata* statement retrieves an entire table as delimited data pushed onto the stack.
- The *nextcell* statement advances to the next cell of the current table.
- The *nextcol* statement advances to the next column of the current row of the current table.
- The *nextrow* statement advances to the next row of the current table.
- The *nexttable* statement advances to the next table on the page.
- The *openhtml* statement begins an HTML table processing session.
- The *setcol* statement sets the current table column location.
- The *setrow* statement sets the current table row location.
- The *settable* statement sets the current table number.
- The *settablepos* statement sets absolute table, row, and column positions.

Regular expressions allow Perl-style/Unix-style regular expressions to be used for expressing patterns.

- The *matchregexp* statement performs a pattern match with regular expressions, setting variables and shortening the HTML on the stack.

Web queries allow Web pages to be queried using SQL-like queries. Data types and attributes are described in the query.

- The *webquery* statement executes a Web query, setting variable values. The HTML on the stack is unchanged.
- The *nextquery* statement repeats the previous query.
- The *endquery* statement ends a query and frees up memory.

Nearly any Web page can be effectively processed with one or more of these five techniques.

NQL also contains statements for performing HTML operations.

- The *cliptable* statement clips a table from an HTML page.
- The removetags statement *detags* HTML, leaving text.
- The *tablerow* statement retrieves the contents of a table row into variables.

NQL scripts can make use of a varied collection of statements for navigating through Web pages and extracting data from them.

CHAPTER

12

Sending and Receiving Email

Email has become the predominant application for which people use computers. The ability to send or receive email programmatically adds a powerful dimension to your applications. NQL scripts can send mail as a means of notifying users or delivering content. Scripts can also scan incoming mail messages and take action on them, such as generating responses, deleting unwanted mail, or routing messages to the best party to handle them.

Network Query Language allows your programs to send and receive messages in a number of formats, including text and HTML, and supports file attachments. Several mail protocols are supported. This chapter describes the various email protocols and then covers how email is sent and received in NQL. In the tutorial at the end of the chapter you will create a program that scans email messages and automatically acts on some of them.

An Overview of NQL's Email Capabilities

NQL allows your scripts to act as a mail client, connecting to a mail server to send or receive mail messages. NQL scripts may perform the following email functions:

- Connect to a mail server
- Send mail messages

- Attach files to messages
- Receive messages
- Save message file attachments
- Disconnect from a mail server

Let's take a quick look at what it's like to work with email in NQL. The following script is a complete program for sending a mail message.

```
openmail
push "Thanks for your order. You'll receive a shipping notice soon."
sendmessage "Order confirmation", "jjeffries@my-web-site.com"
closemail
```

As you can see, email in NQL is easy and straightforward. Let's study this script line-by-line:

- *openmail* connects to a mail server.
- *push* stores a mail message on the stack.
- *sendmessage* sends the mail message.
- *closemail* disconnects from the mail server.

Email Protocols

There are a variety of methods for implementing email. One of the reasons there are so many choices is that email must work with local networks as well as the Internet. Developments in both areas have led to a number of email standards. Many mail servers today support both network and Internet mail standards.

NQL allows your script to act as a mail client. You may choose from the following mail protocols:

Mail Application Programming Interface (MAPI)

Post Office Protocol 3 (POP3)

Simple Mail Transfer Protocol (SMTP)

The MAPI protocol is specific to mail servers on Microsoft platforms, whereas POP3 and SMTP are standard Internet protocols universally available for all platforms. In all cases, the architecture is client/server based, as shown in Figure 12.1. Mail programs on desktops issue commands to mail servers and receive back responses.

MAPI

The MAPI protocol evolved from the idea of harmonizing the many mail clients and mail server products in the Microsoft Windows world. It was theorized that a properly designed interface would allow any mail client to work with any mail server, even if they came from different software vendors. This would, for example, allow Lotus

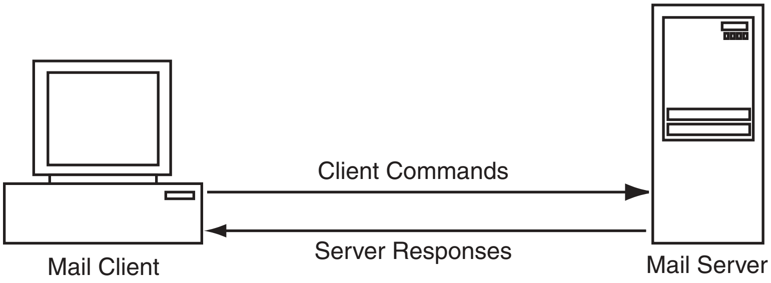


Figure 12.1 Email client/server architecture.

Notes to act as a mail server in an organization yet permit the use of Microsoft Outlook on desktops as a mail client.

With MAPI, mail clients make service calls to the Windows operating system, which in turn takes care of communicating with the network mail server.

POP3

The POP3 protocol is the most popular of the Internet standards for reading mail. With POP3, commands and responses are communicated with TCP/IP. The mail client and the mail server do not have to be on the same local network, as the Internet is the transport medium.

The POP3 protocol requires authentication: POP3 clients are usually required to specify an account ID and password in order to gain access to mail.

Although POP3 is usually paired with SMTP, the two are separate standards.

SMTP

The SMTP protocol is the most popular of the Internet standards for sending mail. With SMTP, commands and responses are communicated with TCP/IP. The mail client and the mail server do not have to be on the same local network, as the Internet is the transport medium.

Most SMTP servers do not require authentication: It is possible to fib with SMTP. Any email address and logical name can be specified when a message is sent. This characteristic is sometimes useful, such as when sending a mailing where you want any replies routed to a different email address than the sender of the original message. This characteristic can also be abused, as in the case of spamming or Internet fraud.

Although SMTP is usually paired with POP3, the two are separate standards. An SMTP server is not necessarily a POP3 server.

Message Formats

The original format for email was plain text, and that is still the form that can be read most universally. Other formats allow for sophisticated content, including the use of typefaces, color, images, special characters, and hyperlinks. NQL supports three message formats: Text, Rich Text Format (RTF), and HTML.

Text Messages

Text messages are the most universally accepted form of mail message, because all mail systems can handle plain text. Although most modern mail clients are able to accommodate fancier formats such as RTF and HTML, it is a good idea to send a text version of a mail message as well, especially when a message is being sent to a large distribution list.

RTF Messages

Rich Text Format allows your messages to have the same basic features as a word-processing document. Text may be rendered in varying typefaces and sizes. Text may be emphasized using bold, underline, italics, color, and other means. In order for an RTF format message to be understood, the recipient needs a mail client capable of interpreting RTF and rendering it properly. Otherwise, the mail message will look very confusing to the recipient. Listing 12.1 shows a small RTF mail message.

With an RTF-capable mail client, this message is rendered as shown in Figure 12.2.

HTML Messages

HTML allows your messages to have the same basic features as a Web page. Message text can be rendered in varying typefaces and sizes. Text may be emphasized using bold, underline, italics, color, and other means. You may also embed images, hyperlinks, and controls (such as buttons) in an HTML message. In order for an HTML format message to be understood, the recipient needs a mail client capable of interpreting HTML and rendering it properly. Otherwise, the mail message will look very confusing to the recipient. Listing 12.2 shows a small HTML mail message.

With an HTML-capable mail client, this message is rendered as shown in Figure 12.3.

Combining Multiple Formats

Using HTML or RTF for mail presents a problem. In exchange for the sophistication of those formats, there is the risk that some recipients may not have mail clients that can handle them. This situation can be overcome by sending a message in more than one format. For example, a mail message can have an HTML portion and a text portion. The recipient's mail client will choose the best format of those it supports. Thus, a modern mail client such as Microsoft Outlook would display HTML, whereas a less sophisticated mail client could still display the plain-text version of the message.

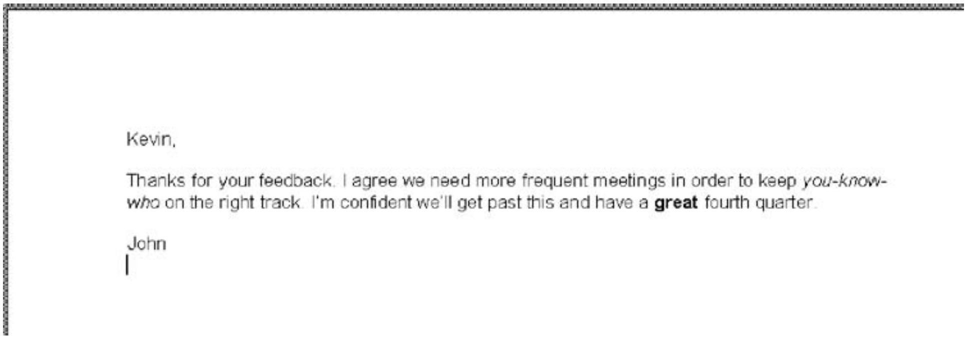
```
{\rtf1\ansi\ansicpg1252\uc1 \deff0\deflang1033\deflangfe1033{\fonttbl
{\f0\froman\fcharset0\fprq2{\*\panose 02020603050405020304}Times New
Roman;}{\f1\fswiss\fcharset0\fprq2{\*\panose 020b0604020202020204}
Arial;}
{\f5\fswiss\fcharset0\fprq2{\*\panose 020b0604020202020204}Helvetica;}
{\f74\froman\fcharset238\fprq2 Times New Roman CE;}{\f75\froman\
fcharset204\fprq2 Times New Roman Cyr;}{\f77\froman\fcharset161\fprq2
Times New Roman Greek;}
{\f78\froman\fcharset162\fprq2 Times New Roman Tur;}{\f79\froman\
fcharset177\fprq2 Times New Roman (Hebrew);}{\f80\froman\
fcharset178\fprq2 Times New Roman (Arabic);}{\f81\froman\fcharset186\
fprq2 Times New Roman Baltic;}
{\f82\fswiss\fcharset238\fprq2 Arial CE;}{\f83\fswiss\
fcharset204\fprq2 Arial Cyr;}{\f85\fswiss\fcharset161\fprq2 Arial
Greek;}{\f86\fswiss\fcharset162\fprq2 Arial Tur;}{\f87\fswiss\
fcharset177\fprq2 Arial (Hebrew);}
{\f88\fswiss\fcharset178\fprq2 Arial (Arabic);}{\f89\fswiss\
fcharset186\fprq2 Arial Baltic;}}{\colortbl;\red0\green0\blue0;\
red0\green0\blue255;\red0\green255\blue255;\red0\green255\blue0;\
red255\green0\blue255;\red255\green0\blue0;\red255\green255\blue0;
\red255\green255\blue255;\red0\green0\blue128;\red0\green128\blue128;\
red0\green128\blue0;\red128\green0\blue128;\red128\green0\blue0;\
red128\green128\blue0;\red128\green128\blue128;\red192\green192\
blue192;}{\stylesheet{
\ql\li0\ri0\widctlpar\aspalpha\aspnum\faauto\adjustright\rin0\lin0\
itap0 \f1\fs22\lang1033\langfe1033\cgrid\langnp1033\langfenp1033 \
snext0 Normal;}{\s1\ql \li0\ri0\sb240\sa60\keepn\widctlpar\aspalpha\
aspnum\faauto\adjustright\rin0\lin0\itap0
\b\f5\fs36\lang1033\langfe1033\kerning32\cgrid\langnp1033\langfenp1033
\sbasedon0 \snext0 heading 1;}{\s2\ql \li0\ri0\sb240\sa60\keepn\
widctlpar\aspalpha\aspnum\faauto\adjustright\rin0\lin0\itap0
\b\f1\fs32\lang1033\langfe1033\cgrid\langnp1033\langfenp1033 \
sbasedon0 \snext0 heading 2;}{\s3\ql \li0\ri0\sb240\sa60\keepn\
widctlpar\aspalpha\aspnum\faauto\adjustright\rin0\lin0\itap0 \b\f1\
fs28\lang1033\langfe1033\cgrid\langnp1033\langfenp1033
\sbasedon0 \snext0 heading 3;}{\*\cs10 \additive Default Paragraph
Font;}}{\info{\title Kevin,}{\author David Pallmann}{\operator David
Pallmann}{\creatim\yr2001\mo6\dy3\hr14\min36}{\revtim\yr2001\mo6\dy3\
hr14\min38}{\version1}{\edmins2}{\nofpages1}
{\nofwords0}{\nofchars0}{\*\company NQL Inc.}{\nofcharsws0}
{\vern8269}}\margl1440\margr1440 \widowctrl\ftnbj\aenddoc\noxlattoyen\
expshrtn\noultrlspc\dntblnsbdb\nospaceforul\formshade\horzdoc\
dgmargin\dghspace180\dgvspace180\dghorigin1440\dgvorigin1440
\dghshow1\dgvshow1\jexpand\viewkind1\viewscale114\viewzk2\pgbrdrhead\
pgbrdrfoot\splytwnine\ftnlytwnine\htmautsp\nolnhtadjtbl\useltbaln\
alntblind\lytcalctblwd\lyttblrtgr\lnbrkrule \fet0\sectd \linex0\
endnhere\sectlinegrid360\sectdefaultcl {\*\pnseclvl1
```

continues

Listing 12.1 RTF message.

```
\pnucrm\pnstart1\pnindent720\pnhang{\pntxta
.}}{\*\pnseclvl2\pnucltr\pnstart1\pnindent720\pnhang{\pntxta
.}}{\*\pnseclvl3\pndec\pnstart1\pnindent720\pnhang{\pntxta
.}}{\*\pnseclvl4\pnlcltr\pnstart1\pnindent720\pnhang{\pntxta
)}}{\*\pnseclvl5
\pndec\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta )}}
{\*\pnseclvl6\pnlcltr\pnstart1\pnindent720\pnhang{\pntxtb (}{\pntxta
)}}{\*\pnseclvl7\pnlcrm\pnstart1\pnindent720\pnhang{\pntxtb (}
{\pntxta )}}{\*\pnseclvl8\pnlcltr\pnstart1\pnindent720\pnhang
{\pntxtb (}{\pntxta )}}{\*\pnseclvl9\pnlcrm\pnstart1\pnindent720\
pnhang{\pntxtb (}{\pntxta )}}\pard\plain \ql \li0\ri0\widctlpar\
aspalpha\aspnum\faauto\adjustright\rin0\lin0\itap0 \f1\fs22\
lang1033\langfe1033\cgrid\langnp1033\langfenp1033 {Kevin,
\par
\par Thanks for your feedback. I agree we need more frequent meetings
in order to keep }{\i you-know-who}{ on the right track. I\rquote m
confident we\rquote ll get past this and have a }{\b great}{ fourth
quarter.
\par
\par John
\par
\par }}
```

Listing 12.1 RTF message (continued).


Kevin,

Thanks for your feedback. I agree we need more frequent meetings in order to keep *you-know-who* on the right track. I'm confident we'll get past this and have a **great** fourth quarter.

John

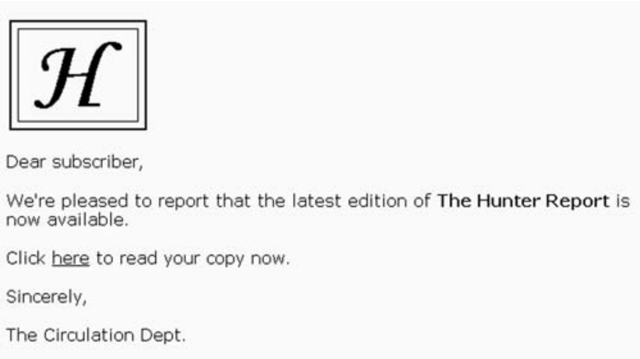
Figure 12.2 RTF message rendered in a mail client.

File Attachments

Regardless of the format(s) you use for mail messages, you may also attach files to mail messages. All three of the email formats that NQL supports (MAPI, POP3, and SMTP) allow one or more files to be attached to mail messages. The attachments, of course, can be files of any kind, such as documents, spreadsheets, programs, or multimedia files.

```
<html>

<head>
<meta http-equiv="Content-Language" content="en-us">
<meta http-equiv="Content-Type" content="text/html; charset=windows-
1252">
<meta name="GENERATOR" content="Microsoft FrontPage 4.0">
<meta name="ProgId" content="FrontPage.Editor.Document">
<title>New Page 2</title>
</head>

<body>

<p><img border="0" src="wpe1.gif" width="138" height="112"></p>
<p><font face="Verdana">Dear subscriber,</font></p>
<p><font face="Verdana">We're pleased to report that the latest
edition of <font size="3"><b>The
Hunter Report</b></font> is now available.</font></p>
<p><font face="Verdana">Click <u>here</u> to read your copy
now.</font></p>
<p><font face="Verdana">Sincerely,</font></p>
<p><font face="Verdana">The Circulation Dept.</font></p>
<p>&nbsp;</p>

</body>

</html>
```

Listing 12.2 HTML message.



Dear subscriber,

We're pleased to report that the latest edition of **The Hunter Report** is now available.

Click here to read your copy now.

Sincerely,

The Circulation Dept.

Figure 12.3 HTML message rendered in a mail client.

Email Support in NQL

NQL includes 18 statements related to email, but only a handful are needed on a regular basis. The two most common sequences of operations are as follows:

1. Sending email: Connect to a mail server, compose a message, add any attachments, send message, and disconnect from the mail server.
2. Reading email: Connect to a mail server, iterate through new messages (or all messages), save any attachments, and disconnect from the mail server.

The first sequence, sending email, typically involves the use of *openmail*, *push*, and possibly *messageattachment*, *sendmessage*, and *closemail* statements. For example:

```
openmail
...compose message in some way and push it on the stack...
sendmessage "Final notice", "kbh@my-customer.com"
closemail
```

The second sequence, reading email, typically involves issuing *openmail*, *firstunreadmessage*, *nextmessage*, and possibly *saveattachments*, and *closemail* statements. For example:

```
openmail
firstunreadmessage
while
{
     ...process message in some way...
     nextmessage
}
closemail
```

The NQL Email Statements

The NQL email statements are listed in Table 12.1. Of these, the six commonly used statements are *openmail*, *firstmessage*, *firstunreadmessage*, *nextmessage*, *sendmessage*, and *closemail*.

The NQL mail statements and functions are described individually in the following sections.

Email Sessions

The NQL email operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

1. Optionally, an email protocol is specified with *mailprotocol*.
2. An email session is opened with *openmail*.

Table 12.1 Email Statements

STATEMENT	PARAMETERS	DESCRIPTION
closemail		Closes the mail session
deletemessage		Deletes the current mail message
firstattachment	[*path*]	Saves the first file attachment for the current message
firstmessage		Reads the first message in the mailbox and sets message variables
firstunreadmessage		Reads the first unread message in the mailbox and sets message variables
forward	*address*	Forwards the current mail message to the specified recipient(s)
logoffmail		Logs off a mail session (MAPI protocol only)
mailidentity	*address*, *name*	Sets the From address and name for sending messages (SMTP protocol only)
mailprotocol	*protocol*	Selects the mail protocol (specify *mapi*, *pop3*, *smtp*, or *internet*)
markmail	*boolean*	Determines whether reading mail messages will mark them read or not
messageattachment	*filespec*	Attaches a file to the current message being prepared for sending
nextattachment		Saves the next file attachment for the current message
nextmessage		Advances to the next message in the mailbox
openmail		Connects to an existing mail session for reading and sending mail (MAPI)
openmail	*user*, *pass*	Connects to a new mail session for reading and sending mail (MAPI)
openmail	*server*	Connects to an Internet mail server for sending mail (SMTP)

continues

Table 12.1 Continued

STATEMENT	PARAMETERS	DESCRIPTION
openmail	*server*, *user*, *pass*	Connects to an Internet mail server for reading mail (POP3)
openmail	*server*, *user*, *pass*, *server*	Connects to Internet mail servers for reading and sending mail (POP3/SMTP)
reply		Sends a reply to the sender of the current mail message
saveattachments	[*path*]	Saves all file attachments for the current message
sendmessage	*subject*, *address*	Sends the value on the stack as a text mail message
sendmessage	*subject*, *address*, *message*	Sends a text mail message
sendmessage	*subject*, *address*, [[*format*, *message*]]	Sends a mail message in the specified format(s)
totalattachments	[*var*]	Returns the number of file attachments for the current message

3. Messages are received or sent as desired. For sending messages, the basic sequence is that message text is constructed in text, HTML, and/or RTF formats and the message is sent with *sendmessage*. For receiving messages, the basic sequence is to begin with *firstmessage* or *firstunreadmessage*, followed by multiple *nextmessage* statements.
4. The email session is closed with *closemail*.

Selecting a Mail Protocol

You may explicitly select a mail protocol in your script; if you fail to do so, NQL will select one automatically. Versions of NQL default to the most natural mail protocol for their target platform. The Windows version of NQL defaults to MAPI. The Java version of NQL defaults to POP3/SMTP.

You select a mail protocol with the *mailprotocol* statement.

```
mailprotocol protocol
```

The *protocol* parameter is a string expression that evaluates to *MAPI*, *POP3*, *SMTP*, or *Internet* (pop3/smtp).

If the selected protocol is *MAPI*, you can connect to an existing MAPI session (such as an instance of Outlook active on your desktop) or log on to a new session, depending on the parameters you give to *openmail*.

If the selected protocol is *POP3*, you'll be able to read messages from an Internet mail server but will not be able to send messages.

If the selected protocol is *SMTP*, you'll be able to send messages to an Internet mail server but will not be able to read messages.

If the selected protocol is *Internet*, you'll be able to send messages to an Internet mail server as well as receive messages.

Connecting to a Mail Server

The *openmail* statement connects to a mail server. Connecting to a mail server is a necessary first step before mail can be sent or received. *Openmail* accepts varying numbers of parameters depending on the mail protocol you are using.

When using MAPI, you have the option of attaching to an existing mail session or creating a new one. The advantage of using an existing mail session is that you do not have to specify account and password information. However, you must have an actual mail client (such as Outlook) running on your desktop or the mail connection will fail. *Openmail* requires no arguments when using MAPI with an existing mail session.

```
openmail
```

If, when using MAPI, you wish to explicitly specify a user name and password for a new mail session, specify the user name and password as parameters to *openmail*.

```
openmail user, password
```

When using POP3, the server name, user name, and password must be specified as parameters.

```
openmail server, user, password
```

When using SMTP, the server name must be specified as a parameter.

```
openmail server
```

When using the Internet protocol (POP3 and SMTP), the POP3 server name, user name, password, and SMTP server name must be specified as parameters.

```
openmail pop3-server, user, password, smtp-server
```

Regardless of how you call it, *openmail* will set a success condition if it is able to successfully make a connection to a mail server. Once you reach this point, you are free to send and receive mail.

Sending Email

Sending a mail message involves the following steps:

1. Compose the mail message, in one or more formats.
2. Add any file attachments with *messageattachment*.
3. Send the message using *sendmessage*.

Mail messages may be in text, RTF, or HTML format. For a simple text-only mail message, place the message text on the stack (*push* or *load* are good for this). For messages that have multiple formats, put the messages into variables and specify them as parameters in the long form of the *sendmessage* statement. The following code line pushes a text message onto the stack.

```
push "This is a test message.\nThe end.\n"
```

The *messageattachment* statement should be called for each file attachment (if any). The parameter is a file specification of the file to be attached.

```
messageattachment filespec
```

The *sendmessage* statement sends the message. There are three forms of the statement, one for sending text messages and one for sending messages in multiple formats. All forms specify a message subject and an address. The address parameter may contain one or more email addresses, separated by commas or semicolons.

```
sendmessage subject, address
sendmessage subject, address, text
sendmessage subject, address, format/message-list
```

The first form of *sendmessage* sends a text email message using the top item on the stack as the message. The following script fragment sends a mail message using this method.

```
push "Please don’t forget to pick up some milk on the way home."
sendmessage "Got Milk?", john@my-work.com
```

The second form of *sendmessage* specifies the text message as a third parameter rather than requiring it to be on the stack. The following script fragment sends a mail message using this method.

```
sendmessage "Got Milk?", "john@my-work.com", "Please don’t forget to
pick up some milk on the way home."
```

The longer third form of *sendmessage* also specifies subject and recipient, but adds additional parameters that specify the format and content of each message. A format parameter (*text*, *RTF*, or *HTML*) and a message parameter (any variable or expression that holds the message in the specified format) make up a pair of parameters. There may be 1, 2, or 3 pairs of format/message parameters. The following script fragment sends a message in both text and HTML formats.

```
MsgText = "Thanks for meeting with us. We'll be in touch shortly."
MsgHTML = "<HTML><BODY>Thanks for meeting with us. We'll be in touch
shortly.</BODY></HTML>"
sendmessage "Meeting follow-up", "duncan@my-prospect.com",
    "text", MsgText, "html", MsgHTML
```

Sendmessage sets a success condition if it has successfully passed the message on to a mail server. You can send messages only if the current mail protocol is MAPI, SMTP, or Internet.

Reading Mail

Reading mail messages involves the following steps:

1. Read the first mail message with *firstmessage* or *firstunreadmessage*.
2. Process the message particulars, which are in variables.
3. Process any file attachments by calling *firstattachment* and *nextattachment*, or by calling *saveattachments*.
4. Call *nextmessage* to advance to the next mail message.
5. Repeat steps 2 through 4 for the next message.

Firstmessage reads the first message in your mailbox. *Firstunreadmessage* is similar, but reads the first new (unread) message in your mailbox. Both statements automatically create variables named MessageDate, MessageFrom, MessageFromAddress, MessageTo, MessageToAddress, MessageSubject, and MessageText. The following script reads all unread mail messages and outputs them as XML.

```
openmail
firstunreadmessage
while
{
    output MessageDate, MessageFrom, MessageSubject, MessageText
    nextmessage
}
closemail
```

Nextmessage moves on to the next message, and takes its cue from whether you used *firstmessage* or *firstunreadmessage* initially. *Firstmessage* followed by repeated *nextmessage* calls will iterate through all messages in your mailbox. *Firstunreadmessage* followed by repeated *nextmessage* calls will iterate through all new messages in your mailbox. The following script sequences through all messages in the mailbox and displays them.

```
openmail
firstmessage
while
{
    show MessageDate " " MessageFrom "\n" MessageSubject "\n"
MessageText
    nextmessage
}
closemail
```

While processing a message, you may also access its file attachments, if any. There are two ways to save file attachments: individually, or in bulk. To save file attachments individually, use the *firstattachment* and *nextattachment* statements. To save all file attachments in a single operation, use the *saveattachments* statement. The following script saves all file attachments for all new messages in the directory c:\Attachments.

```
openmail
firstunreadmessage
while
{
    saveattachments "c:\\Attachments"
    nextmessage
}
closemail
```

You can control whether reading a mail message with *firstmessage/firstunreadmessage/nextmessage* actually marks the mail message as read. The *markmail* statement takes one parameter, a Boolean value: True means messages are marked as read. The default when a script initializes is false.

```
markmail boolean
```

Forwarding Messages

The *forward* statement or function forwards the last mail message read. The parameters are the address to forward the message to and a subject, which is optional. If the subject is omitted, the original subject is used.

```
forward address
boolean forward(address [, subject])
```

The message is sent, and unless an error occurs a success condition results (the function returns true). The following code forwards a mail message with *forward*.

```
openmail
firstunreadmessage
while
{
    if MessageFrom=="$"
    {
        forward "accounting", "FYI"
    }
    nextmessage
}
closemail
```

You can forward messages only if the current mail protocol is MAPI, SMTP, or Internet.

Replying to Messages

The *reply* statement or function replies to the last mail message read. The parameters are an optional subject and a reply message. If the subject is omitted, the subject of the original message is used.

```
reply [subject, ] message
boolean reply([subject,] message)
```

The message is sent, and unless an error occurs a success condition results (the function returns true). The following code replies to a mail message with *reply*.

```
openmail
firstunreadmessage
while
{
    if MessageFrom=="Mom"
    {
        reply "Thanks, Mom."
    }
    nextmessage
}
closemail
```

You can reply to messages only if the current mail protocol is MAPI, SMTP, or Internet.

Disconnecting from a Mail Server

To disconnect from a mail server, issue a *closemail* statement.

```
closemail
```

When a script ends, any unclosed mail sessions are automatically closed at that time.

Tutorial: ScanMail

Now that you are familiar with email processing in NQL, let's put that knowledge to use in a script. In this tutorial, you will create a script that scans new mail messages and automatically acts on some of them. Messages that contain words suggestive of problems will be forwarded to a Technical Support email address. Messages that contain words suggestive of junk mail will be replied to (requesting removal from the mailing list) and deleted.

There are five steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the ScanMail script.
3. Customize the ScanMail script.
4. Run the ScanMail script.
5. Understand the ScanMail script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Connect to a MAPI mail session.
- Scan all unread mail messages.
- Reply to mail messages.
- Forward mail messages.
- Delete mail messages.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the ScanMail Script

In the NQL development environment, enter the script shown in Listing 12.3 and save it under the name ScanMail.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy ScanMail.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial

```
//ScanMail - scans new mail messages and responds to some of them
automatically

mailprotocol "mapi"

if openmail()
{
    boolean bHaveMail = firstunreadmessage()
    while bHaveMail
    {
        //Forward messages containing problem reports to
        //technical support.

        if contains(MessageText, "problem") or
           contains(MessageText, "bug") or
           contains(MessageText, "doesn't work")
        {
            forward "techsupport@my-site.com", "Please look into
this."
            reply "Thank you for your inquiry. Our Tech Support
department will get back to you shortly."
        }

        //Reply to junk mail with a request for removal, and
        //delete the message.

        if contains(MessageText, "free") or
           contains(MessageText, "bargain") or
           contains(MessageText, "special offer")
        {
            reply "Please remove me from your mailing list."
            deletemessage
        }

        bHaveMail = nextmessage()
    }
    closemail
}
else
{
    show "Error: unable to open mailbox"
    end
}
```

Listing 12.3 ScanMail script.

materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select ScanMail.nql from the Tutorials\ch12 folder.

At this point, the script in Listing 12.3 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Customize the ScanMail Script

The script now needs to be customized. Messages containing the text "problem", "bug", or "doesn't work" will be forwarded to a technical support email address. In the script, change the address "techsupport@my-site.com" to an email address you want the messages forwarded to.

At this point, you should have customized the forwarding address to match your environment.

Step 4: Run the ScanMail Script

Now, we are ready to begin. Make sure you have Outlook or some other mail client active on your desktop, then run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script completes, new messages in your inbox will have been scanned and some of them acted upon. To test the script, you may want to mail yourself some messages with deliberate content first, then run the ScanMail script. If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Make sure you have a mail client active on the desktop.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the ScanMail script run, connecting to a mail session and acting on some messages.

Step 5: Understand the ScanMail Script

We now want to make sure we understand every part of the ScanMail script. The first line is a comment line.

```
//ScanMail - scans new mail messages and responds to some of them
automatically
```

First, the mail protocol is declared with *mailprotocol*. This script is written for MAPI mail servers.

```
mailprotocol "mapi"
```

A connection to the current mail session is established with *openmail*. If successful, the remainder of the script executes in the *if* block. If it fails, an error message is displayed and the script ends.

```
if openmail()
{
    ...process mail...
}
else
{
    show "Error: unable to open mailbox"
    end
}
```

The first unread mail message is read with *firstunreadmessage*. If there is at least one unread message, the variable *bHaveMail* is set to true, and the *while* loop that follows executes. The message particulars are automatically stored in message variables (*MessageFrom*, *MessageSubject*, *MessageText*, and so on).

```
boolean bHaveMail = firstunreadmessage()
while bHaveMail
{
```

In the *while* loop, a check is made to see if the message should be forwarded to Technical Support. The *contains* function is used to check the message body (in *MessageText*) for certain keywords. If the message text contains one or more of those keywords, the message is forwarded to Tech Support (with *forward*), and an acknowledgement is also sent to the sender of the message (with *reply*).

```
//Forward messages containing problem reports to
//technical support.

if contains(MessageText, "problem") or
   contains(MessageText, "bug") or
   contains(MessageText, "doesn't work")
{
    forward "techsupport@my-site.com", "Please look into this."
    reply "Thank you for your inquiry. Our Tech Support department will
get back to you shortly."
}
```

The message is also checked to see if it might be a junk mail message. The *contains* function is used again to check the message body (in *MessageText*) for keywords. If the message text contains one or more of those keywords, a request to be removed from the mailing list is sent to the sender of the message (with *reply*). The message is also deleted (with *deletemessage*).

```
//Reply to junk mail with a request for removal, and
//delete the message.

if contains(MessageText, "free") or
   contains(MessageText, "bargain") or
```

```
      contains(MessageText, "special offer")
   {
       reply "Please remove me from your mailing list."
       deletemessage
   }
```

At the bottom of the *while* loop, the *nextmessage* function retrieves the next mail message, setting *bHaveMail* to true if there is another message. The *while* loop continues until all unread mail messages have been scanned.

```
       bHaveMail = nextmessage()
   }
```

Once the *while* loop completes, the mail session is terminated with *closemail*.

```
closemail
```

At this point, you've seen how to scan and act on email in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Scan the sender of each message against a list of recognized addresses, and delete the messages from strangers.
- Scan the messages for attachments, and save the attachments in a directory reserved for virus scanning. If one or more attachments are saved, launch an antivirus program (with the *opendoc* statement).

Chapter Summary

Network Query Language can act as a mail client, sending and receiving email. Supported protocols include MAPI, POP3, and SMTP.

- Mail protocols are declared with the *mailprotocol* statement.
- Mail identities (for SMTP mail) are specified with the *mailidentity* statement.
- Mail sessions are initiated with the *openmail* statement.
- Mail sessions can be logged off with the *logoffmail* statement.
- Mail sessions are terminated with the *closemail* statement.

Mail messages are read in a loop using *first* and *next* statements.

- The *firstmessage* statement reads the first mail message.
- The *firstunreadmessage* statement reads the first new mail message.

- The *nextmessage* statement reads the next mail message.
- The *markmail* statement controls whether or not reading mail messages marks them as read or not.

Once a mail message has been read its fields and message text can be used, and the message can be acted upon.

- The message's data is stored in the variables *MessageDate*, *MessageFrom*, *MessageFromAddress*, *MessageTo*, *MessageToAddress*, *MessageSubject*, and *MessageText*.
- The *deletemessage* statement deletes the last read mail message.
- The *forward* statement forwards the last read mail message to a new address.
- The *reply* statement sends a reply to the last read mail message.

Once a message has been read, its file attachments can be accessed.

- The *firstattachment* statement saves the first file attachment.
- The *nextattachment* statement saves the next file attachment.
- The *totalattachments* statement returns the number of file attachments.
- The *saveattachments* statement saves all of the file attachments.

Mail messages can be sent in one or more formats (text, RTF, and HTML) and can have file attachments.

- The *messageattachment* statement declares a file attachment for the next message to be sent.
- The *sendmessage* statement sends a mail message.

Connected applications frequently make use of email, either as a direct part of their processing or as a means to report errors.

CHAPTER

13

Accessing Newsgroups

Newsgroups, also known as forums or discussion groups, are a huge phenomenon. Millions of people around the world can browse newsgroups and post messages. Newsgroups exist for myriads of dedicated topics, as general as *movies* or as specific as *classic sci-fi movies*. Newsgroups are accessed with the Network News Transfer Protocol (NNTP).

Network Query Language allows your programs to scan newsgroups, which can be useful for monitoring public opinion, complaints, rumors, and interest levels in products. This chapter describes NQL's newsgroup support. The tutorial at the end of the chapter scans a list of newsgroups and captures the email addresses of message posters.

An Overview of NQL's Newsgroup Capabilities

NQL allows your scripts to act as newsgroup clients, connecting to a newsgroup server and reading messages. Specifically, your programs can do the following:

- Connect to a news server and a newsgroup
- List the newsgroups carried by a news server

- Read messages from a newsgroup
- Read message headers from a newsgroup
- Disconnect from a news server

Let's take a quick look at what it is like to work with newsgroups in NQL. The following script is a program for reading all of the postings from a newsgroup dedicated to a discussion of puppies.

```
opennews "news.my-site.com", "alt.pets.puppies"
firstarticle
while
{
    ...do something with article content...
    nextarticle
}
closenews
```

As you can see, working with newsgroups in NQL is easy and straightforward. Let's study this script line-by-line.

- *opennews* connects to a news server.
- *firstarticle* reads the first article in the newsgroup and sets values into variables.
- *nextarticle* reads the next article in the newsgroup.
- *closenews* disconnects from the news server.

Newsgroup Support in NQL

The NQL newsgroup statements are listed in Table 13.1.

There are function editions of the newsgroup statements. The newsgroup functions are listed in Table 13.2.

The newsgroup statements and functions are described individually in the following sections.

Newsgroup Sessions

The NQL newsgroup operations are modeled on the idea of a session in which the statements and functions must be used in a certain order.

1. A newsgroup session is opened with *opennews*.
2. Articles are read by issuing a *firstarticle* statement followed by multiple *nextarticle* statements.
3. The newsgroup session is closed with *closenews*.

Table 13.1 Newsgroup Statements

STATEMENT	PARAMETERS	DESCRIPTION
closenews		Closes the news session
firstarticle		Reads the first article in the current newsgroup
firstarticlehdr		Reads the first article's header in the current newsgroup
listnewsgroups		Lists the newsgroups on the news server
nextarticle		Reads the next article in the current newsgroup
nextarticlehdr		Reads the next article's header in the current newsgroup
opennews	*newsserver*, *newsgroup*	Connects to a news server and opens a newsgroup

Table 13.2 Newsgroup Functions

FUNCTION	DESCRIPTION
bool closenews( )	Closes the news session
bool firstarticle( )	Reads the first article in the current newsgroup
bool firstarticlehdr( )	Reads the first article's header in the current newsgroup
bool listnewsgroups( )	Lists the newsgroups on the news server
bool nextarticle( )	Reads the next article in the current newsgroup
bool nextarticlehdr( )	Reads the next article's header in the current newsgroup
bool opennews(*newsserver*, *newsgroup*)	Connects to a news server and opens a newsgroup

Connecting to a News Server

The *opennews* statement or function connects to a news server. Connecting to a news server is a necessary first step before newsgroup articles can be read. There are two parameters for *opennews*: a news server address, and the name of a newsgroup to open.

```
opennews newsserver, newsgroup
```

If the news server is connected to successfully and the specified newsgroup can be opened, a success condition results. Once you reach this point, you can read articles.

Reading Articles

Reading newsgroup articles involves the following steps:

1. Read the first article with *firstarticle* or *firstarticlehdr*.
2. Process the message particulars, which are in variables.
3. Call *nextmessage* or *nextmessagehdr* to advance to the next article.
4. Repeat steps 2 and 3 for the next message.

The first article is read with *firstarticle*, which takes no parameters.

```
firstarticle
```

The success condition is set if an article was found. The variables ArticleDate, ArticleFrom, ArticleID, ArticleSubject, and ArticleText are set with the date, author, article ID, subject, and message text of the article.

Subsequent articles are read with *nextarticle*, which takes no parameters.

```
nextarticle
```

A success condition results unless there are no more articles in the newsgroup. Like *firstarticle*, *nextarticle* sets the ArticleDate, ArticleFrom, ArticleID, ArticleSubject, and ArticleText variables. The following code shows a sequence for reading all of the articles in a newsgroup using *firstarticle* and *nextarticle*.

```
opennews server, newsgroup
firstarticle
while
{
    ...do something with article...
    nextarticle
}
closenews
```

For some applications, you may be more interested in the article header information (date, author, subject, ID) than you are with the message text itself. In these cases, the *firstarticlehdr* and *nextarticlehdr* statements may be used. They are identical to *firstarticle* and *nextarticle* except that they do not read the message text of articles, saving some time. The following code shows a loop for processing all article headers in a newsgroup.

```
opennews server, newsgroup
firstarticle
while
{
    ...do something with article...
    nextarticle
}
closenews
```

Listing Newsgroups

The *listnewsgroups* statement or function retrieves a list of available newsgroups from the news server.

```
listnewsgroups
boolean listnewsgroups()
```

The newsgroups are pushed as a string onto the stack. Each newsgroup name is separated by a newline. The *nextline* statement or function is useful for iterating through the newsgroup names.

Disconnecting from a News Server

To disconnect from a news server, issue a *closenews* statement.

```
closenews
```

If a script ends, any unterminated news sessions are closed automatically.

Tutorial: Prospects

Let's apply the newsgroup statements in a script. In this tutorial, you will create a script that scans a newsgroup and captures email addresses in a text file. This is to illustrate newsgroup scanning capabilities and not to suggest that you perform spamming, which is looked down upon by most people and disallowed by most ISPs. There are some legitimate applications for scanning newsgroups, however: You may be a newsgroup moderator, or some newsgroups may have participants who agree to receive email from you. In any event, this tutorial does not send any email, it only captures email addresses.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Prospects script.
3. Run the Prospects script.
4. Understand the Prospects script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Connect to a newsgroup on a news server.
- Scan all articles in a newsgroup.

Let’s begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you’re ready to enter the tutorial script.

Step 2: Enter the Prospects Script

In the NQL development environment, enter the script shown in Listing 13.1 and save it under the name prospects.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy prospects.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book’s tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Prospects.nql from the Tutorials\ch13 folder.

```
//Prospects - scan a newsgroup and capture email addresses

if opennews("news.my-site.com", "alt.pets.puppies")
{
    append "email.txt"
    boolean bHaveArticle = firstarticle()
    while
    {
        write ArticleFrom "\r\n"
        bHaveArticle = nextarticle()
    }
    closenews
    close
    opendoc "email.txt"
    end
}
  show "Error: unable to open newsgroup"
end
```

Listing 13.1 Prospects script.

You will need to change the news server name and newsgroup name in the *opennews* statement to reflect a newsgroup you have access to. The names in the listing are dummy names.

At this point, the script in Listing 13.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You also should have replaced the news server and newsgroup names with real names. You are now ready to run the script.

Step 3: Run the Prospects Script

Run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script completes, a text file opens on the desktop showing the captured email addresses from the newsgroup. If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Check that the news server name and newsgroup name are correct.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Prospects script run, connecting to a news server and capturing information about posted articles.

Step 4: Understand the Prospects Script

We now want to make sure we understand every part of the Prospects script. The first line is a comment line.

```
//Prospects - scan a newsgroup and capture email addresses
```

A connection is made to the news server and newsgroup with the *opennews* function. If the newsgroup is opened successfully, the *if* block executes. If not, an error message is displayed and the script ends.

```
if opennews("news.my-site.com", "alt.pets.puppies")
{
    ...process articles...
}
show "Error: unable to open newsgroup"
end
```

Within the *if* block, the file email.txt is opened with *append*. If email.txt already exists, it is opened for appending.

```
append "email.txt"
```

The first article is accessed with the *firstarticle* function. If successful, variables are set and *bHaveArticle* is set to true, causing the *while* loop to be executed. In the *while* loop, the article poster's email address (in *ArticleFrom*) is written to the email.txt file. At the bottom of the *while* loop, a *nextarticle* function advances to the next article and sets *bHaveArticle*. The *while* loop continues as long as there are articles to process.

```
boolean bHaveArticle = firstarticle()
while
{
    write ArticleFrom "\r\n"
    bHaveArticle = nextarticle()
}
```

Once the *while* loop exits, the news server connection is terminated with *closenews*, and the output file is closed with *close*.

```
closenews
close
```

The output file is opened on the desktop with *opendoc*, and the script ends.

```
opendoc "email.txt"
end
```

At this point, you've seen how to scan and capture information from newsgroup articles in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Check the content of each article for keywords of interest to you. Capture only the articles that contain one or more keywords.
- If the newsgroup you are accessing is about a product, see if you can write a script that determines the number of positive articles and the number of negative articles. If you can make such measurements at regular intervals, you have a way of tracking the improvement or lack of improvement of public perception over time.

Chapter Summary

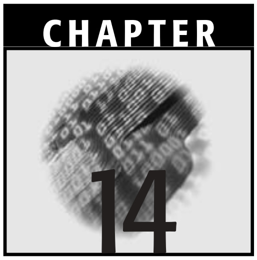
Network Query Language can access newsgroups using the NNTP protocol.

- A connection to a newsgroup on a news server is made with the *opennews* statement.
- The *listnewsgroups* statement lists the newsgroups available on a news server.

- The *firstarticle* and *firstarticlehdr* statements access the first article in a newsgroup.
- The *nextarticle* and *nextarticlehdr* statements access the next article in a newsgroup.
- The *closenews* statement terminates a newsgroup connection.

Newsgroup access is a valuable way to track interest levels, rumors, approval levels, and other measures of public perception.

CHAPTER



14

Searching Directory Servers

Directory servers provide central address books for people, computers, applications, and other resources. They are becoming more and more essential, especially in environments with many elements to be managed, such as a large organization's personnel directory. Lightweight Directory Access Protocol (LDAP) is the chief protocol in use for directory servers.

Network Query Language allows you to connect to and search LDAP directory servers. This chapter describes NQL's LDAP support. The tutorial at the end of the chapter searches for people by name against a public LDAP server.

An Overview of NQL's LDAP Capabilities

NQL allows a script to act as an LDAP client, connecting to an LDAP server and performing searches. Specifically, your programs can do the following:

- Connect to an LDAP server
- Issue a search query
- Process results from a search query
- Disconnect from an LDAP server

Let's take a quick look at what it is like to work with directory servers in NQL. The following script is a program for searching against an LDAP personnel directory.

```
Criteria = "cn=robert apodaca"

opendirectory URL
then
{
    search Criteria
    while
    {
        ...do something with search results...
        nextentry
    }
    closedirectory
}
```

As you can see, working with directory servers in NQL is easy and straightforward. Let's study this script line-by-line.

- *Opendirectory* connects to a directory server.
- *Search* issues a search query, and if successful, loads data from the first result record into variables.
- *Nextentry* advances to the next result record.
- *Closedirectory* disconnects from the directory server.

LDAP Queries

LDAP queries are called search filters. A search filter takes the following form:

```
(attribute operator value)
```

For example, (cn=Alex) means "common name of Alex". Values may use the asterisk for wildcarding, so a search filter like (cn=Al*) would match common names such as Alice, Alex, and Alvin.

The attributes available are specific to the type of directory you are searching. Some commonly used attribute values are listed in Table 14.1.

A search filter can also be a combination of multiple filters using this form

```
(operator(filter1)...(filterN))
```

Thus, a filter to find personnel from two different organizations could be expressed this way:

```
(|(o=acme trucking)(o=pdq shipping))
```

The operators available for filters are listed in Table 14.2.

Table 14.1 Common LDAP Attributes

ATTRIBUTE	DESCRIPTION
c	Country
cn	Common name
dc	Domain component
o	Organization
objectClass	Object class
ou	Organizational unit
sn	Surname

Table 14.2 LDAP Operators

OPERATOR	DESCRIPTION
=	equals
~=	approximately equal to
<=	less than or equal to
>=	greater than or equal to
&	and
|	or
!	not

Some examples of search filters are shown in Table 14.3.

In NQL, the outer parentheses are optional when you specify search filters (queries).

LDAP Directory Support in NQL

The NQL LDAP statements are listed in Table 14.4.

There are function versions of the LDAP statements. Table 14.5 lists the LDAP functions.

The NQL LDAP statements and functions are described individually in the following sections.

LDAP Sessions

The NQL LDAP operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

Table 14.3 Search Filter Examples

FILTER	MEANINGS
(cn=mac*)	All objects with a common name that begins with Mac
(objectCategory=*)	All objects in the directory
(objectCategory=person)	All persons in the directory
(&(objectClass=user)(email=*))	All users who have an email attribute
(&(objectClass=user)(!(cn=jim)))	All users except Jim
(&(objectClass=contact)(|(sn=Smith)(sn=Jones)))	All contacts with a surname of Smith or Jones

Table 14.4 Directory Statements

STATEMENT	PARAMETERS	DESCRIPTION
closedirectory		Closes the directory session
search	[*criteria*]	Issues a search query to the directory server
nextentry		Advances to the next result record from the directory server
opendirectory	*url* [,*user*, *password*]	Connects to a directory server

Table 14.5 Directory Statements

FUNCTION	DESCRIPTION
boolean search([*criteria*])	Issues a search query to the directory server
boolean nextentry()	Advances to the next result record from the directory server
boolean opendirectory(*url* [,*user*, *password*])	Connects to a directory server

1. An LDAP session is opened with *opendirectory*.
2. One or more searches are issued. For each search, the *search* statement is invoked, which returns the first matching entry.
3. Successive matching entries are found by invoking the *nextentry* statement repeatedly.
4. The LDAP session is closed with *closedirectory*.

Connecting to a Directory Server

The *opendirectory* statement or function connects to a directory server. Connecting to a directory server is a necessary first step before search queries can be issued. There are three parameters for *opendirectory*: An LDAP URL (directory server address) and a user name and password. The user name and password are optional; they can be omitted if the LDAP server in question does not require authentication.

```
opendirectory ldap-url [,username, password]
boolean opendirectory(ldap-url [,username, password])
```

If *opendirectory* connects to the directory server, a success condition results (the function returns true). Once you reach this point, you can issue queries.

The LDAP URL looks like a typical URL, beginning with an IP address or logical name, but may also specify additional information that is LDAP-specific. For example, the following LDAP URL indicates an organization name (BigFoot) and a country (us).

```
ldap.bigfoot.com/o=bigfoot,c=us
```

Some additional LDAP URLs are shown here:

```
LDAP://ldapsvr/O=Internet/DC=COM/DC=MSFT/DC=DEV/CN=TopHat
LDAP://ldapsvr/CN=TopHat,DC=DEV,DC=MSFT,DC=COM,O=Internet
LDAP://MyDomain.microsoft.com/CN=TopH,DC=DEV,DC=MSFT,DC=COM,O=Internet
LDAP://CN=TopHat,DC=DEV,DC=MSFT,DC=COM,O=Internet
```

Issuing Search Queries

The *search* statement issues a query to a directory server. There is one required parameter, the criteria for the query, which may optionally be followed by a variable list.

```
search criteria [, var-list]
boolean search(criteria [, var-list])
```

The LDAP query consists of one or more field specifications. If the query is successful and there is at least one result record, a success condition results (the function returns true) and the first record is loaded. If one or more variables were specified, they are set with the values of fields in the result record. For example, if the results have a *cn* (common name) field and one of the variables specified is *cn*, that variable will be set to a value. The following code shows the use of *search* with a variable list.

```
search "cn=smith", cn, sn, o, c, phone, email
```

If no variables are specified, then *all* of the fields in the result record become variables. The following code issues the same query and retrieves all fields.

```
search "cn=smith"
```

Subsequent result records are read with *nextentry*, which takes an optional variable list just as *search* does.

```
nextentry [var-list]
boolean nextentry([var-list])
```

A success condition results (the function returns true) unless there are no more result records. Like *search*, *nextentry* sets variables to field values in the result record. The following code shows a sequence for reading all of the result records from a query using *search* and *nextentry*.

```
opendirectory url
search "cn=smith"
while
{
    ...do something with result variables...
    nextentry
}
closedirectory
```

Disconnecting from a Directory Server

To disconnect from a directory server, issue a *closedirectory* statement.

```
closedirectory
```

If a script ends, any open LDAP connections are closed automatically.

Tutorial: NameSearch

Let's apply the LDAP statements in a script. In this tutorial, you will create a script that searches a public LDAP server for contact information about people. This will be a search by name.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the NameSearch script.
3. Run the NameSearch script.
4. Understand the NameSearch script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Connect to an LDAP server.
- Issue an LDAP search query.
- Iterate through the results of a query.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the NameSearch Script

In the NQL development environment, enter the script shown in Listing 14.1 and save it under the name NameSearch.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy NameSearch.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select NameSearch.nql from the Tutorials\ch14 folder.

At this point, the script in Listing 14.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the NameSearch Script

Run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script runs, it prompts you for a person's name, then searches for entries of that name on the public LDAP server. For each match returned by the server, all of the fields returned are displayed in a message box. If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the NameSearch script run, connecting to an LDAP server and issuing queries.

```
//NameSearch  - connects to an LDAP server and looks up a person by
name.

string Server, Name
int Matches

Server = "ldap.bigfoot.com/o=bigfoot,c=us"
Name = prompt("LDAP Search", "Enter a person’s name to look up",
"George Washington")

if opendirectory(Server)
{
    search "sn={Name}"
    while
    {
        dumpvars
        Matches += 1
        nextentry
    }
    closedirectory
    if Matches==0
    {
        show "No matching names were found"
    }
}
else
{
    show "Error: unable to connect to LDAP server"
}
```

Listing 14.1 NameSearch script.

Step 4: Understand the NameSearch Script

We now want to make sure we understand every part of the NameSearch script. The first line is a comment line.

```
//NameSearch  - connects to an LDAP server and looks up a person by name.
```

Three variables are defined. *Server* will hold the LDAP URL of the server to connect to; *Name* will hold a person’s name, entered by the user; *Matches* will be used to keep track of the number of matches.

```
string Server, Name
int Matches
```

The *Server* variable is set to an LDAP URL (BigFoot's is shown here).

```
Server = "ldap.bigfoot.com/o=bigfoot,c=us"
```

A *prompt* function displays a dialog allowing the user to enter a name, which is stored in the *Name* variable.

```
Name = prompt("LDAP Search", "Enter a person's name to look up", "George
Washington")
```

We are now ready to attempt a connection to the LDAP server. The server name is specified to *opendirectory*, and because we are accessing a public LDAP server there is no need to specify a user ID or password. If the connection is successful, the statements in the *if* block are executed. If the connection is not successful, an error message is displayed and the script ends.

```
if opendirectory(Server)
{
    ...
}
else
{
    show "Error: unable to connect to LDAP server"
}
```

In the *if* block, a search is issued with the *search* statement. In this case, the query is for a name with field name *sn* (surname) and the value entered by the user, which has been held in *Name*.

```
search "sn={Name}"
```

If the search is successful (returning at least one match), a success condition results and the *while* loop that follows executes. A *dumpvars* statement displays all defined variables, which includes the LDAP fields just retrieved. The match count is incremented, and at the bottom of the *while* loop a *nextentry* statement advances the next match. The *while* loop repeats until there are no more LDAP entries.

```
while
{
    dumpvars
    Matches += 1
    nextentry
}
```

The connection to the LDAP server is terminated with *closedirectory*.

```
closedirectory
```

If the number of matches found is zero, a message to that effect is displayed with *show*.

```
if Matches==0
{
    show "No matching names were found"
}
```

At this point, you've seen how to search LDAP servers and retrieve the results.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Reduce the likelihood of large numbers of search results for common names. You could count the number of matches and stop when there are too many.
- Ask for additional search criteria, such as state.

Chapter Summary

Network Query Language supports the ability to connect to LDAP directory servers and search them.

- The *opendirectory* statement connects to an LDAP directory server.
- The *search* statement issues a search query. The fields of the first matching entry are automatically stored in variables.
- The *nextentry* statement advances to the next entry that matches the query.
- The *closedirectory* statement disconnects from an LDAP server.

The ability of NQL to access LDAP allows your scripts to search local and public directories for people, systems, and resources.

CHAPTER

15

Terminal Emulation

Terminal emulation allows you to connect to terminal-based computers over the network, including Unix systems and older legacy systems such as minicomputers and mainframe computers. Although such systems are designed to work with interactive users via terminals, they can also be communicated with programmatically if they are connected to the network and support the TELNET Internet protocol. A terminal emulation session allows you to mimic the actions of an interactive user, issuing commands and receiving responses. Network Query Language supports the TELNET protocol, allowing you to connect and communicate with terminal-based systems. This chapter describes terminal emulation and the NQL terminal emulation statements. The tutorial at the end of the chapter connects to a Unix system, issues a system status command, and captures the response.

The Basics of Terminal Emulation

The common protocol for terminal emulation is called TELNET. The TELNET protocol is popular because it is simple, widespread, and easily implemented on many computer platforms, including older systems. TELNET is not the only terminal emulation protocol; communication with some mainframe computers from the big iron era often requires more extensive measures, including not only specialized software protocols but, in some cases, special hardware as well. For most terminal emulation needs, however, TELNET is sufficient.

TELNET is a simple protocol. Like most TCP/IP protocols, it involves sending requests and receiving responses. In TELNET, the requests are simply input data to send to the remote system that would normally have been entered via a terminal keyboard. Likewise, the response data is the output from the remote system that would normally be displayed on a terminal screen.

When you connect to a remote system via TELNET, the system you connect to spawns a process that mimics a terminal session. The primary difference with a true terminal session is that the input and the output are directed over the network. The standard TCP/IP port number for TELNET connections is port 23, but some environments may require different port numbers. It is also a common practice among network administrators to use non-standard port numbers for security reasons.

Modes

In order to have a successful communication session with a terminal-based system, you will need a good understanding of the system's prompts and command language as well as of the prompts and input required by any application programs you will be running. Terminal-based systems are modal; if the data you send is out of sync with the mode of the remote system, your session will be a failure. When you first connect, most remote systems will be prompting you for a user name or login ID, and soon afterward, a password. After a successful login, you'll probably be at operating system level, where a command may be specified. If you enter a command that starts a program running, you may be prompted for input.

In addition to keeping in sync with the mode of a remote system, timing is also important. For example, some systems require a login and password to be entered within several seconds or the system disconnects.

Terminal Emulation Operations

NQL's terminal emulation statements connect, interact, and disconnect from remote terminal-based systems. Any system that can act as a TELNET server should be accessible by NQL.

Table 15.1 lists the NQL terminal emulation statements.

The full names of these statements contain the word *terminal*, but this may be shortened to *term* in the interest of brevity. Thus, *openterm* may be used in place of *openterminal*. For the remaining sections of this chapter, we'll use the *term* form of the statements.

Table 15.2 lists the terminal emulation functions.

Both the statements and functions accomplish the same tasks, so which should you use? For simple straightforward scripts where you are confident of the availability and command/response pattern of the remote system, the statements are the logical

Table 15.1 Terminal Emulation Statements

STATEMENT	PARAMETERS	DESCRIPTION
clearterminal		Empties the receive buffer
closeterminal		Disconnects from the remote system
getterminal		Captures response data and pushes it onto the stack
getterminal	*var*	Captures response data and stores it in the specified variable
openterminal	[*port*]	Connects to a remote system on the specified port number
sendterminal	*data*	Sends data to the remote system
waitterminal	[*response*,] *seconds*	Waits for response within a time interval
waitterminalquiet	[*response*,] *seconds*	Waits for a period of quiet [after a response is seen]

Table 15.2 Terminal Emulation Functions

FUNCTION	DESCRIPTION
boolean clearterminal()	Empties the receive buffer
boolean closeterminal()	Disconnects from the remote system
boolean getterminal()	Captures response data and pushes it onto the stack
boolean getterminal(*var*)	Captures response data and stores it in the specified variable
boolean openterminal([*port*])	Connects to a remote system on the specified port number
boolean sendterminal(*data*)	Sends data to the remote system
boolean waitterminal([*response*,] *seconds*)	Waits for response within a time interval
boolean waitterminalquiet([*response*,] *seconds*)	Waits for a period of quiet [after a response is seen]

choice. When checking for success, the functions allow you to combine terminal emulation actions with *if* and *while* statements.

The NQL terminal emulation statements and functions are described individually in the following sections.

Terminal Sessions

The NQL terminal emulation operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. A terminal emulation session is opened with *openterminal*.
2. Communications, waiting, and other actions are combined in any order desired, using the other terminal statements.
3. The terminal session is closed with *closeterminal*.

Connecting to a Remote System

The first step in a terminal emulation session is to establish a connection to a remote system. The two pieces of information that are needed are the name of the system and the port number it uses for TELNET. The name of the system may be a DNS name such as *mainframe* or a TCP/IP address such as 103.5.16.3. If the remote system is not on your local network, expect to use a fully qualified Internet DNS name such as *mainframe.my-domain.com*. The port number will either be the standard port number of 23, or some other port number chosen for security reasons.

The TELNET session is established with the *openterm* statement.

```
openterm server [, port]
```

A success condition results if *openterm* successfully connects to the remote server. A failure condition results otherwise.

Sending Input

The *sendterm* statement sends input data to the remote system. Depending on the mode of the remote system, the data you send could be a user ID, a password, a command, or input for a program.

```
sendterm data
```

The input you supply to terminal-based systems may need to end in a carriage return, as line-oriented input is the norm on such systems. You can indicate a carriage return with the escape sequence *r* in your input.

The following script connects to a legacy system and sends this data to the remote system: a login ID, a login password, an operating system command, a program launch command, and input data for the program.

```
openterm "unix"
waitterm "login:"
sendterm "rpalmer\r"
waitterm "password:"
```

```
sendterm "wormwood\r"
sendterm "cd /programs\r"
sendterm "month-end\r"
sendterm "6/1/02\r"
getterm
closeterm
```

To put the preceding script into perspective, we can imagine a full terminal session that takes place as shown. Input is shown in boldface type.

```
Accounting System

login: rpalmer
password: wormwood

$ cd /programs
$ month-end
Month End Posting Program

Please enter the period ending date: 6/1/02

Posting complete.

$
```

Capturing Output

The response from the remote system, normally intended for a display screen, can be captured with the *getterm* statement. *getterm* is frequently combined with *clearterm* in order to capture just the part of the output you wish to store.

```
getterm [var]
```

The data from the remote system is stored in the specified variable. If no variable is specified, the data is stored on the stack.

The following script shows the use of *getterm* to capture a file directory from a remote system.

```
openterm "mini2"
waitterm "User ID:"
sendterm "gmendez\r"
waitterm "Password:"
sendterm "2348*grace\r"
clearterm
sendterm "ls\r"
getterm
closeterm
```

Waiting for a Specific Response

You can wait for a specific response from the remote system with the *waitterm* statement. There are two forms of *waitterm*: The first specifies a time interval in seconds.

```
waitterm seconds
```

If there is any output from the remote system at all—even a single character—*waitterm* returns a success condition and allows the script to proceed. If there is no response within the specified number of seconds, *waitterm* sets a failure condition.

The second form of *waitterm* specifies a response string as well as a time interval.

```
waitterm response, seconds
```

If the specified response is encountered within the time interval, a success condition results and the script proceeds. If the response is not seen within the specified number of seconds, *waitterm* sets a failure condition.

The use of *waitterm* is critical for ensuring that you are in sync with the remote system and that you match its timing.

Waiting for End of Response

Waiting for periods of quiet (no output) can be just as critical as waiting for responses when dealing with terminal-based systems. Imagine you wanted to capture the output from a program on a remote system. How would you know when the output was finished? One method might be to wait for some data that you knew would be at the end of the output, in which case you could use the *waitterm* statement. But what if there were no end-of-output value to rely on? In that case, the only good way to know that the output was finished would be to detect that the remote system had stopped sending output. That's the purpose of the *waittermquiet* statement.

The *waittermquiet* statement waits for an interval of quiet. There are two forms of *waittermquiet*. The first specifies a time interval in seconds.

```
waitterm seconds
```

waitterm waits until after the specified time interval passes without any output from the remote system.

The second form of *waitterm* specifies a response string as well as a time interval.

```
waitterm response, seconds
```

waitterm waits until the specified response has been encountered, then waits until after the specified time interval passes without any output from the remote system.

Clearing Screens

As output is received from the remote system, it accumulates in a receive buffer for your terminal session. The receive buffer is emptied through two means: The *getterm* statement and the *clearterm* statement. Use *getterm* when you want to capture output from the remote system; use *clearterm* when you want to discard output from the remote system.

If the purpose of your terminal emulation session is to capture information from a remote system, you may not want to capture the entire session. For example, the connection and login at the beginning of the session and the logoff and disconnection at the end of the session may be of no interest.

The *clearterm* statement clears away any output from the remote system that is in the receive buffer of the terminal session.

```
clearterm
```

Disconnecting

A terminal emulation session is disconnected through the *closeterm* statement.

```
closeterm
```

Depending on the type of system you are communicating with, it may or may not be necessary to formally log off of the system prior to disconnecting.

Tutorial: Unix

Now we shall see terminal emulation in action. The script below connects to a Unix system, logs in, runs a system status command, and captures the output.

For this tutorial, you will need access to a system running Unix or a Unix-derived operating system, such as Solaris or Linux. The system needs to permit TELNET access. If you don't have local access to such a machine, you may be able find a Unix TELNET server on the Internet (for example, some public libraries provide TELNET access). Whether public or private, you'll need to know the machine's IP address or name, a user ID, and a password that you can use.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Unix script.
3. Run the Unix script.
4. Understand the Unix script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Connect to a TELNET server and log on.
- Wait for prompts and issue commands.

- Capture screens of output.
- Log off and disconnect.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Unix Script

In the NQL development environment, enter the script shown in Listing 15.1 and save it under the name Unix.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Unix.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Unix.nql from the Tutorials\ch15 folder.

You'll need to make some changes to this script. The variables defined at the top of the script specify the TELNET server IP address or name, your user ID, and password. Change all three to what will be needed for the system you plan to access. Also, if your system does not use a # prompt (which is the right prompt for a Solaris system), be sure to change the *waittermquiet* statement accordingly. You might need a dollar sign ($) or some other operating system prompt for your particular system. A good way to find these things out is to connect with an interactive TELNET client and access the system manually.

At this point, the script in Listing 15.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Unix Script

Check that your Unix system is up and on the network, then run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5.

The script performs communication, and after a few seconds a message box displays showing you the results of the *ps* (process status) command that was run on the Unix system, which might resemble the following:

```
//Unix - connects to a Unix system and captures process status
information

string system = "sparky"
string user = "admin"
string pass = "secret*3452"

if !openterm(system)
{
    show "Error: Unable to connect to Unix system"
    end
}

//log on

waitterm "login:"
sendterm "{user}\r"
sleep 1
sendterm "{pass}\r"
waittermquiet "#"

//send a process status command, wait for it to complete, and capture
it

clearterm
sendterm "ps\r"
waittermquiet 1
getterm

//log off

closeterm
show
```

Listing 15.1 Unix script.

```
Last login: Sat Jul 28 10:53:39 from 172.16.200.111
Sun Microsystems Inc.   SunOS 5.8       Generic February 2000
$ PID  TTY      TIME CMD
  2716 pts/4    0:00 sh
$
```

If this does not happen, check the following:

- Check your script for typographical errors.
- Ensure that your script is specifying the correct server name, user ID, and password.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Unix script run, connecting to a terminal-based system and mimicking the actions of an interactive, terminal-based user.

Step 4: Understand the Unix Script

We now want to make sure we understand every part of the Unix script. The first line is a comment line.

```
//Unix - connects to a Unix system and captures process status
information
```

Next, three variables are defined which specify the address of the server and the account information (user ID and password) to be used for identification.

```
string system = "sparky"
string user = "admin"
string pass = "secret*3452"
```

The *openterm* statement connects to the Unix system, specifying the system name defined above. If it fails, an error is displayed and the script ends.

```
if !openterm(system)
{
    show "Error: Unable to connect to Unix system"
    end
}
```

With a successful connection, the script can proceed to identify itself to the system. If you had connected to the server with an interactive TELNET client, you probably would have been greeted with a *login:* prompt. The script now explicitly waits for that text to appear with a *waitterm* statement. Once that response is seen, the script responds by sending the user ID and the return key (*r*) with *sendterm*. The password and the return key are sent in the same way a second later.

```
  //log on

waitterm "login:"
  sendterm "{user}\r"
  sleep 1
  sendterm "{pass}\r"
```

Before proceeding, we want to be sure that the login has been accepted and the system is ready for our command. The *waittermquiet* statement accomplished this, waiting until the operating system prompt (# in our script for Solaris) has been seen along with a cessation of output.

```
waittermquiet "#"
```

With a successful login, we can issue a command to the system. First, we dispense with any previous data received, because we are interested only in capturing what comes next. The *clearterm* statement clears our input receive buffer.

The command we wish to issue, *ps*, will report the status of system processes. The *sendterm* statement issues *ps* along with the return key (\r). The script then waits for a second of quiet with *waittermquiet* so that it knows when the command has finished.

```
//send a process status command, wait for it to complete, and capture it

clearterm
sendterm "ps\r"
waittermquiet 1
```

The output is captured with a *getterm* statement, which pushes the received data onto the stack.

```
getterm
```

To logoff and disconnect from the system, a *closeterm* suffices. This terminates the TELNET connection to the Unix system. Note that on some other operating systems, it might be necessary to explicitly log off first.

```
//log off

closeterm
```

Lastly, we want to see the data that was captured. The *show* statement pops the data off of the stack and displays it on-screen.

```
show
```

At this point, you've seen an NQL script connect, log on, and send commands to a Unix system, and capture the response.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Run programs that are more complex than *ps* and learn to interact with them in your scripts.
- Instead of displaying the captured information, save it to a file, spreadsheet, or database using the NQL statements described in other chapters of this book.
- Have the script analyze the information it receives from the Unix system, perhaps alerting you if a problem situation is detected.

Chapter Summary

Terminal emulation allows you to connect to terminal-based computers, such as Unix systems, minicomputers, and mainframe computers. TELNET is the primary terminal emulation protocol.

Network Query Language supports terminal emulation through the TELNET protocol.

- TELNET sessions are opened with the *openterm* statement.
- Commands and data are sent to a TELNET server with the *sendterm* statement.
- Output from the TELNET server is captured with the *getterm* statement. The receive buffer can be cleared with *clearterm*.
- Responses and periods of quiet (no output) can be waited for with the *waitterm* and *waittermquiet* statements.
- TELNET sessions are terminated with the *closeterm* statement.

NQL-driven terminal emulation provides you with an automated way to make older systems full-fledged members of your network.

CHAPTER 16

Interacting with Applications

Interacting with other applications allows programs to interoperate with the local network to a high degree. Technologies such as OLE Automation are intended to allow applications to work together well. Network Query Language includes general statements for interacting with other applications, and also has specific support for generating Microsoft Office documents, spreadsheets, and presentations. This chapter describes the NQL statements for exporting and interacting with applications. The tutorial at the end of the chapter reads mail messages and contacts from Microsoft Outlook using OLE Automation.

Exporting to Microsoft Office

Due to its dominance on the desktop, Microsoft Office is a frequent destination for exporting. Network Query Language has direct support for exporting data to Office documents, spreadsheets, and presentations. The NQL export statements are listed in Table 16.1. These statements are available only in the Microsoft Windows edition of NQL and require the presence of Microsoft Office.

Table 16.2 lists the export functions.

The export statements and functions are described individually in the following sections.

Table 16.1 Export Statements

STATEMENT	PARAMETERS	DESCRIPTION
closeexport		Closes an export session
createexport	*filespec*	Opens Office to create a new document, spreadsheet, or presentation
export	*OutputFile*, *InputFile*, *Type*	Creates a document, spreadsheet, or presentation from a text, table, or picture file
exportgotoline	*line*	Moves to a line in the current page of the current document
exportgotopage	*page*	Moves to a page in the current document
exportnewpage		Generates a new page, sheet, or slide in the current document
exportnextline		Moves to the next line in the current document
exportnextpage		Moves to the next page in the current document
insertpicture	*filespec*	Inserts an image into the current document
inserttable	*filespec*	Inserts a table into the current document
inserttext	*filespec*	Inserts text into the current document
openexport	*filespec*	Opens an existing document, spreadsheet, or presentation
setexportcolor	*color*	Sets the color for text insertion
setexportformat	*font*, *size*, *bold*, *italic*	Sets the font for text insertion

One-line Exporting

The *export* statement creates a document, spreadsheet, or presentation in a single statement. All of the other export statements are used together, where a document is created or opened, built up in stages, and closed. The *export* statement is a one-line alternative for simple exporting needs where one item of text, a table, or an image will be exported. The parameters are an output file, an input file, and a type.

```
export OutputFile, InputFile, Type
boolean export(OutputFile, InputFile, Type)
```

The output file is a file specification that must be of type *.doc*, *.xls*, or *.ppt*. The input file contains text, table data, or an image to export. The type parameter is *text*, *table*, or *picture*.

Table 16.2 Export Functions

FUNCTION	DESCRIPTION
boolean createexport(*filespec*)	Opens Office to create a new document, spreadsheet, or presentation
boolean export(*OutputFile*, *InputFile*, *Type*)	Creates a document, spreadsheet, or presentation from a text, table, or picture file
boolean exportgotoline(*line*)	Moves to a line in the current page of the current document
boolean exportgotopage(*page*)	Moves to a page in the current document
boolean exportnewpage()	Generates a new page, sheet, or slide in the current document
boolean exportnextline()	Moves to the next line in the current document
boolean exportnextpage()	Moves to the next page in the current document
boolean insertpicture(*filespec*)	Inserts an image into the current document
boolean inserttable(*filespec*)	Inserts a table into the current document
boolean inserttext(*filespec*)	Inserts text into the current document
boolean openexport(*filespec*)	Opens an existing document, spreadsheet, or presentation
boolean setexportcolor(*color*)	Sets the color for text insertion
boolean setexportformat(*font*, *size*, *bold*, *italic*)	Sets the font for text insertion

If the type is text, the input file is a text file whose contents are inserted into the document, spreadsheet, or presentation being created.

If the type is *table*, the input file is a text data file containing table data, where each line contains a record of field values separated by commas. The very first line contains the number of columns, a comma, and the number of rows. A sample table data file is shown.

```
2,7
Biro Schreiber, Screenwriter
May Estro, Director
Manny Facture, Producer
Youse Full, Best Boy
Cam Mann, Key Grip
Mark Cissus, Lead Actor
Holly Wood, Lead Actress
```

If the type is "*picture*", the file is an image file, such as a .bmp or .jpg file.

If *export* is able to successfully create the document, spreadsheet, or presentation file and import the specified data, a success condition results. For more ambitious exporting, such as when you have a mixture of text, tables, and pictures, the discrete export statements that follow should be used instead of *export*.

Export Sessions

The NQL transaction operations are modeled on the idea of a session, where the statements and functions must be used in a certain order.

1. An export session is opened with *createexport* or *openexport*.
2. One or more items of data are imported, using *inserttext*, *inserttable*, and *insertpicture*.
3. The export session is closed with *closeexport*.

Creating a New Office File

The *createexport* statement or function creates a document, spreadsheet, or presentation file for export. Creating or opening an export file is a necessary first step in the export process. The parameter is a file specification, which must be of type *.doc*, *.xls*, or *.ppt*.

```
createexport filespec
boolean createexport(filespec)
```

If the appropriate Office application can be launched and the specified file is created, a success condition results and the other export statements may be used to insert data into the file. The following code uses *createexport* to create a PowerPoint presentation.

```
if createexport("new.ppt")
{
    inserttext "new.txt"
    closeexport
}
```

To open an existing file instead of creating a new one, use the *openexport* statement instead.

Opening an Existing Office File

The *openexport* statement or function opens an existing document, spreadsheet, or presentation file for modification. Creating or opening an export file is a necessary first step in the export process. The parameter is a file specification, which must be of type *.doc*, *.xls*, or *.ppt*.

```
openexport filespec
boolean openexport(filespec)
```

If the appropriate Office application can be launched and the specified file is opened, a success condition results and the other export statements may be used to insert data into the file. The following code uses *openexport* to open a Microsoft Word document.

```
if openexport("letter.ppt")
{
    gotopage 2
    inserttext "update.txt"
    closeexport
}
```

To create a new file instead of opening an existing one, use the *createexport* statement instead.

Inserting Text

To insert text into an Office file, the text must be stored in a text file. The *inserttext* statement or function inserts the text file into the Office file. The parameter is the file specification of the text file.

```
inserttext filespec
boolean inserttext(filespec)
```

If the specified file exists and Office imports it without error, a success condition results. The following code creates a PowerPoint presentation and imports text into the first slide.

```
createxport "summary.ppt"
inserttext "summary.txt"
closeexport
```

You can combine multiple text, image, and table insertions to create compound documents.

Inserting Tables

To insert a table into an Office file, the table data must be stored in a text data file. The *inserttable* statement or function inserts the table data into the Office file. The parameter is the file specification of the table data file.

```
inserttable filespec
boolean inserttable(filespec)
```

If the specified file exists and Office imports it without error, a success condition results.

The input file is a text data file containing table data, where each line contains a record of field values separated by commas. The very first line contains the number of columns, a comma, and the number of rows. A sample table data file is shown.

```
2,7
Biro Schreiber, Screenwriter
May Estro, Director
Manny Facture, Producer
Youse Full, Best Boy
Cam Mann, Key Grip
Mark Cissus, Lead Actor
Holly Wood, Lead Actress
```

The following code creates an Excel worksheet and then imports table data into it.

```
createexport "cast.xls"
inserttable "cast.txt"
closeexport
```

You can combine multiple text, image, and table insertions to create compound documents.

Inserting Pictures

To insert an image into an Office file, you must start with an image file. The *insertpicture* statement or function inserts the image file into the Office file. The parameter is the file specification of the image file.

```
insertpicture filespec
boolean insertpicture(filespec)
```

If the specified file exists and Office imports it without error, a success condition results. The following code inserts a picture into a Word document.

```
createexport "poster.doc"
insertpicture "poster.bmp"
closeexport
```

You can combine multiple text, image, and table insertions to create compound documents.

Interacting with Applications Using Automation

NQL has statements for interaction with Windows applications through Automation (also known as OLE Automation, or COM). The NQL Automation statements are

Table 16.3 Automation Statements

STATEMENT	PARAMETERS	DESCRIPTION
closeobject		Closes an Automation object
do	*command*, *parameter-list*	Executes an Automation command
obtain	*command*, *parameter-list*	Retrieves an Automation value
openobject	*program-ID* [, *server*]	Opens an Automation object
set	*command*, *value*	Sets an Automation value

Table 16.4 Automation Functions

FUNCTION	DESCRIPTION
boolean do(*command*, *parameter-list*)	Executes an Automation command
string obtain(*command*, *parameter-list*)	Retrieves an Automation value
boolean openobject(*program-ID* [, *server*])	Opens an Automation object
boolean set(*command*, *value*)	Sets an Automation value

listed in Table 16.3. These statements are available only in the Microsoft Windows edition of NQL.

Table 16.4 lists the Automation functions.

The Automation statements and functions are described individually in the following sections.

Automation Object Sessions

The NQL Automation operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. An Automation object session is opened with *openobject*.
2. Multiple commands, setting values, and retrieving values are performed as desired using *do*, *set*, and *obtain*.
3. The Automation object session is closed with *closeobject*.

Opening an Automation Object

The *openobject* statement (or function) opens an Automation object. The required parameter is a *program-ID* that identifies the application to be opened. An optional second parameter specifies a server, which allows the application to be launched on a remote computer.

```
openobject     program-ID [, server]
boolean openobject(program-ID [, server])
```

The program ID must match the intended application's Automation registry name, such as "Word.Application". The application launches on the same machine that the NQL script is running on, unless the second server parameter is specified, in which case it launches on the specified machine.

If the application launches successfully, a success condition results (the function returns true). The following code uses *openobject* to open Microsoft Word on the local machine.

```
openobject "word.application"
```

Once an object has been opened, the *do*, *obtain*, and *set* statements and functions may be used to interact with the application.

Issuing Commands

The *do* statement (or function) issues a command to an Automation object. The first parameter is a command, which should be an Automation command that the application recognizes. The valid commands vary from one application to the next, so you must be familiar with the commands your target application supports. The command may be a method name, such as "Quit", but the command may also be preceded by one or more interface names separated by periods, such as "Documents.Add".

If the command requires values, they are specified as additional parameters.

```
do command [, parameter-list]
boolean do(command, parameter-list)
```

If the command executes successfully, the statement sets the success condition and the function returns true. The following code launches Microsoft Word, creates a new document, inserts a text file, and saves the result to disk.

```
openobject "word.application"
do "Documents.Add"
do "Selection.InsertFile", "c:\\My Documents\\test1.txt"
do "ActiveDocument.SaveAs", "c:\\My Documents\\test1.doc"
do "ActiveDocument.Close"
do "Quit"
closeobject
```

Some Automation commands return values that you may be interested in capturing. In such instances, use the *obtain* statement in place of *do*.

Retrieving Values

The *obtain* statement (or function) retrieves a value from an Automation object. This is done by issuing a command that returns a value. The first parameter is a command, which should be an Automation command that the application recognizes. The valid commands vary from one application to the next, so you must be familiar with the commands your target application supports. The command may be a method name, such as “Quit”, but the command may also be preceded by one or more interface names separated by periods, such as “Documents.Add”.

If the command requires values, they are specified as additional parameters.

```
obtain command [, parameter-list]
boolean obtain(command, parameter-list)
```

If the command executes successfully, a success condition results and the statement pushes the requested value on the stack. The function version of *obtain* returns the value. The following code launches Microsoft Word, creates a new document, inserts a text file, saves the result to disk, and uses the *obtain* function to retrieve the full file specification of the created document.

```
openobject "word.application"
do "Documents.Add"
do "Selection.InsertFile", "test1.txt"
do "ActiveDocument.SaveAs", "test1.doc"
sFilespec = obtain("Selection.Document.FullName")
do "ActiveDocument.Close"
do "Quit"
closeobject
opendoc sFilespec
```

To set values rather than retrieve them, use the *set* statement. To issue Automation commands without regard to return values, use the *do* statement in place of *obtain*.

Setting Values

The *set* statement (or function) sets a value from an Automation object. This is done by issuing a command and one or more values. The first parameter is a command, which should be an Automation command that the application recognizes. The valid commands vary from one application to the next, so you must be familiar with the commands your target application supports. The command may be a method name, such as “Quit”, but the command may also be preceded by one or more interface names separated by periods, such as “Selection.InsertFile”.

If the command requires values, they are specified as additional parameters.

```
set command [, parameter-list]
boolean set(command, parameter-list)
```

If the command executes successfully, a success condition results. The function version of *set* returns true if successful. The following code launches Microsoft Excel and creates a new spreadsheet, using *set* to set values in spreadsheet cells.

```
openobject "excel.application"
do "workbooks.add"
set "activeworkbook.ActiveSheet.Cells(1,1).Value", "Total"
set "activeworkbook.ActiveSheet.Cells(1,2).Value", nSubtotal + nTax +
nFreight
do "ActiveWorkbook.SaveAs", "test1.xls"
do "ActiveWorkbook.Close"
do "Quit"
closeobject
```

To retrieve values rather than setting them, use the *obtain* statement.

Closing an Automation Object

The *closeobject* statement closes an Automation object. There are no parameters.

```
closeobject
```

If you fail to close an Automation object, it is closed automatically when your script ends.

Tutorial: Outlook

The tutorial that follows uses OLE Automation to interoperate with Microsoft Outlook. The script will obtain information from the Contacts and Inbox folders.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Outlook script.
3. Run the Outlook script.
4. Understand the Outlook script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Open a session to an Automation application.
- Request information from an Automation application.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Outlook Script

In the NQL development environment, enter the script shown in Listing 16.1 and save it under the name outlook.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//Outlook.nql - retrieves Outlook Inbox and Contacts information

errors ignore

//Open the Outlook object

if !openobject("outlook.application")
{
    show 'openobject failed'
    end
}

//Loop though the Outlook Inbox

sOutlookInboxObject = 'GetNameSpace("MAPI").GetDefaultFolder(6)'
nTotal = 0
nTotal = obtain(sOutlookInboxObject & '.Items.Count')

if nTotal == 0
{
    show 'There are no messages in your Outlook Inbox folder'
}
else
{
    if ask('Inbox', 'Do you want to loop through the ' & nTotal &
```

continues

Listing 16.1 Outlook script.

```
        ' messages in your Outlook Inbox folder?')
    {

        for nIndex = 1, nTotal
        {
            sSubject = ''
            sSubject = obtain sOutlookInboxObject & '.Items(' nIndex
').Subject'
            show 'Message Subject: ' & sSubject
        }
    }
}

//Loop though the Outlook contacts

sOutlookContactObject = 'GetNameSpace("MAPI").GetDefaultFolder(10)'
nTotalContacts = 0
nTotalContacts = obtain(sOutlookContactObject & '.Items.Count')
if nTotalContacts == 0
{
    show 'There are no contacts in your Outlook Contacts folder'
}
else
{
    if ask('Contacts', 'Do you want to loop through the ' &
        nTotalContacts & ' entries in your Outlook Contacts folder?')
    {
        for nIndex = 1, nTotalContacts
        {
            sFullName = ''
            sEmail = ''
            sFullName = obtain(sOutlookContactObject & '.Items(' &
                nIndex & ').FullName')
            sEmail = obtain(sOutlookContactObject & '.Items(' &
                nIndex & ').Email1Address')
            if sFullName != '' OR sEmail != ''
            {
                show 'Fullname: ' & sFullName & '\nEmail: ' & sEmail
            }
        }
    }
}

closeobject
```

Listing 16.1 Outlook script (continued).

If you prefer, you may copy outlook.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select outlook.nql from the Tutorials\ch16 folder.

At this point, the script in Listing 16.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Outlook Script

Make sure you have Outlook running on your desktop, then go ahead and run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5.

A dialog appears telling you the number of messages in your Inbox folder and asking if you want them displayed. If you select Yes, the subject of each mail message is displayed in turn (you might *not* want to do this if you have thousands of mail messages). The script then does the same thing for the Contacts folder, displaying the number of contacts and asking if you want them displayed. If this does not happen, check the following:

- Check your script for typographical errors.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Outlook script run, interacting with Microsoft Outlook and retrieving folder data. Although we haven't discussed how it works yet, you've seen the results of an NQL script first-hand, in this case exercising NQL's abilities to apply OLE Automation to interoperate with an application. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Outlook Script

We now want to make sure we understand every part of the Outlook script. The first line is a comment line.

```
//Outlook.nql - retrieves Outlook Inbox and Contacts information
```

The error handling is set to *errors ignore*. The reason for this will become apparent later in the script when we extract contact information.

```
errors ignore
```

The script opens an Automation connection to Outlook through an *openobject* call. The parameter specifies the application name, which happens to be outlook.application for Microsoft Outlook. If *openobject* fails, the script displays an error message and ends.

```
//Open the Outlook object

if !openobject("outlook.application")
{
    show 'openobject failed'
    end
}
```

Now that communication with Outlook has been successfully established, commands and requests can be issued to Outlook. The specific commands that may be issued are dependent on the application you are talking to. In the case of Outlook, the script issues the command to count the number of mail messages in the Inbox folder using the *obtain* function. In Outlook, the Inbox is referenced as folder number 6 of the GetDefaultFolder method.

```
//Loop though the Outlook Inbox

sOutlookInboxObject = 'GetNameSpace("MAPI").GetDefaultFolder(6)'
nTotal = 0
nTotal = obtain(sOutlookInboxObject & '.Items.Count')
```

If the message count is zero, a message to that effect is displayed.

```
if nTotal == 0
{
    show 'There are no messages in your Outlook Inbox folder'
}
```

If there are messages, the script asks the user if they want to display the messages (via the *ask* function). If the response from the user is yes, the *if* block is executed. A *for* loop iterates through the messages, again with *obtain*.

```
else
{
    if ask('Inbox', 'Do you want to loop through the ' & nTotal &
        ' messages in your Outlook Inbox folder?')
    {
        for nIndex = 1, nTotal
        {
            sSubject = ''
            sSubject = obtain sOutlookInboxObject & '.Items(' &
                nIndex ').Subject'
            show 'Message Subject: ' & sSubject
        }
    }
}
```

Now the script performs the same maneuvers, but this time for the Contacts folder. The count of contacts is obtained with *obtain*. In Outlook, the Contacts folder is referenced as folder number 10 of the GetDefaultFolder method.

```
//Loop though the Outlook contacts

sOutlookContactObject = 'GetNameSpace("MAPI").GetDefaultFolder(10)'
```

```
nTotalContacts = 0
nTotalContacts = obtain(sOutlookContactObject & '.Items.Count')
```

If the contact count is zero, a message to that effect is displayed.

```
if nTotalContacts == 0
{
    show 'There are no contacts in your Outlook Contacts folder'
}
```

If there are contacts, the script asks the user if they want to display the contacts (via the *ask* function). If the response from the user is yes, the *if* block is executed. A *for* loop iterates through the contacts, again with *obtain*.

```
else
{
    if ask('Contacts', 'Do you want to loop through the ' &
        nTotalContacts & ' entries in your Outlook Contacts folder?')
    {
        for nIndex = 1, nTotalContacts
        {
            sFullName = ''
            sEmail = ''
            sFullName = obtain(sOutlookContactObject & '.Items(' &
                nIndex & ').FullName')
            sEmail = obtain(sOutlookContactObject & '.Items(' &
                nIndex & ').Email1Address')
            if sFullName != '' OR sEmail != ''
            {
                show 'Fullname: ' & sFullName & '\nEmail: ' & sEmail
            }
        }
    }
}
```

We are now finished talking to Outlook, so the Automation object can be closed with *closeobject*.

```
closeobject
```

At this point, you've seen an NQL script control an application through Automation.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Retrieve more fields of information about messages and contacts. You'll need access to the documentation on Outlook's Automation commands in order to do this.

- Instead of displaying the information, save it to a disk file or export it in some other way.
- Instead of returning all messages or all contacts, prompt the user for a keyword and return only the messages and contacts that contain the keyword.

Chapter Summary

Network Query Language has features for exporting and for interacting with applications.

The export statement and functions export text, tables, and pictures to Microsoft Office applications (Word, Excel, and PowerPoint).

- The *export* statement creates a document, spreadsheet, or presentation in a single statement.
- New office documents are created with the *createexport* statement.
- Existing office documents are opened for modification with the *openexport* statement.
- Text is inserted with the *inserttext* statement.
- Tables are inserted with the *inserttable* statement.
- Pictures are inserted with the *insertpicture* statement.

Applications can be controlled programmatically through Automation. The applications controlled can be on the same machine or a remote system.

- Automation sessions are opened with *openobject*.
- The *do* statement issues Automation commands.
- The *obtain* statement retrieves Automation property values.
- The *set* statement sets Automation property values.
- Automation sessions are closed with *closeobject*.

Exporting and automation give your scripts the power to pass data to applications and initiate processing.

CHAPTER 17

Socket Communications

Internet protocols such as HTTP, NNTP, POP3, and SMTP are all based on socket communications, where datagrams are sent and received using TCP/IP, the underlying protocol of the Internet. Network Query Language not only supports popular Internet protocols but also provides facilities for direct socket communications. This capability allows you to implement TCP/IP protocols not native to NQL and can also be used to create a custom protocol of your own. This chapter introduces the basics of socket communications, then covers socket programming in NQL. The tutorial at the end of the chapter implements a simple custom protocol that demonstrates the server and client ends of socket communication.

Background on Socket Communications

Socket communications is the basis of TCP/IP, the underlying protocol of the Internet and intranets (corporate networks). A socket is an endpoint of communication between two machines. One machine initiates communications and is known as a client. The other responds to clients and is known as a server. Servers are designed to service requests from many incoming clients. Figure 17.1 illustrates the basic client/server model.

Both IP addresses and TCP port numbers figure prominently in socket communications. An IP address is a numbered locator of the form n1.n2.n3.n4, such as 176.12.10.73.

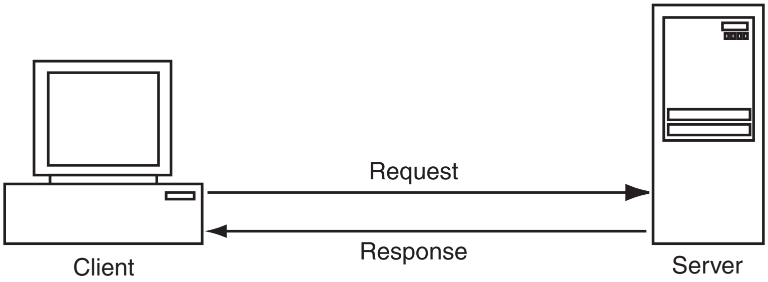


Figure 17.1 TCP/IP client/server model.

It identifies a system's location on a network in the same way that a street address identifies the location of a house. Logical names can be mapped to IP addresses, allowing users to work with logical names as www.my-web-site.com rather than numbers.

Communications specify both IP addresses and an additional number called a TCP port number. Port numbers are kind of like television channel numbers: Different things are happening on different port numbers. Certain port numbers have special meanings associated with them and are referred to as the *well-known port numbers*. To appreciate the importance of port numbers, consider the following three activities:

- Browsing a Web site
- Transferring files
- Sending an email message

The Web browsing takes place on port 80, which both the client and server machines associate with Web protocols. Likewise, port 21 indicates the File Transfer Protocol, and email protocols use yet other port numbers. The use of standardized port numbers allows all three activities to be handled without confusion, even if all three events are being serviced by the same server simultaneously.

To be more specific about port numbers, the well-known port numbers are in the range 0 through 1023. When you want to make a socket connection, then, you need to have an address (named or numbered) as well as a port number. The address is determined by the machine you are interested in connecting to. The port number is determined by the application you have in mind. If you are creating your own custom protocol for communication, you are free to come up with your own port number—but it's wise to avoid using a well-known port number.

Clients

The client is the machine initiating a socket communication. The client must connect to a server with an agreed-upon port number.

A client sends a byte stream of information to the server, known as a *request*. The contents of the byte stream depend on the *protocol* (agreed-upon language) for the application in question. For example, in the FTP protocol commands (requests) are sent using simple text commands such as GET and STOR.

In response to a request, a byte stream known as a *response* is returned. Like the request, the contents of the response are in the appropriate protocol. Some protocols allow for a great diversity of responses; for example, HTTP, the protocol of the World Wide Web, allows servers to respond with HTML, text, XML, and many other formats including multimedia content.

An application follows the steps below to act as a client:

1. A socket connection is opened, specifying a port number indicative of the intended protocol.
2. A connection is made to a server, specifying the server's IP address.
3. For each communication desired, a request is sent to the server and a response is received back from the server.
4. When the client is finished communicating, the socket connection is closed.

Servers

The server is the machine responding to a socket communication. The server's port number is standardized and implies a protocol. The client and server must agree on the port number.

The server enters a listening mode, awaiting an incoming connection from a client. Once a client connection occurs, the incoming request (a byte stream) is received. The software on the server analyzes the response and decides what to do about it. A response (a byte stream) is generated and sent back to the client.

An application follows these steps to act as a server:

1. A socket connection is created, specifying a port number indicative of the intended protocol.
2. The server enters listening mode, awaiting an incoming connection.
3. When an incoming connection occurs, the incoming request is received.
4. A response is generated based on the request and sent back to the client.
5. The server repeats steps 2 through 4 repeatedly.

Socket Programming in NQL

NQL has statements that allow scripts to act as socket clients or socket servers. The NQL socket statements are listed in Table 17.1.

There are function versions of the socket statements. Table 17.2 lists the socket functions.

The socket statements and functions are described individually in the following sections.

Table 17.1 Socket Statements

STATEMENT	PARAMETERS	DESCRIPTION
closesocket		Closes a socket
connect	*url*	Connects to a server
listen		Listens for an incoming connection
opensocket	*port*	Opens a socket
receive	[*var*]	Receives data from a socket
send	*value*	Sends data to a socket

Table 17.2 Socket Functions

FUNCTION	DESCRIPTION
bool connect(*url*)	Connects to a server
bool opensocket(*port*)	Opens a socket
bool receive(*var*)	Receives data from a socket
bool send(*value*)	Sends data to a socket

Socket Sessions

The NQL socket operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. A socket session is opened with *opensocket*.
2. If a server, a connection is waited for with *listen*. If a client, a connection is made using *connect*.
3. Multiple communications may now take place between client and server, always initiated by the client. If a client, information is transmitted with *send* and the response is captured with *receive*. If a server, requests are captured with *receive* and responses are generated with *send*.
4. The socket session is closed with *closesocket*. Generally, clients issue *closesocket*.

Opening a Socket

The *opensocket* statement or function opens a TCP/IP socket. The parameter is a port number.

```
opensocket     port
bool opensocket(port)
```

If the socket is successfully opened, a success condition results (the function returns true). The following code opens a socket on port 80, the standard port for HTTP (World Wide Web) communications.

```
opensocket 80
```

Once a socket has been successfully opened, a server may use the *listen*, *receive*, and *send* statements and functions. A client may use the *connect*, *send*, and *receive* statements and functions.

Listening for an Incoming Connection

Once a socket has been opened, a server needs to listen for an incoming connection. The *listen* statement waits until an incoming connection occurs from a client. There are no parameters.

```
listen
```

Once a connection is made from a client, script execution continues at the next statement. The following code opens a socket, listens for an incoming connection, receives the request from the client, and sends a hard-coded response.

```
opensocket 123
listen
receive
send "I see you!"
closesocket
```

The reciprocal of *listen* is the *connect* statement, which a client issues. The server should open a socket and go into listen mode as early as possible in order to be able to handle as many incoming connections as possible.

Making a Connection to a Server

Once a socket has been opened, a client needs to connect to a server. The *connect* statement or function attempts a connection. The parameter is the URL of the server.

```
connect(url)
bool connect(url)
```

If the connection is established, a success condition results (the function returns true). The following code opens a socket and makes a connection to a server.

```
opensocket 123
connect "server.my-net.com"
```

```
send "I see you!"
closesocket
```

The counterpart to *connect* is the *listen* statement, which a server issues.

Sending Data

A client or a server sends data with the *send* statement or function. The parameter is a value to transmit.

```
send value
bool send(value)
```

If the data is transmitted without error, a success condition results (the function returns true). Clients always send data first and then receive responses. Servers do the reverse, accepting requests first and then sending responses. The following client code opens a socket, makes a connection to a server, and sends several requests to the server using *send*.

```
opensocket 123
connect "server.my-net.com"
send "command1"
receive sResponse1
send "command2"
receive sResponse2
closesocket
```

The counterpart to *send* is the *receive* statement, which captures data.

Receiving Data

A client or a server receives data with the *receive* statement or function. The parameter is a variable to receive the data. The parameter is optional in the statement form of *receive*, and if omitted the response is pushed onto the stack.

```
receive [var]
bool receive(var)
```

If the data is received without error, a success condition results (the function returns true). Clients always send data first and then receive responses. Servers do the reverse, receiving requests first and then sending responses. The following server code opens a socket, listens for an incoming connection, and receives responses using *receive*.

```
opensocket 123
listen
receive sRequest
```

```
send "got it!"
closesocket
```

The counterpart to *receive* is the *send* statement, which transmits data.

Closing a Socket

The *closesocket* statement closes a TCP/IP socket. There are no parameters.

```
closesocket
```

If the socket is successfully opened, a success condition results (the function returns true). The following code opens a socket on port 80, the standard port for HTTP (World Wide Web) communications.

```
opensocket 80
```

Once a socket has been successfully opened, a server may use the *listen*, *receive*, and *send* statements and functions. A client may use the *connect*, *send*, and *receive* statements and functions.

Tutorial: Client/Server

Now it's time to see socket connections work in NQL. In this tutorial, you will create two scripts that will implement a custom TCP/IP protocol, one acting as a client and the other as a server. The client will issue request messages on a specific port number. The server will respond to those requests with response messages.

The protocol we will implement is a simple one that we are making up, which we'll call Tutorial17. The Tutorial17 protocol will use port 1000, and its response/request definitions are shown in Table 17.3.

You will need to run the client and server scripts on separate machines, and will need NQL on both machines.

There are seven steps in this tutorial:

Table 17.3 The Tutorial17 Protocol

REQUEST	RESPONSE
"hello"	"how are you?"
"date"	the current system date
"time"	the current system time
anything else	"come again?"

1. Launch the NQL development environment.
2. Enter the client script.
3. Enter the server script.
4. Run the server script.
5. Run the client script.
6. Understand the server script.
7. Understand the client script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Create a socket on a server and respond to requests from clients.
- Create a socket on a client and issue requests to servers.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

Since we are creating two scripts in this tutorial, you should launch the NQL development environment twice, on separate machines.

At this point, you should have two instances of the NQL development environment active on your desktop, each with an empty code window. Now you're ready to enter your scripts.

Step 2: Enter the Client Script

In the first instance of the NQL development environment, enter the script shown in Listing 17.1 and save it under the name Client.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Client.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Client.nql from the Tutorials\ch17 folder.

You need to make one change to the client script. Change the connect statement's parameter to be the IP address of your server (the same computer you are working on right now). This can be a numeric IP address such as "16.10.100.3" or a logical system name, such as "rwalters".

```
//client - socket client

opensocket "1000"
connect "172.16.200.63"

sRequest = "hello"
send sRequest
receive sResponse
show sResponse

sRequest = "date"
send sRequest
receive sResponse
show sResponse

sRequest = "time"
send sRequest
receive sResponse
show sResponse

sRequest = "surprise"
send sRequest
receive sResponse

show sResponse

closesocket
end
```

Listing 17.1 Client script.

At this point, the script in Listing 17.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

Step 3: Enter the Server Script

Now for the second script, the server side. In the second instance of the NQL development environment, on a different machine from step 2, enter the script shown in Listing 17.2 and save it under the name Server.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Server.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Server.nql from the Tutorials\ch17 folder.

```
//server - socket server

opensocket "1000"
listen
while
{
    receive sRequest
    if sRequest=="hello"
    {
        sResponse = "how are you?"
    }
    elseif sRequest=="date"
    {
        sResponse = date()
    }
    elseif sRequest=="time"
    {
        sResponse = time()
    }
    else
    {
        sResponse = "come again?"
    }
    send sResponse
    listen
}
closesocket
end
```

Listing 17.2 Server script.

At this point, the script in Listing 17.2 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD.

With both the client and server scripts entered, you are now ready to run them.

Step 4: Run the Server Script

Start by running the Server.nql script. Go to the instance of the NQL development environment where you entered the server script. To run the script, select *Build, Run* from the menu, click the *Run* toolbar button, or press F5.

When you run the server script, you see the hourglass indicating busyness, but nothing else appears to happen, no matter how long you wait. This is not a problem: It is exactly what the script should be doing, waiting for an incoming client connection. If this is not the behavior you see, check the following:

- Check your script for typographical errors.
- Check the NQL client's *Errors* tab for error messages.

Leave the server script running, and proceed to the next step.

Step 5: Run the Client Script

Now the client script can be run, on a different machine. Go to the instance of the NQL development environment where you entered the Client script. To run the script, select *Build, Run* from the menu, click the *Run* toolbar button, or press F5.

When you run the client script, a connection is made to the server and communication follows. The client issues four separate requests to the server, and each time it displays the response it received in a message box. If you do not see this happen, check the following:

- Make sure the client script specified the correct address for the server.
- Check your script for typographical errors.
- Make sure the server script is running.
- Check the NQL client's *Errors* tab for error messages.

The first request issued by the client is the text "hello", and the response received back should have been "how are you?", in agreement with the rules we devised for our imaginary Tutorial17 protocol.

The second request issued by the client is the text "date", and the response received back should have been the current system date.

The third request issued by the client is the text "time", and the response received back should have been the current system time.

The fourth request is a deliberate value that is not an expected command. In the Tutorial17 protocol, we chose "come again?" as the response for an unexpected request.

At this point you should have seen client/server socket communication in action.

Step 6: Understand the Server Script

We now want to make sure we understand every part of the scripts, beginning with the server script. The first line is a comment line.

```
//server - socket server
```

A socket is opened with the *opensocket* statement, specifying a port number of 1000. Why that number? Because that is the well-defined port number we invented for our tutorial's imaginary protocol. If you were writing an implementation of a standardized protocol, you would follow that protocol's port number convention. For example, you would specify port 80 if you were writing an HTTP Web server.

```
opensocket "1000"
```

Once the socket has been opened, the server's job is to be available for clients. The *listen* statement puts the server in a waiting mode. The script will not execute further until an incoming connection is made by a client.

```
listen
```

Once a client connects, the listen statement completes and sets a success condition. This causes us to enter a *while* loop to process the request. At the bottom of the loop is another *listen* statement. Thus, after the server handles a request it goes back to listening for the next incoming connection.

```
while
{
    ...process request...
    listen
}
```

Within the body of the *while* loop, a *receive* statement obtains the request message from the client and stores it in the variable *sRequest*.

```
receive sRequest
```

The value in *sRequest* is now checked against the values that might be reasonably expected, namely the definitions in our made-up protocol. A series of *if* and *elseif* statements check for the messages *hello*, *date*, and *time*. In each case, a suitable response is devised and stored in the variable *sResponse*, including a default response for unrecognized requests.

```
if sRequest=="hello"
{
    sResponse = "how are you?"
}
elseif sRequest=="date"
{
    sResponse = date()
}
elseif sRequest=="time"
{
    sResponse = time()
}
else
{
    sResponse = "come again?"
}
```

The response in *sResponse* is then sent back to the client with a *send* statement.

```
send sResponse
```

The server goes back to listening at this point. Because the server script is intended to run in perpetuity, it never gets to the final part of the script:

```
closesocket
end
```

At this point, you've seen the server's view of socket communication. Now let's get the client's perspective.

Step 7: Understand the Client Script

Let's go through the client script and see how it works. The first line is a comment line.

```
//client - socket client
```

A socket is opened on port 1000 with *opensocket*. The client and server must be using the same port number or nothing will work.

```
opensocket "1000"
```

A connection is now made to the server, which we assume is ready for us and awaiting us to connect. The connect statement specifies the IP address or name of the server.

```
connect "172.16.200.63"
```

After a connection has been made, the client is free to send requests to the server. We send four to demonstrate the range of commands we have defined, and to test the server's response to an unexpected command. For each command, a request is assembled in the variable *sRequest* and sent to the server via a *send* statement. The response back from the server is received into *sResponse* with a *receive* statement. The script uses a *show* statement to display the response on your screen.

```
sRequest = "hello"
send sRequest
receive sResponse
show sResponse

sRequest = "date"
send sRequest
receive sResponse
show sResponse

sRequest = "time"
send sRequest
receive sResponse
```

```
show sResponse

sRequest = "surprise"
send sRequest
receive sResponse
show sResponse
```

The socket is closed with *closesocket*, and the script ends.

```
closesocket
end
```

At this point, you've seen NQL scripts handle both the client and server sides of socket communication.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Change the custom protocol to be more robust. Allow the client to specify requests that include the names of files, and have the server respond with the contents of those files.
- Add error checking to handle failures in opening sockets or sending and receiving data. You could change *opensocket*, *connect*, *listen*, *send*, and *receive* to be functions rather than statements and check the true/false return values.
- Change the port number to 80 and turn the server script into a simple HTTP Web server. You can use a standard Web browser as the client.

Chapter Summary

TCP/IP is based on client/server socket communications.

- Servers await responses from clients and respond to them.
- Clients initiate connections to servers and issue requests.

IP addresses and port numbers are relied upon in socket communications.

- IP addresses appear as dotted numbers, such as 200.45.6.13, or as logical names such as www.my-site.com that are mapped to IP numbered addresses.
- Port numbers define the purpose of the communication. Well-known port numbers are used for standardized protocols such as HTTP, FTP, and many others. A client and server must use the same port number or they will not be able to communicate.

NQL contains statements for socket communication that permit scripts to act as clients or servers.

- Sockets are opened with the *opensocket* statement.
- Servers listen for incoming client connections with the *listen* statement.
- Clients connect to servers with the *connect* statement.
- Clients and servers send data with the *send* statement.
- Clients and servers receive data with the *receive* statement.
- Sockets are closed with the *closesocket* statement.

Socket support in NQL allows you to implement new protocols, including your own custom ones, with short, simple scripts.

CHAPTER 18

Serial Communications

Serial ports are a traditional way to connect modems, printers, and other kinds of local devices to computers. Serial communications allow you to interact with devices directly connected to your computer. Network Query Language provides basic services for sending and receiving data over serial ports. This chapter describes how serial programming is accomplished in NQL. The tutorial at the end of the chapter interacts with a digital scale to weigh merchandise.

Serial Programming in NQL

NQL has statements that allow scripts to work with serial ports. The NQL serial communication statements are listed in Table 18.1.

Table 18.2 lists the serial communications functions.

The serial communications statements and functions are described individually in the following sections.

Serial Communication Sessions

The NQL serial communication operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

Table 18.1 Serial Communications Statements

STATEMENT	PARAMETERS	DESCRIPTION
closecommport		Closes a communications port
opencommport	*port*	Opens a communications port
readcommport	*port*, *var*, *byte-count*	Receives data over a communications port
setcommport	*port*, *baud-rate*, *byte-size*, *parity*, *stop-bits*	Configures a communications port
setcommtimeout	*port*, [*read-interval-timeout* [, *read-total-timeout-multiplier* [, *read-total-timeout-constant* [, *write-total-timeout-multiplier* [, *write-total-timeout-constant*]]]]]	Sets the time-out features for read and write operations
writecommport	*port*, *data*, *byte-count*	Sends data over a communications port

Table 18.2 Serial Communications Functions

FUNCTION	DESCRIPTION
boolean opencommport(*port*)	Opens a communications port
boolean readcommport(*port*, *var*, *byte-count*)	Receives data over a communications port
boolean setcommport(*port*, *baud-rate*, *byte-size*, *parity*, *stop-bits*)	Configures a communications port
boolean setcommtimeout(*port*, [*read-interval-timeout* [, *read-total-timeout-multiplier* [, *read-total-timeout-constant* [, *write-total-timeout-multiplier* [, *write-total-timeout-constant*]]]]])	Sets the time-out features for read and write operations
boolean writecommport(*port*, *data*, *byte-count*)	Sends data over a communications port

1. A serial communications session is opened with *opencommport*.
2. The communication port is configured with *setcommport*.
3. Multiple reads and/or writes are performed using *readcommport* and *writecommport*.
4. The serial communications session is closed with *closecommport*.

Opening a Communications Port

The *opencommport* statement opens a serial port. The parameter is the name of the port, such as “COM1”.

```
opencommport port
boolean opencommport(port)
```

If the port is successfully opened, a success condition results (the function returns true). The following code opens the COM2 port.

```
opencommport "COM2"
```

The port is initially opened with the following settings: 9600 baud, 8-bit byte, no parity, and 1 stop bit. After opening the port, it may be configured to other settings with *setcommport*. Once a port has been successfully opened and configured, the *readcommport* and *writecommport* statements and functions may be used to pass data.

Configuring a Communications Port

The *setcommport* statement configures a serial port, specifying baud rate (speed) and other settings. The parameters are a port name, a baud rate, a byte size, parity, and the number of stop bits. The port must be open before *setcommport* can be used.

The baud rate can be any positive number the port supports. The byte size must be between 4 and 8 bits inclusive. The parity value must be one of the following string values: *noparity*, *oddparity*, *evenparity*, *markparity*, or *spaceparity*. The stop bit value must be either 1, 1.5, or 2.

```
setcommport port, baud-rate, byte-size, parity, stop-bits
boolean setcommport(port, baud-rate, byte-size, parity, stop-bits)
```

If the port is successfully configured to the new settings, a success condition results (the function returns true). The following code opens the COM1 port and configures it to 14,400 baud, a byte size of 8 bits, no parity, and 1 stop bit.

```
opencommport "com1"
setcommport "com1", "14400", "8", "noparity", "1"
writecommport "com1", "Hello", 5
closecommport "com1"
```

If you open a serial port and do not configure it with setcommport, the *opencommport* statement’s default conditions (9600 baud, 8-bit byte, no parity, 1 stop bit) apply.

Sending Data

The *writecommport* statement sends data over a serial port. The parameters are a port name, the data to send, and a byte count. The byte count is the maximum number of characters that are written at a time to the port.

```
writecommport port, data, byte-count
boolean writecommport(port, data, byte-count)
```

If the data is transmitted successfully, a success condition results (the function returns true). The following code uses *writecommport* to transmit the contents of a text file to a serial port.

```
opencommport "com1"
load "data.txt"
pop data
writecommport "com1", data, 1
closecommport
```

The reciprocal of *writecommport* is the *readcommport* statement, which receives data from a serial port.

Reading Data

The *readcommport* statement receives data over a serial port. The parameters are a port name, the data to send, and a byte count. The byte count is the maximum number of characters that are read at a time from the port.

```
readcommport port, var, byte-count
boolean readcommport(port, var, byte-count)
```

If the data is received successfully, a success condition results (the function returns true). The following code uses *readcommport* to receive data and store it in a text file.

```
opencommport "com1"
readcommport "com1", data, 1
push data
save "data.txt"
closecommport
```

The reciprocal of *readcommport* is the *writecommport* statement, which sends data to a serial port.

Configuring Timeouts

Reading and writing to serial ports can be immediate operations, or the operating system can be cued to wait a certain amount of time for characters to be sent or received.

The *setcommtimeout* statement controls a variety of timeout settings. The parameters for Windows are shown.

```
setcommtimeout port, [read-interval-timeout [, read-total-timeout-
multiplier [, read-total-timeout-constant [, write-total-timeout-
multiplier [, write-total-timeout-constant]]]]]
```

This statement's parameters and their exact descriptions are platform-specific. For more specific information, refer to the statement reference sheets in the NQL documentation.

Tutorial: Scale

Now it's time to put serial communications to work in an NQL script. In this tutorial, you will create script that interfaces to a digital scale connected to your computer via a serial port. The scale uses a simple interface: When it receives a carriage return over its serial connection, it weighs whatever object is on its bed and returns a numeric string indicating the weight, such as 0.0245.

You probably won't be able to run this tutorial for real, unless you happen to have access to a digital scale. If you have a different serial device, you may be able to modify the tutorial script accordingly to work with your device.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Scale script.
3. Run the Scale script.
4. Understand the Scale script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Open a serial port.
- Send and receive characters over a serial port.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or command line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

Step 2: Enter the Scale Script

In the NQL development environment, enter the script shown in Listing 18.1 and save it under the name Scale.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Scale.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Scale.nql from the Tutorials\ch18 folder.

At this point, the script in Listing 18.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

```
//Scale - interacts with a digital scale via a serial port

Port = "COM1"

//open comm port

if !opencommport(Port)
{
    show "Error: cannot open comm port " Port
    end
}
setcommport Port, "9600", "8", "noparity", "1"

//prompt user to weight something

show "Place an item on the scale, and click OK"

//get a weight from the scale

writecommport Port, "\r", 1
sleep 1
readcommport Port, Weight, 20

closecommport Port

//Display the weight

show "The weight is " Weight
end
```

Listing 18.1 Scale script.

Step 3: Run the Scale Script

Run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script completes, a weight from the scale is displayed on the screen. If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Check that the scale is connected to your computer.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Scale script run, interacting with a device via a serial port.

Step 4: Understand the Scale Script

We now want to make sure we understand every part of the Scale script. The first line is a comment line.

```
//Scale - interacts with a digital scale via a serial port
```

A variable named *Port* is set to the name of the serial port to be accessed (COM1).

```
Port = "COM1"
```

The serial port is opened for communications with the *opencommport* function. If it fails, an error message is displayed and the script ends. If the port is opened, its speed, number of data bits, parity, and other settings are *configured* with setcommport.

```
//open comm port

if !opencommport(Port)
{
    show "Error: cannot open comm port " Port
    end
}
setcommport Port, "9600", "8", "noparity", "1"
```

The user is now prompted to put something on the scale, as we don't want to request weight before we know something is on the scale's bed. The *show* statement prompts the user and waits until the message window's *OK* button is clicked.

```
//prompt user to weight something

show "Place an item on the scale, and click OK"
```

To obtain the weight from the scale, the scale is sent a carriage return with a *writecommport* statement. The scale should reply very quickly with a weight. The script waits one second with a *sleep* statement to be sure the response has been issued, then reads up to 20 characters with a *readcommport* statement. The weight is assigned to the variable *Weight*.

```
//get a weight from the scale
writecommport Port, "\r", 1
sleep 1
readcommport Port, Weight, 20
```

Serial communications are finished, so the serial port can be closed with *closecommport*.

```
closecommport Port
```

The weight is then displayed with a *show* statement, and the script ends.

```
//Display the weight

show "The weight is " Weight
end
```

At this point, you've seen how to transfer data to and from a device through a serial port in NQL.

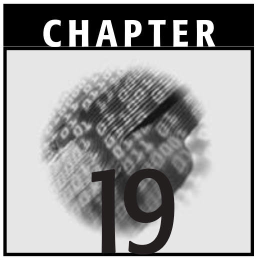
Chapter Summary

Network Query Language can communicate with serial ports.

- The *opencommport* statement opens a communications port.
- The *readcommport* statement receives data over a communications port.
- The *setcommport* statement configures a communications port.
- The *setcommtimeout* statement sets a variety of time-out features for read and write operations.
- The *writecommport* statement sends data over a communications port.

Serial port communication allows NQL scripts to interact with, monitor, and control local devices.

CHAPTER



19

Synchronization

This chapter describes Network Query Language's synchronization features. Many applications today are of a distributed nature, such that multiple programs need to cooperate with each other. Synchronization features allow multiple programs to coordinate their activities. The chapter describes both inter-process communication methods and locking methods and provides a tutorial for each.

Inter-process Communication

NQL allows scripts to communicate with each other through the use of *communication pipes*. Communication pipes are network wide, allowing even programs running on different computers to exchange messages. Table 19.1 summarizes the communication pipe statements.

The communication pipe functions are listed in Table 19.2.

The NQL communication pipe statements are described individually in the following sections.

Communication Pipe Sessions

The NQL communication pipe operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

Table 19.1 Communication Pipe Statements

STATEMENT	PARAMETERS	DESCRIPTION
closepipe		Closes a communications pipe
createpipe	*name* [, *type*]	Creates a new communication pipe
getpipe	[*var*]	Receives a message
openpipe	*name* [, *type*]	Connects to an existing communication pipe
sendpipe	[*data*]	Sends a message to the open communications pipe
waitpipe	[*var*]	Waits for a message and receives it

Table 19.2 Communication Pipe Functions

FUNCTION	DESCRIPTION
boolean createpipe(*name* [, *type*])	Creates a new communication pipe
string getpipe([*var*])	Receives a message
boolean openpipe(*name* [, *type*])	Connects to an existing communication pipe
boolean sendpipe([*data*])	Sends a message to the open communications pipe
boolean waitpipe([*var*])	Waits for a message and receives it

1. A communications pipe session is created with *createpipe* or *openpipe*.
2. One or more pipe operations take place, such as sending, receiving, or waiting, using *sendpipe*, *getpipe*, and *waitpipe*.
3. The communications pipe session is closed with *closepipe*.

Types of Communication Pipes

There are various names and types of communication pipes that vary between operating systems, but two common types are (to use Microsoft terminology) *mailslots* and *named pipes*, both of which are described in the following sections.

Pipes have to be identified in some way. Under Windows, mailslots and named pipes are identified by filename.type style names that resemble disk filenames. These names may be preceded by a \\server\ prefix if communicating with a remote system.

Mailslots

Mailslots are a one-way method of process-to-process communication. One program creates a mailslot with a specific name (the *server*), and one or more other programs (the *clients*) can send messages to the mailslot. Clients can send messages only to the server and the server can receive messages only from clients.

Named Pipes

Named pipes are a two-way method of process-to-process communication. One program creates a named pipe with a specific name (the *server*), and one or more other programs (the *clients*) can connect to it. Both the clients and the server are free to send messages to the server as well as receive responses from the server.

Creating a Communication Pipe

The *createpipe* statement or function creates a communication pipe for the server process. There are two parameters, a name and an optional type.

```
createpipe name [, type]
boolean createpipe(name [, type])
```

The first parameter specifies the name of the communication pipe. Some operating systems allow any kind of name, but others place restrictions on them. The second parameter is the type of communication pipe, which may be "mailslot" or "named pipe". This parameter may be omitted and defaults to "mailslot".

If *createpipe* encounters an error, a failure condition results (the function returns false). The following code creates a named pipe called shop.msg.

```
createpipe "shop.msg", "named pipe"
```

Opening a Communication Pipe

The *openpipe* statement opens an existing communication pipe for client processes. The statement form is

```
openpipe name, type
```

The first parameter specifies the name of the communication pipe.

The second parameter is the type of communication pipe, which may be "mailslot" or "named pipe". This parameter may be omitted and defaults to "mailslot".

The following sample *openpipe* statement connects to a named pipe called shop.msg on the server machine named server1.

```
openpipe "\\server1\\shop.msg", "named pipe"
```

If *openpipe* encounters an error, a failure condition results.

Sending a Message

Once a communication pipe has been created or opened, messages are sent with the *sendpipe* statement. The *sendpipe* statement takes the form below.

```
sendpipe message
```

There are no predefined rules about what a message may contain—that's up to you. You can invent your own format for messages, or you may want to adopt one of the standards for agent-to-agent communication, such as Knowledge Query Management Language (KQML). If you design your own message format, you may find it useful to include both a number and a text part to your messages. The numbers are more readily processed by programs, and the text is more easily reported to human beings.

The following sample *sendpipe* statement sends a status message to a pipe.

```
sendpipe "15 Out of disk space"
```

If *sendpipe* encounters an error, a failure condition results.

Receiving a Message

Once a communication pipe has been created or opened, messages are received with the *getpipe* statement. The *getpipe* statement takes one of the forms below.

```
getpipe
getpipe variable
```

Both forms of *getpipe* receive any pending message information from the pipe, setting a failure condition if nothing is received. The first form places the message received on the stack. The second form stores the message in the specified variable.

The following sample *getpipe* statement retrieves a message and stores it in the variable *msg*.

```
getpipe msg
```

If *getpipe* encounters an error, a failure condition results.

Waiting for a Message

The *waitpipe* statement waits for a message and retrieves it. It is just like *getpipe* except that it waits for an incoming message. The *waitpipe* statement takes one of the following forms.

```
waitpipe
waitpipe variable
```

Both forms of *waitpipe* wait for a message from the pipe and retrieve it. The first form places the received message on the stack. The second form stores the message in the specified variable.

The following sample *waitpipe* statement waits for a message and stores it in the variable *msg*.

```
waitpipe msg
```

If *waitpipe* encounters an error, a failure condition results.

Closing a Pipe

The *closepipe* statement terminates a pipe. You should call *closepipe* when you are finished working with a pipe. The *closepipe* statement takes the following form.

```
Closepipe
```

If you fail to close a pipe, it closes automatically when the script ends.

Tutorial: Communications

Now to observe communication in action. In this tutorial, you will create two scripts which will need to communicate with each other. The first script, comm1, will be the computation half of the solution. The second script, comm2, will prompt for a subtotal and tax rate, pass the information to comm1 for computation, and display the results.

1. Launch the NQL development environment.
2. Create the comm1 Script.
3. Create the comm2 Script.
4. Run both scripts and observe their interaction.

Step 1: Launch the NQL Development Environment

Launch two instances of the NQL development environment, since there are two scripts to enter and run. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client. Repeat the process to launch a second instance.

At this point, you should have two instances of the NQL development environment active on your desktop, with an empty code window in each. Now you're ready to enter the two scripts.

Step 2: Creating the Comm1 Script

Our first script will handle the computation side of things. It creates a pipe, waits to receive a subtotal and tax rate, and passes back the computed total.

In the NQL development environment, enter the script shown in Listing 19.1 and save it under the name comm1.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy comm1.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select comm1.nql from the Tutorials\ch19 folder.

Let's walk through what the statements in this script do. The first line creates a new named pipe called *tax.1*. The second line waits for an incoming message, which is stored in the variable *subtotal*. The third line acknowledges the communication by sending a response of "ack". The fourth line waits for an additional message which is stored in the variable *tax*. The fifth line responds with the computed total. Finally, the sixth line closes the pipe.

At this point you should have created a script named comm1.nql containing the code shown in Listing 19.1.

Step 3: Creating the Comm2 Script

Our second script will prompt for subtotal and tax rate, interact with comm1 for the computation, and display the total.

In the second instance of the NQL development environment, enter the script shown in Listing 19.2 and save it under the name comm2.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy comm2.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select comm2.nql from the Tutorials\ch19 folder.

Let's walk through what the statements in this script do. The first two lines prompt for a subtotal and store the result in the variable *subtotal*. The next two lines prompt for

```
//Comm1 - named pipe example (1 of 2)

float subtotal, tax

createpipe "tax.1", "named pipe"
waitpipe subtotal
sendpipe "ack"
waitpipe tax
sendpipe subtotal + (subtotal * tax)
closepipe
```

Listing 19.1 Comm1 script.

```
//Comm2 - named pipe example (2 of 2)

float subtotal, tax

prompt "Subtotal", "Enter a subtotal"
pop subtotal
prompt "Tax rate", "Enter a tax rate"
pop tax
openpipe "tax.1", "named pipe"
sendpipe subtotal
waitpipe ack
sendpipe tax
waitpipe total
show total
closepipe
```

Listing 19.2 Comm2 script.

a tax rate and store the result in the variable *tax*. The fifth line opens a connection to the pipe which was created by the other script, comm1.nql. The sixth line sends the subtotal and the seventh line waits for an acknowledgement. The eighth line sends the tax rate and the ninth line waits for the computed total. The tenth line displays the total. The eleventh line closes the pipe.

At this point you should have created a script named comm2.nql containing the code shown in Listing 19.2.

Step 4: Running Both Scripts

Now we are ready to run the scripts in parallel and observe how they interact.

1. Start comm1.nql running.
2. Start comm2.nql running.
3. When prompted, enter a subtotal amount (such as 150.00) and tax rate (such as 0.08).
4. The computed total will be displayed.

At this point you should have witnessed communication between two NQL scripts.

Inter-process Locking

When you need to control access to a shared resource, locking is essential. For example, consider multiple agents that wish to periodically update a data file. If they do not take turns, information could be lost or an agent might encounter a file-in-use error from the operating system. The use of locking allows this multiple process access to

Table 19.3 Locking Statements

STATEMENT	DESCRIPTION
lock	Sets a lock, waiting if lock is in use
unlock	Clears a lock

proceed smoothly. NQL employs a structure known as a *mutex* to implement interprocess locking. Table 19.3 summarizes the locking statements.

The lock and unlock statements are described individually in the following sections.

Setting a Lock

The *lock* statement sets a lock. The statement form is shown.

```
lock name
```

The one parameter is the name of the lock. If this parameter is omitted, the name *nql* is used by default.

The *lock* statement first determines if the named lock already exists or not. If it does, *lock* waits patiently until the lock is freed. After any necessary waiting, the named lock is created and script execution proceeds.

The sample *lock* statement below sets a lock named *customer-file*.

```
lock "customer-file"
```

When a script ends, any held locks are automatically unlocked.

Clearing a Lock

Once you are finished using a locked resource, the *unlock* statement is used to free up the lock. The statement form is shown.

```
unlock name
```

The one parameter is the name of the lock. If this parameter is omitted, the name *nql* is used by default.

The following sample *unlock* statement clears a lock named “customer-file”.

```
unlock "customer-file"
```

Tutorial: Locking

Let’s put locking to work. In this tutorial, you will create two scripts which will need to take turns accessing a disk file. The first script will periodically write to the file, while the second script will periodically read from it. Four steps are required, as follows:

1. Launch the NQL development environment.
2. Create the lock1 script.
3. Create the lock2 script.
4. Run both scripts and observe their interaction.

Step 1: Launch the NQL Development Environment

Launch two instances of the NQL development environment, since there are two scripts to enter and run. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client. Repeat the process to launch a second instance.

At this point, you should have two instances of the NQL development environment active on your desktop, with an empty code window in each. Now you're ready to enter the two scripts.

Step 2: Creating the Lock1 Script

Our first script will write the system time to a disk file every few seconds.

In the NQL development environment, enter the script shown in Listing 19.3 and save it under the name lock1.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy lock1.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select lock1.nql from the Tutorials\ch19 folder.

```
//Lock1 - demonstrate agent synchronization (1 of 2)

timeout 60
every 15
{
    lock "time"
    create "time.dat"
    time now
    write now
    close
    unlock "time"
}
```

Listing 19.3 Lock1 script.

At this point you should have created a script named comm1.nql containing the code shown in Listing 19.3.

Step 3: Creating the Lock2 Script

Our second script will read the system time from the same disk file every few seconds and output the time.

In the second instance of the NQL development environment, enter the script shown in Listing 19.4 and save it under the name lock2.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy lock2.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select lock1.nql from the Tutorials\ch19 folder.

At this point you should have created a script named comm2.nql containing the code shown in Figure 19.4.

Step 4: Running Both Scripts

Now it is time to run both scripts—simultaneously.

1. Start lock1.nql running.
2. Start lock2.nql running.
3. Wait for both scripts to finish running.
4. Examine the output of lock2.nql.

The output of lock2.nql will be the system time over five-second intervals as read from the disk file. You should notice pauses in the sequence, which reflect the times that lock2.nql had to wait for lock1.nql to finish writing.

```
//Lock2 - demonstrate agent synchronization (2 of 2)

timeout 60
every 5
{
    lock "time"
    open "time.dat"
    read now
    output now
    close
    unlock "time"
}
```

Listing 19.4 Lock2 script.

Chapter Summary

Two methods of synchronizing processes are communication and locking. Communication between processes is through communication pipes.

There are two kinds of communication pipes, mailslots and named pipes. Mailslots are unidirectional, named pipes are bidirectional. Once an NQL script creates a communication pipe, other scripts can connect to the pipe and send or receive messages.

- The *createpipe* statement creates a pipe on a server.
- The *openpipe* statement connects to a pipe.
- The *sendpipe* statement transmits data to a pipe.
- The *getpipe* statement receives data from a pipe.
- The *waitpipe* statement waits for data to be received from a pipe.
- The *closepipe* statement closes a pipe.

Locking between processes is through locks called mutexes.

- The *lock* statement sets a lock.
- The *unlock* statement frees a lock.

Communication and locking provide the synchronization that is essential for distributed processing solutions.

CHAPTER 20

E-Commerce

Network Query Language has features for submitting credit card transactions and for interacting with financial organizations. This chapter begins with merchant credit card submission, followed by Open Financial Exchange. The tutorial at the end of the chapter submits a credit card approval request.

Credit Card Transactions

Accepting payments by credit card involves a partnership of three organizations: a merchant, a bank, and a service provider. The merchant is you, the person or organization accepting payments. The bank is your bank. The service provider is a company that interfaces with credit card systems such as Visa and MasterCard, as well as with the thousands of banks and other organizations that issue credit cards. Figure 20.1 shows the relationship between your customer, you the merchant, your service provider, the credit card system, and the banks.

Your customers deal only with you, providing credit card information when they wish to order a product. Your applications identify themselves to the merchant provider and pass on both the credit card information and the amount to approve, debit, or credit. The service provider then interacts with many parties. The credit card system is accessed to ensure the card is valid, has not expired, and has sufficient funds

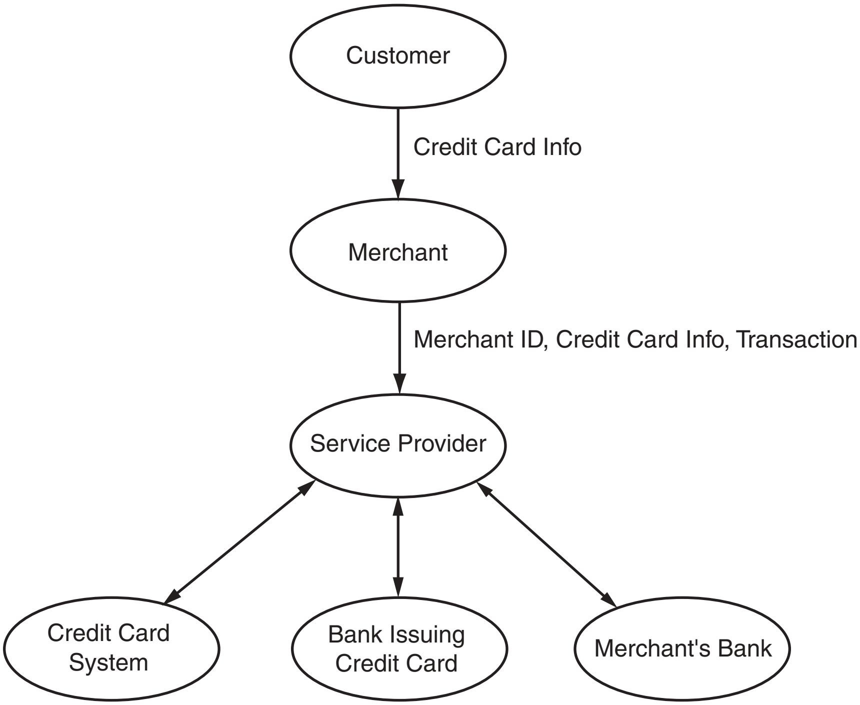


Figure 20.1 Credit card processing.

available. If everything validates, funds are removed from the bank that issued the customer's credit card and are deposited in your merchant account, minus processing fees.

The method of interacting with service providers varies from one provider to the next. NQL supports several of the popular providers. At the time of this writing, the two supported providers are CyberCash and CyberSource, but be aware that the supported providers may change over time. If you aren't using one of the supported providers, you won't be able to use NQL for credit card payments.

Before you can address the technical side of accepting credit card payments, you will need to choose a service provider. You will need to spend time working closely with the service provider and your bank, which could take days to weeks. On the technical front, you will need to set up a dedicated e-commerce server, obtain a digital certificate, and write an e-commerce application. The specific steps required are as follows:

1. Select a service provider, such as CyberCash or CyberSource. Be sure to find out what computer platforms the service provider's software is compatible with.
2. Fill out an application with the service provider. The service provider will in turn work directly with your bank.
3. Follow any steps outlined by your bank and service provider to set up a merchant account. This may take days to weeks, depending on the organizations you are working with.

4. Your service provider will provide you with account information, a software development kit, and programming instructions and examples.
5. Set up a dedicated e-commerce server, and install the service provider's software on it.
6. Obtain a digital certificate from a certificate authority (such as VeriSign) for your e-commerce server. Your bank and service provider will insist on the use of a digital certificate and the Secure Sockets Layer (SSL, also known as HTTPS) for security reasons.
7. Develop and test your e-commerce application.
8. Instruct your service provider when you want to switch from test status to live status.

There are three basic operations in the world of credit card processing: approvals, charges, and refunds. An *approval* checks the availability of funds for a future charge; a *charge* withdraws funds; a *refund* returns funds.

NQL's Transaction Operations

The NQL statements for credit card transactions are listed in Table 20.1.

There are function versions of some of the credit card transaction statements. The credit card functions are shown in Table 20.2.

The transaction statements and functions are described individually in the following sections.

Transaction Sessions

The NQL transaction operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

Table 20.1 Transaction Statements

STATEMENT	PARAMETERS	DESCRIPTION
closetrx		Closes a transaction session
opentrx	*provider*	Opens a transaction session
trxapprove	[*message-type*]	Submits an approval
trxcharge	[*message-type*]	Submits a charge
trxget	[*field-name* [, *var*]]	Retrieves a field value after an operation
trxid	*merchant-ID*, *merchant-key*, *provider-url*	Identifies a merchant account to a service provider
trxrefund	[*message-type*]	Submits a refund
trxset	*field-name*, *value*	Sets a field value before an operation

Table 20.2 Transaction Functions

FUNCTION	DESCRIPTION
boolean opentrx(*provider*)	Opens a transaction session
boolean trxapprove([*message-type*])	Submits an approval
boolean trxcharge([*message-type*])	Submits a charge
boolean trxget([*field-name* [, *var*]])	Retrieves a field value after an operation
boolean trxid(*merchant-ID*, *merchant-key*, *provider-url*)	Identifies a merchant account to a service provider
boolean trxrefund([*message-type*])	Submits a refund
boolean trxset(*field-name*, *value*)	Sets a field value before an operation

1. A transaction session is opened with *opentrx*.
2. Merchant identification takes place with *trxid*.
3. One or more transactions are issued. For each transaction, fields are set with *trxset*, the transaction itself is issued with *trxapprove*, *trxcharge*, or *trxrefund*, and result fields may be examined with *trxget*.
4. The transaction session is closed with *closetrx*.

Opening a Transaction Session

A transaction session is opened with the *opentrx* statement or function. The parameter is the name of a provider NQL supports, such as “cybercash” or “cybersource”.

```
opentrx provider
boolean opentrx(provider)
```

If the transaction is opened successfully, a success condition results (the function returns true). No communication with the service provider has taken place yet, however. All *opentrx* does is prepare NQL for transaction work and identify the service provider to be used. The following code opens a transaction session for CyberSource.

```
opentrx "cybersource"
```

Once a transaction session has been opened, merchant identification is necessary using the *trxid* statement.

Closing a Transaction Session

Once all credit card processing is completed, a transaction session is closed with the *closetrx* statement. There are no parameters.

```
clostrx
```

The transaction session is terminated. The following code opens, then closes a transaction session.

```
opentrx "cybersource"
    ...
closetrx
```

If you don't explicitly close a transaction session, it is closed automatically when a script terminates.

Merchant Identification

After opening a transaction session with *opentrx*, you must identify your merchant information using the *trxid* statement or function. The parameters are a merchant ID, a merchant key, and a URL for accessing the service provider. Your service provider will supply you with your account information and a URL.

```
trxid merchant-ID, merchant-key, provider-url
boolean trxid(merchant-ID, merchant-key, provider-url)
```

If the fields you specify appear to be valid, a success condition results (the function returns true).

The specific meaning and format of the three parameters is specific to your service provider. For CyberCash, the *merchant-ID* parameter is the name of a local file (such as merchant_conf.txt) that is created by the CyberCash software. Be sure to specify a full path. The *merchant-key* field is not used and may be an empty string. The *url* parameter is the test URL or live URL CyberCash has assigned to you. The following code opens a transaction and performs merchant identification for CyberCash.

```
opentrx "cybercash"
trxid "c:\\inetpub\\wwwroot\\merchant_conf.txt", "",
    "cr.cybercash.com/cgi-bin/cr21api.cgi/"
```

For CyberSource, you will have been issued a *merchant-ID* (that is, an account name such as acmeshipping) and *merchant-key* (a PIN number or password, such as 03). The *url* parameter is the test URL or live URL CyberSource has assigned to you. The following code opens a transaction and performs merchant identification for CyberSource.

```
opentrx "cybersource"
trxid "customerone", "17", "ics2test.ic3.com"
```

Once merchant identification is completed, the other transaction statements and functions may be used to process approvals, charges, and refunds.

Setting Fields

Prior to performing an approval, charge, or refund, credit card information and transaction information need to be specified. The *trxset* statement or function sets an information field. The first parameter is a field name. The second parameter is a value for the field. The specific fields required are dependent upon your service provider.

```
trxset field-name, value
boolean trxset(field-name, value)
```

If the field name is recognized, a success condition results (the function returns true). The following code sets various fields before performing an approval:

```
trxset "card-number", "4113764542105701"
trxset "card-exp", "10/00"
trxset "card-name", "Sidney Jamph"
trxset "order-id", "neworder3qaed058394y"
trxset "amount", "usd 124.99"
if trxapprove()
{
    ...approval was granted...
}
else
{
    ...declined...
}
```

Once fields have been set with *trxset*, the *trxapprove*, *trxcharge*, or *trxrefund* statements may be used to perform a transaction.

Issuing an Approval

An approval is issued using the *trxapprove* statement or function. The optional parameter is a message type, which is used by some providers.

```
trxapprove [message-type]
boolean trxapprove([message-type])
```

When using CyberCash, the optional parameter specifies a CashRegister message. See your Merchant Connection Kit documentation provided by CyberCash for a listing of these messages and how to use them. If CyberCash is the service provider and the parameter is omitted, the message defaults to “mauthonly”. The message parameter is ignored if CyberSource is the service provider.

Prior to issuing *trxapprove*, various fields must be set using the *trxset* statement. The specific fields required are dependent on your service provider.

If the transaction is successful, a success condition results (the function returns true). The following code issues an approval.

```
trxset "card-number", "4113764542105701"
trxset "card-exp", "10/00"
trxset "card-name", "Sidney Jamph"
trxset "order-id", "neworder3qaed058394y"
trxset "amount", "usd 124.99"
if trxapprove()
{
    ...approval was granted...
}
else
{
    ...declined...
}
```

Once an approval has taken place, result fields may be retrieved using *trxget*.

Issuing a Charge

A charge is issued using the *trxcharge* statement or function. The optional parameter is a message type, which is used by some providers.

```
trxcharge [message-type]
boolean trxcharge([message-type])
```

When using CyberCash, the optional parameter specifies a CashRegister message. See your Merchant Connection Kit documentation provided by CyberCash for a listing of these messages and how to use them. If CyberCash is the service provider and the parameter is omitted, the message defaults to “mauthonly”. The message parameter is ignored if CyberSource is the service provider.

Prior to issuing *trxcharge*, various fields must be set using the *trxset* statement. The specific fields required are dependent on your service provider.

If the transaction is successful, a success condition results (the function returns true). The following code issues a charge.

```
trxset "card-number", "4113764542105701"
trxset "card-exp", "10/00"
trxset "card-name", "Sidney Jamph"
trxset "order-id", "neworder3qaed058394y"
trxset "amount", "usd 124.99"
if trxcharge()
{
    ...charge was granted...
}
else
```

```
{
    ...declined...
}
```

Once a charge has taken place, result fields may be retrieved using *trxget*.

Issuing a Refund

A refund is issued using the *trxrefund* statement or function. The optional parameter is a message type, which is used by some providers.

```
trxrefund [message-type]
boolean trxrefund([message-type])
```

When using CyberCash, the optional parameter specifies a CashRegister message. See your Merchant Connection Kit documentation provided by CyberCash for a listing of these messages and how to use them. If CyberCash is the service provider and the parameter is omitted, the message defaults to "mauthonly". The message parameter is ignored if CyberSource is the service provider.

Prior to issuing *trxrefund*, various fields must be set using the *trxset* statement. The specific fields required are dependent on your service provider.

If the transaction is successful, a success condition results (the function returns true). The following code issues a refund.

```
trxset "card-number", "4113764542105701"
trxset "card-exp", "10/00"
trxset "card-name", "Sidney Jamph"
trxset "order-id", "neworder3qaed058394y"
trxset "amount", "usd 124.99"
if trxrefund()
{
    ...refund was granted...
}
else
{
    ...declined...
}
```

Once a refund has taken place, result fields may be retrieved using *trxget*.

Retrieving Fields

After performing an approval, charge, or refund, result information can be retrieved. The *trxget* statement or function retrieves an information field. The first parameter is a field name. The second parameter is a variable to receive the field value. The specific field names used are dependent upon your service provider.

```
trxget [field-name [, var]]
boolean trxget([field-name [, var]])
```

If no parameters are specified, *trxget* returns the entire response data from your last transaction on the stack. The format of this response data is specific to your service provider.

If you specify one parameter only, a field name, that field's value alone is pushed onto the stack. If you specify both field name and variable parameters, the field's value is stored in the specified variable.

If the field name is recognized, a success condition results (the function returns true). The following code retrieves various fields after performing an approval:

```
trxset "card-number", "4113764542105701"
trxset "card-exp", "10/00"
trxset "card-name", "Sidney Jamph"
trxset "order-id", "neworder3qaed058394y"
trxset "amount", "usd 124.99"
if trxapprove()
{
    trxget "auth-code", AuthorizationCode
    trxget "order-id", OrderID
}
```

Open Financial Exchange

Open Financial Exchange (OFX) is a communications protocol that allows a large body of financial institutions to be safely accessed by a variety of organizations and applications. It is used by products such as Intuit's Quicken and Microsoft Money to perform online banking. The full specification on OFX is available at www.ofx.net.

Network Query Language supports OFX, allowing your programs to connect to financial organizations and interact with them. Depending on what a bank authorizes you to do, it is possible for your programs to download account information or even transfer funds.

A financial organization supports OFX by running an *OFX serve*r. The OFX server accepts secure Web requests (using SSL, the https:// protocol) and responds with information, confirmation of an operation, or an error message. Someone wanting to access a financial organization issues secure Web requests, much as a browser would, and accepts the responses. The software doing the accessing is known as an *OFX client.* An OFX client can be a well-known application, such as Quicken, or an application you create. It is very simple to create an OFX client application in NQL. Figure 20.2 illustrates the OFX communication between client and server.

Although Web protocols are used for communication, OFX is not the Web. Unlike the Web's URL requests and HTML responses, OFX requests and responses are in a specific format, using Structured Generalized Markup Language (SGML). The OFX specification describes the various requests and responses that are legal in OFX. You may find that some financial organizations use their own specific formats for requests and responses, which they may or may not be willing to share with you.

As widespread as OFX is, you should be aware that not all financial organizations choose to support it. Even among those who do support OFX, you may need to obtain

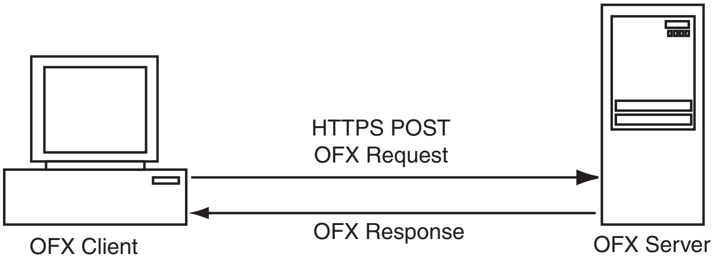


Figure 20.2 Open Financial Exchange.

authorization for OFX access in order to learn the URL, a valid ID, and a valid password or PIN. OFX requests may require that you also know a bank's routing/transit number (RTN), your own account number, or other information.

NQL's OFX Operations

The NQL OFX statements are listed in Table 20.3. The OFX functions are shown in Table 20.4. The OFX statements and functions are described individually in the following sections.

Table 20.3 OFX Statements

STATEMENT	PARAMETERS	DESCRIPTION
ofxget	*field-name*, *var*	Retrieves an SGML field from an OFX response
ofxload	*filespec*	Loads an OFX template
ofxsend	*url*	Sends an OFX request
ofxset	*field-name*, *value*	Sets an SGML field prior to issuing an OFX request

Table 20.4 OFX Functions

FUNCTION	DESCRIPTION
boolean ofxget(*field-name*, *var*)	Retrieves an SGML field from an OFX response
boolean ofxload(*filespec*)	Loads an OFX template
boolean ofxsend(*url*)	Sends an OFX request
boolean ofxset(*field-name*, *value*)	Sets an SGML field prior to issuing an OFX request

OFX Sessions

The NQL transaction operations are modeled on the idea of a session in which the statements and functions must be used in a certain order.

1. An OFX template file is loaded onto the stack using *ofxload*.
2. Field values are set using *ofxset*.
3. The OFX request is sent, using *ofxsend*. The response replaces the request on the stack.
4. Response fields are retrieved using *ofxget*.
5. The response is popped off of the stack.

Loading an OFX Template

A transaction template is an SGML file with placeholders for information that will need to be set at run time, such as the account number and PIN of a bank account. The *ofxload* statement or function loads a template file and pushes it onto the stack. The parameter is a filename. By default, the file type .ofx is assumed. Unless you specify a full path, the template will be loaded from the NQL \ofx directory.

```
ofxload template
boolean ofxload(template)
```

If successful, an SGML document will be on the stack and a success condition results (the function returns true). The following code uses *ofxload* to load a template.

```
ofxload "acctinfo"
```

Once a template has been loaded, use *ofxset* to configure its fields and *ofxsend* to send the request.

Setting OFX Fields

Once a template has been loaded, fields in the template need to be set with real values. This is the purpose of the *ofxset* statement. The parameters are a field name and a value.

```
ofxset field-name, value
boolean ofxset(field-name, value)
```

If the specified field name is found as a tag in the SGML template on the stack, it is set to the specified value and a success condition results (the function returns true). The following code uses *ofxset* to set fields in an account information request.

```
errors continue
if ofxload("acctinfo")
```

```
{
    ofxset "USERID", "450657499"
    ofxset "USERPASS", "uhu*32424"
    ofxset "ORG", "Bank1"
    ofxset "APPID", "Agent"
    ofxsend "https://ofx.bank1.com/ofx/ofx_server.dll"
}
```

After setting transaction fields with *ofxset*, use *ofxsend* to send the request.

Sending a Request

Once a template has been loaded and field values have been set, the *ofxsend* statement is used to send the request to an OFX server. The parameter is the secure URL of the OFX server.

```
ofxsend url
boolean ofxsend(url)
```

If the request is successfully transmitted, a response is received which replaces the request on the stack and a success condition results (the function returns true). The following code uses *ofxsend* to send an OFX request.

```
ofxload "acctinfo"
ofxset "USERID", "450657499"
ofxset "USERPASS", "uhu*32424"
ofxset "ORG", "Bank1"
ofxset "APPID", "Agent"
if ofxsend("https://ofx.bank1.com/ofx/ofx_server.dll")
{
    ofxget "DESC", name
    ofxget "PHONE", phone
    ofxget "STATUS", status
    ofxget "MESSAGE", message
    show name, phone, status, message
    show
}
else
{
    show "ofxsend failed"
}
```

After a successful *ofxsend*, the response data, an SGML document, will be on the stack. To retrieve individual fields, use the *ofxget* statement.

Retrieving OFX Fields

After an OFX request has been sent, fields may be retrieved from the response using the *ofxget* statement. The parameters are a field name and a variable to receive the field value.

```
ofxget field-name, var
boolean ofxget(field-name, var)
```

If the specified field name is found as a tag in the SGML template on the stack, its value is returned in the specified variable and a success condition results (the function returns true). The following code uses *ofxget* to retrieve fields after an account information request.

```
ofxload "acctinfo"
ofxset "USERID", "450657499"
ofxset "USERPASS", "uhu*32424"
ofxset "ORG", "Bank1"
ofxset "APPID", "Agent"
if ofxsend("https://ofx.bank1.com/ofx/ofx_server.dll")
{
    ofxget "DESC", name
    ofxget "PHONE", phone
    ofxget "STATUS", status
    ofxget "MESSAGE", message
    show name, phone, status, message
    show
}
else
{
    show "ofxsend failed"
}
```

After retrieving all desired information from the OFX response, use *pop* to clean up the stack.

Tutorial: Approval

Now it is time to put e-commerce to work in an actual NQL program. In this tutorial, you will create a script that submits a credit card approval using CyberSource. You won't actually be able to run this script unless you have a merchant account and are set up with CyberSource for electronic credit card processing.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Approval script.
3. Run the Approval script.
4. Understand the Approval script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Open an e-commerce session.
- Set the submittal fields prior to an approval.
- Submit an approval to CyberSource.
- Retrieve the result fields after an approval.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter a script.

Step 2: Enter the Approval Script

In the NQL development environment, enter the script shown in Listing 20.1 and save it under the name approval.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Approval.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select approval.nql from the Tutorials\ch20 folder.

At this point, the script in Listing 20.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Approval Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. If all goes well, a credit card approval will be processed, and the result codes from the transaction will be displayed. If this does not happen, check the following:

- Check for typographical errors in your script.
- Check that the merchant identification information is correct.
- Check that the payment information is correct.
- Check the NQL client's *Errors* tab for error messages.

```
/Approval - submits a credit card approval through CyberSource

    //open transaction session

    if !opentrx("cybersource") { goto ErrorUnknownVendor }
    //identify ourselves

    if !trxid("ICS2Test", "01", "ics2test.ic3.com") { goto ErrorBadID
}

    //set up the fields for an approval

    if !trxset("customer_firstname", "John") { goto ErrorInvalidField
}
    if !trxset("customer_lastname", "Doe") { goto ErrorInvalidField }
    if !trxset("customer_email", "johndoe@yourcompany.com") { goto
ErrorInvalidField }
    if !trxset("customer_phone", "714-555-5555") { goto
ErrorInvalidField }
    if !trxset("customer_cc_number", "4111111111111111") { goto
ErrorInvalidField }
    if !trxset("customer_cc_expmo", "08") { goto ErrorInvalidField }
    if !trxset("customer_cc_expyr", "2002") { goto ErrorInvalidField }
    if !trxset("bill_address1", "555 Main Street") { goto
ErrorInvalidField }
    if !trxset("bill_city", "Santa Ana") { goto ErrorInvalidField }
    if !trxset("bill_state", "CA") { goto ErrorInvalidField }
    if !trxset("bill_zip", "92704") { goto ErrorInvalidField }
    if !trxset("bill_country", "US") { goto ErrorInvalidField }
    if !trxset("currency", "USD") { goto ErrorInvalidField }
    if !trxset("offer0",
"amount:1.00^merchant_product_sku:Toy5555^quantity:1") { goto
ErrorInvalidField }

    //submit the approval

    if !trxapprove() { goto ErrorServer }

    //retrieve approval results

    if !trxget("ics_rcode", sCode) { goto GeneralError }
    show sCode
    closetrx
    end

ErrorUnknownVendor:
```

continues

Listing 20.1 Approval script.

```
    show "Error: cannot open transaction"
    end

ErrorBadID:
    show "Error: invalid ID parameters"
    closetrx
    end

ErrorInvalidField:
    show "Error setting named value pairs"
    closetrx
    end
```

Listing 20.1 Approval script (continued).

At this point you should have seen the Approval script run, accessing search engine sites in parallel to find the fastest one. In the next step, we will dissect the script and explain exactly how it works.

Step 4: Understand the Approval Script

We now want to make sure we understand every part of the Approval script. The first line is a comment line.

```
//Approval - submits a credit card approval through CyberSource
```

A transaction session is opened with *opentrx*. The provider is identified as CyberSource. If the session cannot be opened, a jump takes place to an error handler that displays an error message and ends the script. One reason *opentrx* might fail is if you don't have CyberSource's software set up on your computer for NQL to interface with.

```
//open transaction session

if !opentrx("cybersource") { goto ErrorUnknownVendor }
```

The script next identifies itself with the *trxid* function, specifying merchant identification information and a URL for submitting transactions. Again, if an error occurs, a jump to an error-handling section of the script takes place.

```
//identify ourselves

if !trxid("ICS2Test", "01", "ics2test.ic3.com") { goto ErrorBadID }
```

The details of the transaction now need to be defined. The *trxset* statements set field values that define the approval. These include the customer name and contact information, billing address, credit card information, and amount. The specific fields and formats of their values are dependent on the provider in use (CyberSource, in this case). If any of the *trxset* functions return false, it is unsafe to proceed and a jump is made to an error handler.

```
//set up the fields for an approval

if !trxset("customer_firstname", "John") { goto ErrorInvalidField }
if !trxset("customer_lastname", "Doe") { goto ErrorInvalidField }
if !trxset("customer_email", "johndoe@yourcompany.com") { goto
ErrorInvalidField }
if !trxset("customer_phone", "714-555-5555") { goto ErrorInvalidField }
if !trxset("customer_cc_number", "4111111111111111") { goto
ErrorInvalidField }
if !trxset("customer_cc_expmo", "08") { goto ErrorInvalidField }
if !trxset("customer_cc_expyr", "2002") { goto ErrorInvalidField }
if !trxset("bill_address1", "555 Main Street") { goto ErrorInvalidField }
if !trxset("bill_city", "Santa Ana") { goto ErrorInvalidField }
if !trxset("bill_state", "CA") { goto ErrorInvalidField }
if !trxset("bill_zip", "92704") { goto ErrorInvalidField }
if !trxset("bill_country", "US") { goto ErrorInvalidField }
if !trxset("currency", "USD") { goto ErrorInvalidField }
if !trxset("offer0", "amount:1.00^merchant_product_sku:Toy5555^
quantity:1") { goto ErrorInvalidField }
```

The approval has now been defined, and can be submitted to the provider. The *trxapprove* function returns true if the approval is allowed. If it is not, a jump to an error handler occurs.

```
//submit the approval

if !trxapprove() { goto ErrorServer }
```

The approval results can now be retrieved. The *trxget* function is used to retrieve an approval code. A *show* statement displays the approval code, and the script ends.

```
//retrieve approval results

if !trxget("ics_rcode", sCode) { goto GeneralError }
show sCode
closetrx
end
```

The error handlers are next. Each displays an appropriate error message and ends the script.

```
ErrorUnknownVendor:
    show "Error: cannot open transaction"
    end
```

```
ErrorBadID:
    show "Error: invalid ID parameters"
    closetrx
    end

ErrorInvalidField:
    show "Error setting named value pairs"
    closetrx
    end
```

At this point, you've witnessed NQL-driven credit card transactions.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Submit charges.
- Submit refunds.

Chapter Summary

Network Query Language has the ability to submit credit card approvals, charges, and debuts.

- The two supported providers are CyberCash and CyberSource. NQL knows how to interface with these providers' software. If you use a different provider, you won't be able to use NQL's credit card features.
- A transaction session is opened with the *opentrx* statement.
- A merchant identification is made with the *trxid* statement.
- Before a transaction, fields are set with the *trxset* function.
- After a transaction, fields are retrieved with the *trxget* function.
- The *trxapprove* statement submits an approval.
- The *trxcharge* statement submits a charge.
- The *trxrefund* statement submits a refund.
- The required format of parameters and required fields are dependent on the provider.

Credit card processing abilities in NQL allow you to create connected applications that can perform transactions.

CHAPTER 21

Graphics

Graphics and presentation are important elements of modern software. Network Query Language is one of the few programming languages that can create images dynamically on the fly, including Web graphics. Some examples of what can be achieved with NQL's graphics statements are listed.

- An NQL-powered Web site can create dynamic labels and buttons that contain personalized text for the current visitor.
- An agent can acquire images from a Web camera and add a caption and timestamp to each image.
- A logo graphic can be overlaid with custom text, such as a restaurant sign showing the number of hamburgers sold.
- A performance chart can be created from historical stock quote data and be displayed, emailed, or viewed through the Web.
- A graphic in BMP format can be converted to JPEG or GIF format for inclusion in a Web page.

In addition to generating graphics, NQL can also interface with TWAIN digital devices such as scanners and cameras. Captured images can be saved or modified using the graphics statements.

Image Operations

The NQL image operations provide general facilities for creating and modifying images. They include the ability to set colors, draw shapes, and render text.

The NQL statements for image operations are listed in Table 21.1.

There are function versions of some of the image statements. The image functions are shown in Table 21.2.

The image statements and functions are described individually in the following sections.

Table 21.1 Image Statements

STATEMENT	PARAMETERS	DESCRIPTION
backcolor	[#*image-ID*,] *color*	Sets the background color
closeimage	[#*image-ID*]	Closes an image session
createimage	[#*image-ID*,] *width*, *height*	Creates a new image session
drawcircle	[#*image-ID*,] *x1*, *y1*, *x2*, *y2*	Draws a circle or ellipse
drawcolor	[#*image-ID*,] *color*	Sets the drawing color
drawfont	[#*image-ID*,] *state*, *size*, *style*	Sets the font for text rendering
drawline	[#*image-ID*,] *x1*, *y1*, *x2*, *y2*	Draws a line
drawpie	[#*image-ID*,] *x*, *y*, *radius*	Draws a pie chart
drawpieslice	[#*image-ID*,] *percent* [, *color* [, *text* [, *degree*]]]	Draws a pie chart slice
drawpoint	[#*image-ID*,] *x*, *y*	Draws a point
drawrectangle	[#*image-ID*,] *x1*, *y1*, *x2*, *y2*	Draws a rectangle
drawstyle	[#*image-ID*,] *style*	Sets the drawing style
drawtext	[#*image-ID*,] *x*, *y*, *text* [, *rotation*]	Draws text, left-aligned
drawtextcenter	[#*image-ID*,] *x*, *y*, *text* [, *rotation*]	Draws text, center-aligned
drawtextright	[#*image-ID*,] *x*, *y*, *text* [, *rotation*]	Draws text, right-aligned
drawwidth	[#*image-ID*,] *pixels*	Sets the drawing width
imagecopy	*#image-ID1*, *x1*, *y1*, *width*, *height*, *#image-ID2*, *x2*, *y2*	Copies a section of an image to another image
openimage	[#*image-ID*,] *filespec*	Opens an existing image
saveimage	[#*image-ID*,] *filespec*	Saves an image to disk

Table 21.2 Image Functions

FUNCTION	DESCRIPTION
boolean closeimage([#*image-ID*])	Closes an image session
boolean createimage([#*image-ID*,] *width*, *height*)	Creates a new image session
boolean imagecopy(#*image-ID1*, *x1*, *y1*, *width*, *height*, #*image-ID2*, *x2*, *y2*)	Copies a section of an image to another image
boolean openimage([#*image-ID*,] *filespec*)	Opens an existing image
boolean saveimage([#*image-ID*,] *filespec*)	Saves an image to disk

Image Sessions

The NQL image operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. An image session is opened with *createimage* or *openimage*.
2. Any desired series of drawing operations is performed using the other image statements.
3. The image is saved to disk using *saveimage*.
4. The image session is closed with *closeimage*.

Multiple Image Sessions

NQL uses a very simple syntax for image operations, such as the following line.

```
createimage 800, 600
```

When there is a need to open multiple images simultaneously, identifiers are needed to distinguish one image session from another. In NQL, these identifiers are known as image session IDs. An image session ID is a name preceded by a pound sign, such as #*chart* or #*logo*. When a session ID is specified, it is the first parameter of a statement or function. For example,

```
createimage #pic1, 640, 480
```

Creating a New Image

The *createimage* statement or function begins an image session by creating a new image in memory. The parameters are a width and height, both in pixels.

```
createimage [#image-ID,] width, height
boolean createimage([#image-ID,] width, height)
```

If the image is created successfully, a success condition results (the function returns true). The draw settings are initialized to a background color of white, a draw color of black, a drawing width of one pixel, and a draw style of hollow. The following code uses *createimage* to create a 600 × 800 pixel image.

```
createimage 800, 600
```

To open an existing image instead of creating a new one, use *openimage*. Once an image session has been created, use the other image statements to set colors, draw shapes, set fonts, and render text. To save the image on disk, use *saveimage*.

Opening an Existing Image

The *openimage* statement or function begins an image session by loading an existing image from a disk file. The parameter is the file specification of a .bmp, .gif, or .jpg file.

```
openimage [#image-ID,] filespec
boolean openimage([#image-ID,] filespec))
```

If the image is opened successfully, a success condition results (the function returns true). The draw settings are initialized to a background color of white, a draw color of black, a drawing width of one pixel, and a draw style of hollow. The following code uses *openimage* to open a GIF file.

```
openimage "test.gif"
```

Once an image session has been created, use the other image statements to set colors, draw shapes, set fonts, and render text.

Saving an Image

The *saveimage* statement or function saves an image to disk. The parameter is the file specification of a .bmp, .gif, or .jpg file.

```
saveimage [#image-ID,] filespec
boolean saveimage([#image-ID,] filespec)
```

If the image is saved successfully, a success condition results (the function returns true). The following code creates an image and saves it to disk as a .jpg file.

```
createimage 100, 100
drawtext 0,0, "hello"
drawcircle 20, 20, 60, 60
saveimage "test.jpg"
```

Once an image has been saved, the image session in memory can be freed through *closeimage*.

Closing an Image Session

Once you are finished with an image session, it can be closed with *closeimage*. There are no parameters other than the optional image ID.

```
closeimage [#image-ID]
boolean closeimage([#image-ID])
```

The image session is terminated. The following code opens and then closes an image session.

```
openimage 400, 200
    ...
closeimage
```

If you don't explicitly close an image session, it is closed automatically when a script terminates.

Drawing Circles and Ellipses

The *drawcircle* statement draws a circle or ellipse. The parameters are the X-Y coordinates of the top-left and bottom-right corners of the bounding rectangle of the ellipse.

```
drawcircle [#image-ID,] x1, y1, x2, y2
```

The specified ellipse is drawn onto the image in the current draw color. The thickness of the line is controlled by the current draw width. The ellipse is hollow or filled depending on the current draw style. The following code draws a circle with *drawcircle*.

```
createimage 400, 200
drawcircle 10, 10, 389, 189
```

Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Lines

The *drawline* statement draws a line. The parameters are the X-Y coordinates of the top-left and bottom-right endpoints of the line.

```
drawline [#image-ID,] x1, y1, x2, y2
```

The specified line is drawn onto the image in the current draw color. The thickness of the line is controlled by the current draw width. The following code draws two diagonal lines using *drawline*.

```
createimage 400, 200
drawline 0, 0, 399, 199
drawline 0, 199, 399, 0
```

Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Pie Charts

The *drawpie* and *drawpieslice* statements work together to draw a pie chart. The *drawpie* statement draws the outside circle of the pie chart. A *drawpieslice* statement is used to draw each pie slice.

The parameters for *drawpie* are the X-Y coordinates of the top left of the bounding rectangle of the pie, and the radius of a pie slice.

```
drawpie [#image-ID,] x, y, radius
```

The specified pie chart circle is drawn onto the image in the current draw color. The thickness of the line is controlled by the current draw width. The pie chart is hollow or filled depending on the current draw style.

The parameters for *drawpieslice* are a percentage, a draw color, text for the pie slice, and a degree of rotation for the text. Only the percent parameter is required.

```
drawpieslice [#image-ID,] percent [, color [, text [, degree]]]
```

The pie slice is drawn within the pie chart circle (previously created by *drawpie*) with the appropriate size for the specified percentage.

The following code draws a pie chart using *drawpie* and *drawpieslice*.

```
createimage 400, 400
drawpie 0, 0, 200
drawstyle "solid"
drawpieslice "30", "#FF0000", "30%"
drawpieslice "15", "#0000FF", "15%"
drawpieslice "55", "#00FF00", "55%"
saveimage "pie1.bmp"
closeimage
opendoc "pie1.bmp"
```

Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Points

The *drawpoint* statement draws a point. The parameters are the X-Y coordinates of the point.

```
drawpoint [#image-ID,] x, y
```

The specified point is drawn onto the image in the current draw color. The following code draws a rectangle with *drawrectangle*.

```
createimage 400, 200
drawpoint 200, 100
```

Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Squares and Rectangles

The *drawrectangle* statement draws a square or rectangle. The parameters are the X-Y coordinates of the top-left and bottom-right corners of the rectangle.

```
drawrectangle [#image-ID,] x1, y1, x2, y2
```

The specified rectangle is drawn onto the image in the current draw color. The thickness of the line is controlled by the current draw width. The rectangle is hollow or filled depending on the current draw style. The following code draws a rectangle with *drawrectangle*.

```
createimage 400, 200
drawrectangle 10, 10, 389, 189
```

Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Text

The *drawtext* statement draws text with left-justified alignment. The parameters are the X-Y coordinates of the starting point of the text, the text to render, and an optional degree of rotation for the text.

```
drawtext [#image-ID,] x, y, text [, rotation]
```

The specified text is drawn left-aligned starting at the specified point. The current font, draw color, and background color affect the appearance of the text. The following code draws some text with *drawtext*.

```
createimage 400, 200
drawtext 0,0, "top left corner"
saveimage "text.bmp"
closeimage
```

To draw text with centered or right-justified alignment, use *drawtextcenter* or *drawtextright*. Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Centered Text

The *drawtextcenter* statement draws text with centered alignment. The parameters are the X-Y coordinates of the mid-point of the text, the text to render, and an optional degree of rotation for the text.

```
drawtextcenter [#image-ID,] x, y, text [, rotation]
```

The specified text is drawn horizontally centered at the specified point. The current font, draw color, and background color affect the appearance of the text. The following code draws some text with *drawtextcenter*.

```
createimage 400, 200
drawtextcenter 200,100, "centered text"
saveimage "text.bmp"
closeimage
```

To draw text with left-justified or right-justified alignment, use *drawtext* or *drawtextright*. Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Drawing Right-Justified Text

The *drawtextright* statement draws text with right-justified alignment. The parameters are the X-Y coordinates of the ending point of the text, the text to render, and an optional degree of rotation for the text.

```
drawtextright [#image-ID,] x, y, text [, rotation]
```

The specified text is drawn right-aligned to the specified point. The current font, draw color, and background color affect the appearance of the text. The following code draws some text with *drawtextright*.

```
createimage 400, 200
drawtextright 399,0, "top right corner"
saveimage "text.bmp"
closeimage
```

To draw text with left-justified or centered alignment, use *drawtext* or *drawtextcenter*. Once you are finished with all drawing operations, use *saveimage* to save the image to disk.

Setting Colors

The *backcolor* and *drawcolor* statements set the background and draw colors. The background color applies to drawing text. The draw color applies to all drawing statements. Both take one parameter, a color code of the form *#rrggbb*, which includes two digits for each hexadecimal value of red, green, or blue.

```
backcolor [#image-ID,] color
drawcolor [#image-ID,] color
```

The following code sets the draw color in order to draw several different color lines.

```
createimage 400, 200
drawcolor "#FF0000"
drawline 0, 0, 399, 199
drawcolor "#0000FF"
drawline 0, 199, 399, 0
saveimage "colors.bmp"
closeimage
```

When you first begin an image session, the background color defaults to white (*#FFFFFF*) and the draw color defaults to black (*#000000*).

Setting Fonts

The *drawfont* statement sets the font family, size, and style used in drawing text. The parameters are a typeface name, a font size in points, and a string of style flags.

```
drawfont [#image-ID,] state, size, style
```

The style parameter is a string that may contain a combination of the following flags: *P* for plain text, *B* for boldface type, *I* for italicized type, and *U* for underline type. The following code sets the font, then draws text in that font.

```
createimage 400, 200
drawfont "Times New Roman", 18, "BI"
drawtext 0,0, "Invitation"
saveimage "invite.bmp"
closeimage
opendoc "invite.bmp"
```

Setting Draw Width

The *drawwidth* statement sets the width of the pen for drawing lines and shapes. The parameter is a line width in pixels.

```
drawwidth [#image-ID,] width
```

The following code sets the draw width, then draws a rectangle.

```
createimage 400, 200
drawwidth 3
drawrectangle 50, 50, 350, 150
saveimage "rect.bmp"
closeimage
```

Setting Drawing Style

The *drawstyle* statement sets the drawing style for shapes. The parameter is the word "hollow" or "solid".

```
drawstyle [#image-ID,] style
```

The following code sets the draw style, then draws a filled rectangle.

```
createimage 400, 200
drawstyle "solid"
drawrectangle 50, 50, 350, 150
saveimage "rect.bmp"
closeimage
```

Copying an Image

When you have more than one image session open, you can copy some or all of the first image onto the second image. The *imagecopy* statement performs an image copy between two image sessions. The parameters specify the image source and the destination. On the source side, the first five parameters are an image-ID, a top-left corner X-Y coordinate, a width, and a height. The next three parameters specify the destination image's image ID and a target X-Y coordinate.

```
imagecopy   #image-ID1, x1, y1, width, height, #image-ID2, x2, y2
```

The specified area of the source image is copied into the destination image at the specified coordinate. The following code copies the top-left quadrant of a large image onto a smaller image.

```
openimage #img1, "large200x200.jpg"
openimage #img2, "small50x50.jpg"
imagecopy #img1, 0, 0, 50, 50, #img2, 0, 0
saveimage "rect.bmp"
closeimage
```

Acquiring Images from TWAIN Devices

Network Query Language can acquire images from image-oriented devices that support the TWAIN protocol, such as digital cameras, scanners, and Webcams.

The NQL statements for TWAIN operations are listed in Table 21.3.

There are function versions of some of the TWAIN statements. The TWAIN functions are shown in Table 21.4.

Table 21.3 TWAIN Statements

STATEMENT	PARAMETERS	DESCRIPTION
closetwain		Closes a TWAIN device
colordepth	[#*image-ID*,] *depth*	Sets the color depth for a TWAIN device
opentwain	[*twain-source-name*]	Opens a TWAIN device
scan	[*filespec*]	Acquires and stores an image from a TWAIN device
twaingetimage	[#*image-ID*,] [*display-interface*]	Acquires an image from a TWAIN device
twaingetsources		Returns a list of available TWAIN devices
twainshowsources	[*var*]	Displays TWAIN sources and gets a user selection

Table 21.4 TWAIN Functions

FUNCTION	DESCRIPTION
boolean colordepth([#*image-ID*,]	Sets the color depth for a TWAIN device *depth*)
boolean opentwain([*twain-source-name*])	Opens a TWAIN device
boolean scan([*filespec*])	Acquires and stores an image from a TWAIN device
boolean twaingetimage([#*image-ID*,] [*display-interface*])	Acquires an image from a TWAIN device
boolean twaingetsources()	Returns a list of available TWAIN devices
boolean twainshowsources([*var*])	Displays TWAIN sources and gets a user selection

The TWAIN statements and functions are described individually in the following sections.

TWAIN Sessions

The NQL TWAIN operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. A TWAIN session is opened with *opentwain*.
2. Images are acquired using *scan*, or using combinations of the other TWAIN statements.
3. The TWAIN session is closed with *closetwain*.

Opening a TWAIN Session

The *opentwain* statement or function opens a TWAIN session. The parameter is the name of the TWAIN source to open, which is optional. If omitted, the last referenced TWAIN device is used.

```
boolean opentwain([twain-source-name])
opentwain [twain-source-name]
```

If the TWAIN device is accessible, a success condition results (the function returns true). The following code uses *opentwain* to open a TWAIN session.

```
if opentwain()
{
    ...work with TWAIN device...
}
```

Once a TWAIN session is open, you have the option of acquiring and storing images with the all-in-one *scan* statement, or through sequences of the other, lower-level TWAIN statements.

Closing a TWAIN Session

The *closetwain* statement closes a TWAIN session. There are no parameters.

```
closetwain
```

The TWAIN device is closed. The following code shows the use of *closetwain* to close a TWAIN session.

```
if opentwain()
{
    ...work with TWAIN device...
    closetwain
}
```

If you do not explicitly close a TWAIN session, it is closed automatically when the script terminates.

Scanning an Image

The *scan* statement scans an image and stores it. The parameter is an optional file name of type .bmp, .gif, or .jpg. If no file name is specified, an image session is created in memory, and the graphics image statements (described earlier in this chapter) may be used to manipulate and store the image.

```
scan [filespec]
boolean scan([filespec])
```

The image is acquired and stored, either to disk or to an in-memory image session, depending on parameters. If no errors occur, a success condition results (the function returns true). The following code shows the use of *scan* to acquire and save images.

```
if opentwain()
{
    scan "picture.jpg"
    closetwain
}
```

The sequence *scan* (without parameters) is identical to the following sequence:

```
twaingetimage : colordepth 24
```

The sequence *scan* (with a file name parameter) is identical to the following sequence:

```
twaingetimage : colordepth 24 : saveimage filespec : closeimage
```

Acquiring an Image

The *twaingetimage* statement acquires an image from the current TWAIN device and creates an image session in memory. The parameters are an optional image ID (if you have multiple image sessions open at the same time) and a display interface flag, which defaults to false. If set to true, the TWAIN device's drive may display a dialogue during image capture.

```
twaingetimage [#image-ID,] [display-interface]
boolean twaingetimage([#image-ID,] [display-interface])
```

The image is acquired and stored in-memory in an image session. If no errors occur, a success condition results (the function returns true). The following code shows the use of *twaingetimage* to acquire an image.

```
if opentwain()
{
    twaingetimage
    colordepth 8
    saveimage "pic1.bmp"
    closetwain
}
```

For simple scan-and-store applications, the *scan* statement is more automatic.

Obtaining a List of TWAIN Sources

The *twaingetsources* statement or function retrieves a list of TWAIN devices. There are no parameters.

```
twaingetsources
boolean twaingetsources()
```

The result is a string value pushed onto the stack, which is a list of TWAIN sources separated by newlines. If the TWAIN device list is returned, a success condition results (the function returns true). The following code shows the use of *twaingetsources*.

```
twaingetsources
while nextline(source)
{
    show source
}
```

User Selection of a TWAIN Source

The *twainshowsources* statement or function displays a list of TWAIN devices to the user and allows one to be selected. The parameter is an optional variable to receive the name of the selected TWAIN source.

```
twaingetsources [var]
boolean twaingetsources([var])
```

If the user selects a TWAIN source, a success condition results (the function returns true). If the user cancels the dialogue, a failure condition results (the function returns false). The following code shows the use of *twainshowsources*.

```
if twainshowsources(sSource)
{
    opentwain(sSource)
    scan "image1.jpg"
    closetwain
}
```

Tutorial: Counter

Now it is time to create an NQL script that uses the graphics statements. In this tutorial, you will create a program that creates an odometer-style counter Web graphic, as shown in Figure 21.1. Each time the script is run, the number will increment by one.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Counter script.
3. Run the Counter script.
4. Understand the Counter script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Create an image in memory.
- Draw lines and rectangles.
- Render text.
- Save an image as a JPEG file.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you're ready to enter the tutorial script.

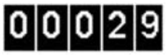


Figure 21.1 Counter graphic.

Step 2: Enter the Counter Script

In the NQL development environment, enter the script shown in Listing 21.1 and save it under the name counter.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//Counter - generates an odometer-style counter graphic

int nLastNumber
string sNumber

//read last counter value from file

open "counter.dat"
if !read(nLastNumber) { nLastNumber = 0 }
close

//increment counter and write udpate value to file

nLastNumber += 1

create "counter.dat"
write nLastNumber
close

//pad number with leading zeroes to get to 5 digits

sNumber = nLastNumber
sNumber = right("00000" | sNumber, 5)

//create the image

createimage 80, 25
drawstyle "solid"
drawcolor "#000000"
drawrectangle 0, 0, 79, 24

drawcolor "#ffffff"
drawstyle "hollow"
drawrectangle 0, 0, 79, 24

drawline 16, 0, 16, 24
drawline 32, 0, 32, 24
drawline 48, 0, 48, 24
drawline 64, 0, 64, 24
```

Listing 21.1 Counter script.

```
//render the digits

backcolor "#000000"
drawfont "arial", 14, ""

for d = 1, 5
{
    c = mid(sNumber, d, 1)
    drawtextcenter ((d-1)*16)+8, 2, c
}

saveimage "counter.jpg"
closeimage

opendoc "counter.jpg"
```

Listing 21.1 Counter script (continued).

If you prefer, you may copy counter.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select counter.nql from the Tutorials\ch21 folder.

At this point, the script in Listing 21.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 3: Run the Counter Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. If all goes well, a few seconds will pass and an odometer image will open up on your desktop. Run the script a second time, and the number in the odometer will have increased by one. If this does not happen, check the following:

- Make sure you have entered the script correctly.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Counter script run, dynamically generating a different image each time it is run. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the Counter Script

We now want to make sure we understand every part of the Counter script. The first line is a comment line.

```
//Counter - generates an odometer-style counter graphic
```

Two variables are declared, *nLastNumber* and *sNumber*. The number last shown on the odometer will be read into the integer variable *nLastNumber*, incremented, and written back to disk for next time. The string variable *sNumber* will hold a copy of the odometer number padded with leading zeroes so that its length is five digits.

```
int nLastNumber
string sNumber
```

The first order of business is to find out what the last number displayed on the odometer was. The script stores this number in a data file named counter.dat. The *open* statement opens counter.dat (or creates an empty file if it does not exist). The *read* statement reads a number from this file into *nLastNumber*. If it fails, this is the first time the script is being run and *nLastNumber* is set to 0. The file is then closed with *close*.

```
//read last counter value from file

open "counter.dat"
if !read(nLastNumber) { nLastNumber = 0 }
close
```

The *nLastNumber* variable is incremented by one, which is the number that we will display in the graphic image we are about to generate.

```
//increment counter and write update value to file

nLastNumber += 1
```

The new value needs to be written to the counter.dat file so that the next time the script runs, the number will advance. The file is created new with a *create* statement. The value in nLastNumber is written with a *write* statement. The file is closed.

```
create "counter.dat"
write nLastNumber
close
```

Next, we need a five-character copy of the number to display: The odometer image we will create has five digits, so we need leading zeroes to get the length right. The value in *nLastNumber* is copied to the string variable *sNumber*. The right function ensures the *sNumber* has leading zeroes to fill it to a length of five.

```
//pad number with leading zeroes to get to 5 digits

sNumber = nLastNumber
sNumber = right("00000" | sNumber, 5)
```

The image can now be created. The first step is to create an image 80 pixels wide by 25 high using *createimage*. The background of the image is a black rectangle. To draw it, the drawing style is set to solid with *drawstyle*, the draw color is set to *000000* (black) with *drawcolor*, and a solid black rectangle is drawn with *drawrectangle*.

```
//create the image

createimage 80, 25
drawstyle "solid"
drawcolor "#000000"
drawrectangle 0, 0, 79, 24
```

The rest of our drawing will be in white, so we set the draw color to *ffffff* (white) with *drawcolor*. We now want to draw a white frame around the outside of the solid black rectangle that was just drawn. The *drawstyle* statement sets the hollow drawing mode, and another *drawrectangle* statement draws a white rectangle.

```
drawcolor "#ffffff"
drawstyle "hollow"
drawrectangle 0, 0, 79, 24
```

We need four vertical white lines to make the image resemble an odometer with five individual digits. The image is 80 pixels across, so the lines need to be drawn at X coordinates of 16, 32, 48, and 64. The Y coordinates for the lines will be from 0 to 24, the full height of the image. The lines are drawn with the *drawline* statement.

```
drawline 16, 0, 16, 24
drawline 32, 0, 32, 24
drawline 48, 0, 48, 24
drawline 64, 0, 64, 24
```

We now have a convincing-looking odometer, except that there are no digits. We have to render the digits as text to complete the image. The text foreground color is already white, because it is the same as the drawing color. The text background color needs to be set to black, which is handled by the *backcolor* statement. The desired font is set to Arial 14 with the *drawfont* statement.

```
//render the digits

backcolor "#000000"
drawfont "arial", 14, ""
```

A loop from one to five draws the digits. The digit character to display is extracted into the variable *c* by the *mid* function. The *drawtextcenter* statement draws each digit, centered on the unit space for that digit on the image. The digit's location is computed by taking the current digit (*d*), subtracting one, and multiplying by 16. This gives us 0 for the leftmost digit, 16 for the next, 32 for the next, and so on. We then add 8, half the unit width, because we are using *drawtextcenter*. The Y coordinate is not zero because that would overwrite part of the border. A value of 2 drops the digits slightly below the border.

```
for d = 1, 5
{
    c = mid(sNumber, d, 1)
    drawtextcenter ((d-1)*16)+8, 2, c
}
```

The image is complete, but it does not exist on disk yet. The *saveimage* statement saves the image as a JPEG file named counter.jpg. The image session is no longer needed, and is terminated with *closeimage*.

```
saveimage "counter.jpg"
closeimage
```

The image is opened on the desktop with *opendoc*. The default application for displaying JPEG images will be launched, and counter.doc will be opened in it.

```
opendoc "counter.jpg"
```

At this point, you've witnessed the generation of dynamic Web graphics in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Alter the Y coordinate of one of the digits for *drawtextcenter*, to give the illusion of a real odometer with a digit that hasn't fully rolled into place yet.
- Change the file to use for counting to the registry, so that the directory the script is run from is immaterial in computing the number to display.
- Create an NQL script that runs as a Web application, generating and displaying an updated counter each time the page is accessed.

Chapter Summary

Network Query Language can create or modify images. Graphics in GIF, JPEG, and bitmap formats can be generated.

- New images are created with the createimage statement.
- Existing images are opened for modification with the openimage statement.
- Images are saved to disk with the saveimage statement.
- Image sessions are closed with the *closeimage* statement.

Drawing operations allow various shapes to be drawn.

- Circles and ellipses are drawn with the drawcircle statement.
- Lines are drawn with the drawline statement.
- Pie charts are drawn with drawpie and drawpieslice statements.
- Points are drawn with the drawpoint statement.
- Rectangles and squares are drawn with the drawrectangle statement.
- The drawing color is set with the drawcolor statement.
- The drawing width is set with the drawwidth statement.
- The drawing style, solid or hollow, is set with the drawstyle statement.
- Images or portions of images can be copied from one image to another with the *imagecopy* statement.

Text can be rendered onto images.

- The drawtext statement draws left-aligned text.
- The drawtextcenter statement draws centered text.
- The drawtextright statement draws right-aligned text.
- The backcolor statement sets the text background color. The text foreground color is determined by the drawing color.
- The *drawfont* statement specifies a font for text rendering.

NQL can interface with TWAIN devices, such as digital cameras and scanners. Images captured from TWAIN devices can be operated on by the graphic statements.

- The *opentwain* statement opens a TWAIN session for a device. If a specific device is not specified, it defaults to the last TWAIN device used.
- The closetwain statement closes a TWAIN session.
- An image can be scanned with the scan statement. One form of scan handles all details of device location, scanning an image and saving it to a file in a single statement.

For more control, there are TWAIN statements that operate at a lower level than the *scan* statement:

- The *colordepth* statement sets the color depth for a TWAIN image capture.
- The *twaingetimage* statement acquires an image from a TWAIN device and opens an image session.
- The *twaingetsources* statement returns a list of the available TWAIN devices.
- The *twainshowsources* statement displays TWAIN sources and allows the user to select one.

NQL allows your connected applications to generate graphics dynamically, create images and charts of real-time information, and capture images from digital devices.

CHAPTER 22

Agent Characters

On-screen characters give your scripts a personable way to present information to users on the desktop. Network Query Language has direct support for Microsoft Agent, which provides desktop animated characters that speak and move. This chapter describes how to control agent characters from NQL scripts. The tutorial at the end of the chapter enacts some Shakespeare with three agent characters.

Microsoft Agent

Microsoft Agent is an interface for displaying agent characters and having them speak, move, and take actions. It is available free from Microsoft, and can be downloaded from http://msdn.microsoft.com/msagent.

Figure 22.1 shows Merlin, one of the characters that comes with Microsoft Agent. Several other characters come with Microsoft Agent, and you can add your own. Many third-party characters for Microsoft Agent can be found on the Internet.

Characters can speak dialogue balloons visibly on the desktop, or they can speak audibly if your PC has a text-to-speech engine installed. Agent characters have movements, such as waving or bowing. All of this can easily be put under your control when programming in NQL.

Figure 22.1 Merlin character.

Agent Operations

The NQL statements for agent character operations are listed in Table 22.1.

There are function versions of some of the agent character statements. The agent character functions are shown in Table 22.2.

The agent character statements and functions are described individually in the following sections.

Agent Character Sessions

The NQL agent character operations are modeled on the idea of a session, in which the statements and functions must be used in a certain order.

1. An agent character session is opened with *openagent*.
2. Any desired series of agent operations is performed using the other agent statements.
3. The agent character session is closed with *closeagent*.

Multiple Agent Character Sessions

NQL uses a very simple syntax for agent operations, as shown in the following line:

```
showagent "Hello, you have new mail"
```

Table 22.1 Agent Character Statements

STATEMENT	PARAMETERS	DESCRIPTION
closeagent	[#*agent-ID*]	Closes an agent session
hideagent	[#*agent-ID*]	Makes an agent character invisible on the desktop
listagentactions	[#*agent-ID*]	Returns a list of available agent actions
listagents		Returns a list of available agent characters
moveagent	[#*agent-ID,*] *x*, *y*	Moves an agent character to a different position on the desktop
openagent	[#*agent-ID,*] *character-name* [, *visible*]	Opens an agent session
playagent	[#*agent-ID,*] *action*	Causes an agent character to play an action on the desktop
pointagent	[#*agent-ID,*] *x*, *y*	Causes an agent character to gesture toward a position on the desktop
showagent	[#*agent-ID,*] [*x*, *y*] [*message*]	Makes an agent character visible, moves it to a position, and speaks a message

Table 22.2 Agent Character Functions

FUNCTION	DESCRIPTION
boolean listagentactions([#*agent-ID*])	Returns a list of available agent actions
boolean listagents()	Returns a list of available agent characters
boolean openagent([#*agent-ID,*] *character-name* [, *visible*])	Opens an agent session

When multiple agents are needed on the desktop simultaneously, identifiers distinguish one agent character session from another. In NQL, these identifiers are known as agent session IDs. An agent session ID is a name preceded by a pound sign, such as *#merlin* or *#StockBot*. When a session ID is specified, it is the first parameter of a statement or function. For example,

```
showagent #peedy, "Hello, you have new mail"
```

Opening an Agent Character Session

The *openagent* statement or function begins an agent character session. The parameters are the name of the character to use and an optional visible flag, which defaults to true.

Some of the agent characters that come with Microsoft Agent are Merlin, Peedy, Robby, and Genie, but the system also allows custom characters to be defined.

```
openagent [#agent-ID,] character-name [, visible]
boolean openagent([#agent-ID,] character-name [, visible])
```

If the agent character is successfully located, it displays on the desktop (unless the visible parameter is set to false) and a success condition results (the function returns true). The following code opens an agent character with *openagent*.

```
openagent "merlin"
```

Once an agent character has been opened, it may be positioned, made visible or invisible, made to gesture, and made to speak.

Positioning an Agent Character

The *moveagent* statement moves an agent to a specific location on the desktop. The parameters are the X-Y coordinates to move to.

```
moveagent [#agent-ID,] x, y
```

The agent moves to the specified position. It is valid to reposition an agent with *moveagent* whether or not it is visible. The following code opens an agent character and repositions it to the top left corner of the screen with *moveagent*.

```
openagent "peedy"
moveagent 0, 0
```

Making an Agent Character Speak

The *showagent* statement can be used to speak a message. The message is displayed in a dialogue balloon and, if the computer supports it, is also rendered as speech. The parameters are optional X-Y coordinates and message text.

```
showagent [#agent-ID,] [x, y] [message]
```

The agent is made visible, is repositioned (if coordinates were specified), and speaks the specified message. The following code uses *showagent* to make an agent character speak.

```
openagent "robby"
showagent "Time for your coffee break"
```

```
wait 10
showagent "Time for lunch"
closeagent
```

Playing an Agent Action

An agent character comes with actions that can be played. The *playagent* statement plays an action. The parameter is an action word, such as "bow." The available actions can be determined with the *listagentactions* statement.

```
playagent     [#agent-ID,] action
```

The agent plays the specified action. The following code makes an agent wave.

```
playagent "wave"
```

Making an Agent Gesture

Agent characters can gesture toward a point on the screen. The *pointagent* statement gestures to a specific location. The parameters are an X, Y coordinate on the screen.

```
pointagent     [#agent-ID,] X, Y
```

The agent gestures toward the specified coordinate in an animation that varies from one character to the next. The following code makes an agent gesture toward the upper-left-hand corner of the screen.

```
pointagent 0, 0
```

Closing an Agent Session

The *closeagent* statement closes an agent session.

```
closeagent   [#agent-ID]
```

If a script ends, any open agent sessions are closed automatically.

Showing and Hiding Agents

Agent characters can be visible or invisible at times under your direct control. The *showagent* statement makes an agent visible. The simplest form of *showagent* has no parameters and makes the agent character visible on the desktop.

```
showagent
```

The converse of *showagent* is *hideagent*, which makes an agent character invisible.

```
hideagent
```

Displaying Messages

The *showagent* statement serves a number of purposes, one of which is displaying messages. The messages will be shown in a dialog balloon and, if a speech engine is installed, spoken audibly. The parameters are optional X-Y coordinates and an optional message.

```
showagent [#agent-ID,] [x, y] [message]
```

The *showagent* statement can perform several functions. It always makes the agent character visible. If X-Y coordinates are specified, it causes the agent to reposition (like *moveagent*). If a message is specified, the agent speaks the message. The following code positions an agent character at a point on the screen, where it then speaks a message.

```
showagent 300, 100, "The backup is complete".
```

Tutorial: Hamlet

Now that you are well-versed in agent characters, let's write a script that uses them. In this tutorial, you will create a script that launches a total of three agent characters and has them enact a brief bit of Shakespeare's "Hamlet." The characters will speak parts as well as make movements.

Figure 22.2 shows one of the characters speaking a part. Three standard characters that come with Microsoft Agent will be used: Robby the robot will play Polonius, Merlin the magician will play King Claudius, and Peedy the parrot will play Hamlet. In order to run this tutorial, you will need Microsoft Agent installed.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Hamlet script.
3. Run the Hamlet script.
4. Understand the Hamlet script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

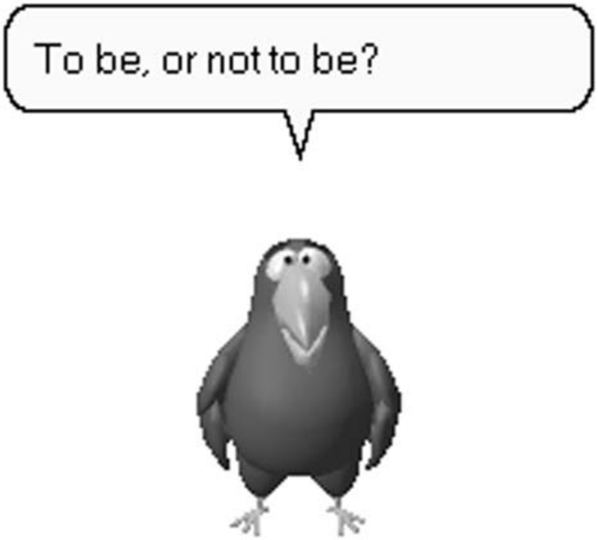


Figure 22.2 Peedy character playing Hamlet.

- Display agent characters.
- Reposition agent characters.
- Display agent character messages.
- Play agent character actions and gestures.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the Hamlet Script

In the NQL development environment, enter the script shown in Listing 22.1 and save it under the name Hamlet.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Hamlet.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Hamlet.nql from the Tutorials\ch22 folder.

At this point, the script in Listing 22.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD.

Step 3: Run the Hamlet Script

You are now ready to run the script (and to be entertained). You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. Merlin (King Claudius) and Polonius (Robby) appear on the screen and interact, move, and speak dialog. Peedy (Hamlet) follows with his own movements and dialog. If this does not happen, check the following:

- Check your script for typographical errors.
- Ensure that Microsoft Agent is installed on your system, and that the Robby, Merlin, and Peedy characters have been installed.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Hamlet script run, controlling multiple agent characters. In the next step, we will dissect the script and explain exactly how it works.

Step 4: Understand the Hamlet Script

We now want to make sure we understand every part of the Hamlet script. The first few lines are comments.

```
//Hamlet - demonstrates use of agent characters to enact a bit of
Shakespeare.

//Note that you need to have Microsoft Agent and the characters used in
//the script - you can get these at http://msdn.microsoft.com/msagent/.
//Be sure to install a speech engine as well.

//Dramatis personae - characters for this short take from Hamlet:
//
//    Polonius = Robby
//    King Claudius = Merlin
//    Hamlet = Peedy
```

The first two characters are opened using *openagent*. If either one fails, the user is told the character is not present, and the script ends.

```
//since we have multiple characters we use session identifiers
//for each instance of an Agent character.  We supply the false
//argument so they will become visible exactly where we want them.
```

```
//Hamlet - demonstrates use of agent characters to enact a bit of
Shakespeare.

//Note that you need to have Microsoft Agent and the characters used
in the
//script - you can get these at http://msdn.microsoft.com/msagent/.
Be sure
//to install a speech engine as well.

//Dramatis personae - characters for this short take from Hamlet:
//
//    Polonius = Robby
//    King Claudius = Merlin
//    Hamlet = Peedy

//since we have multiple characters we use session identifiers
//for each instance of an Agent character.  We supply the false
//argument so they will become visible exactly where we want them.

if !openagent (#King, "Merlin", false)
{
    show "You need to have the Microsoft Merlin Agent character
installed"
    end
}

if !openagent (#Polonius, "Robby", false)
{
    show "You need to have the Microsoft Robby Agent character
installed"
    end
}


//move to the starting location and then make them appear

moveagent #King, 300,300
moveagent #Polonius, 200,300
showagent #King, 300,300
showagent #Polonius, 200,300

//they are fine actors.... use some body movements

playagent #Polonius, "gestureleft"
playagent #King, "lookleft"
```

continues

Listing 22.1 Hamlet script.

```
//deliver the dialog

showagent #Polonius, "I hear him coming: let’s withdraw, my lord."

//exit stage left

closeagent #Polonius
closeagent #King

//pause a second, for dramatic purposes

sleep 1
if !openagent (#Hamlet, "Peedy", false)
{
    show "You need to have the Microsoft Peedy Agent character
installed"
    end
}

moveagent #Hamlet, 300,400

//Now Peedy can deliver a well known dialog

showagent #Hamlet, "To be, or not to be?"
playagent #Hamlet, "think"
showagent #Hamlet, "That is the question."
playagent #Hamlet, "gestureup"
showagent #Hamlet, "Whether 'tis nobler in the mind to suffer"
showagent #Hamlet, "The slings and arrows of outrageous fortune,"
showagent #Hamlet, "Or to take arms against a sea of troubles,"
showagent #Hamlet, "And by opposing end them?"
playagent #Hamlet, "think"
showagent #Hamlet, "To die,"
showagent #Hamlet, "to sleep..."

//time to take a bow!

sleep 2
playagent #Hamlet, "lookdownleft"
playagent #Hamlet, "lookdownright"
playagent #Hamlet, "lookdownleft"
playagent #Hamlet, "lookdownright"
showagent #Hamlet, "that’s all folks!"
closeagent #Hamlet

end
```

Listing 22.1 Hamlet script (continued).

```
if !openagent (#King, "Merlin", false)
{
    show "You need to have the Microsoft Merlin Agent character
installed"
    end
}

if !openagent (#Polonius, "Robby", false)
{
    show "You need to have the Microsoft Robby Agent character
installed"
    end
}
```

The characters are now positioned with *moveagent* and made visible with *showagent*.

```
//move to the starting location and then make them appear

moveagent #King, 300,300
moveagent #Polonius, 200,300
showagent #King, 300,300
showagent #Polonius, 200,300
```

Polonius is made to play a *gestureleft* action and King Claudius a *lookleft* action with the *playagent* statement.

```
//they are fine actors.... use some body movements

playagent #Polonius, "gestureleft"
playagent #King, "lookleft"
```

The *showagent* statement speaks a part.

```
//deliver the dialog

showagent #Polonius, "I hear him coming: let's withdraw, my lord."
```

Both characters are closed, and they vanish from the screen. This is the end of Act I for our script. In the next act, Hamlet (Peedy) will enter.

```
//exit stage left

closeagent #Polonius
closeagent #King
```

After a one-second delay, a third agent character is opened with *openagent*.

```
//pause a second, for dramatic purposes

sleep 1
if !openagent (#Hamlet, "Peedy", false)
{
    show "You need to have the Microsoft Peedy Agent character
installed"
    end
}
```

Hamlet is moved to a point on the screen with *moveagent*, then made to speak dialog and perform actions repeatedly with *playagent* and *showagent*.

```
moveagent #Hamlet, 300,400

//Now Peedy can deliver a well known dialog

showagent #Hamlet, "To be, or not to be?"
playagent #Hamlet, "think"
showagent #Hamlet, "That is the question."
playagent #Hamlet, "gestureup"
showagent #Hamlet, "Whether 'tis nobler in the mind to suffer"
showagent #Hamlet, "The slings and arrows of outrageous fortune,"
showagent #Hamlet, "Or to take arms against a sea of troubles,"
showagent #Hamlet, "And by opposing end them?"
playagent #Hamlet, "think"
showagent #Hamlet, "To die,"
showagent #Hamlet, "to sleep..."

//time to take a bow!

sleep 2
playagent #Hamlet, "lookdownleft"
playagent #Hamlet, "lookdownright"
playagent #Hamlet, "lookdownleft"
playagent #Hamlet, "lookdownright"
showagent #Hamlet, "that's all folks!"
```

The show is over. The Peedy agent character is closed, and vanishes from the stage. The script ends.

```
closeagent #Hamlet

end
```

At this point, you've seen an NQL script control the locations, speech, and actions of multiple agent characters.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Use different Microsoft Agent characters for the three parts.
- Use a different set of speech and actions for the characters.
- Adapt the script to present useful information from your enterprise or the Internet, such as a stock price or the current outside temperature.

Chapter Summary

Agent characters allow your scripts to control desktop emissaries who can move and speak. To use agent characters in Network Query Language requires the presence of Microsoft Agent software.

- The *listagents* statement lists the available agent characters.
- The *openagent* statement opens an agent session.
- The *showagent* statement makes an agent visible, and can also position an agent or speak a message.
- The *hideagent* statement makes an agent invisible.
- The *moveagent* statement positions an agent.
- The *listagentactions* statement lists the actions an agent can perform.
- The *playagent* statement plays an agent action.
- The *pointagent* statement causes an agent to gesture toward a point on the screen.
- The *closeagent* statement closes an agent session.

Agent characters are a useful form of user interface for connected applications that reside on the desktop.

CHAPTER 23

Fuzzy Logic

Traditional programming relies on the concepts of true and false. The binary nature of computers has nurtured this strong reliance on two-state logic and a black-and-white view of information. Fuzzy logic, in contrast, recognizes degrees of belief and can be used to work with gray areas, partial information, and concepts such as *almost*. Fuzzy logic allows software to make better decisions, even if there are complex combinations of pros and cons that must be taken into account. Fuzzy logic is also a useful tool in representing the weight (relative importance) of rules.

Network Query Language provides language facilities for expressing fuzzy conditions and working with fuzzy values. This chapter begins with background information on fuzzy logic, then goes on to describe how it is used in NQL. In the tutorial at the end of the chapter you will create a program that uses fuzzy logic to find and rank musical events.

Introduction to Fuzzy Logic

The fundamental concept of fuzzy logic is an extremely simple one: Representing logic in the constraining black-and-white world of ones and zeroes is often inadequate for software applications that deal with the real world. To appreciate this, consider the problems that arise when attempting to represent any of the following scenarios as traditional true/false values:

- Whether or not someone is tall
- Whether or not a vehicle is moving quickly
- Whether or not someone is sick
- Whether or not a carton is full
- Whether or not a price is good
- Whether or not two photographs are similar
- Whether or not someone is trustworthy

Although you *can* use a simple Boolean true/false value to embody these kinds of values, the results will likely be disappointing. Let's consider the first scenario, whether or not someone is tall. If the programmer decides that anyone 6 feet high or taller is *tall*, then a program would consider someone with a height of 6 feet, 4 inches to be tall (true), and someone with a height of 4 feet, 3 inches as *not tall* (false). It is easy to see where this kind of simple representation breaks down: Someone 5 feet, 11 inches is considered *short*. Moreover, that person's tallness is represented identically to someone who is only 3 inches tall! Programs that represent values so imprecisely can't help but make poor judgments.

Fuzzy logic is a far more appropriate way to represent values such as *tallness*. As we shall see, although fuzzy logic has a name that implies imprecision to some, the opposite is true. Fuzzy values provide a way to represent shades of meaning and degrees of belief. Programs that represent values accurately are able to make better decisions.

Fuzzy Values

The key to representing information in fuzzy logic is to allow all of the real numbers between 0 and 1 to be used to represent a true/false value. For example, something could be 0.75 true, which is more true than 0.66 but less true than 0.9. In this system, 0.0 is absolutely false and 1.0 is absolutely true. The number line in Figure 23.1 illustrates how fuzzy values work. A value of 0.60 is somewhat true, in that it is more true than false. A value of 0.85 is far more true. A value of 1.0 is as true as can be.

If we use fuzzy values to represent a characteristic like tallness, we can represent all degrees of tallness in the same 0.0-1.0 scale, as Figure 23.1 shows. To do so, we must

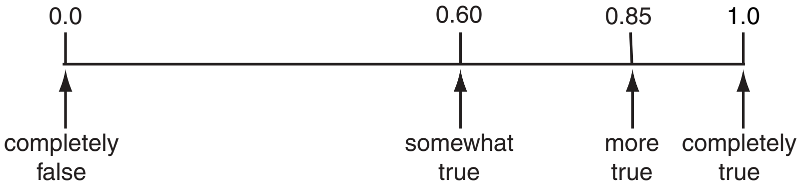


Figure 23.1 Fuzzy values number line.

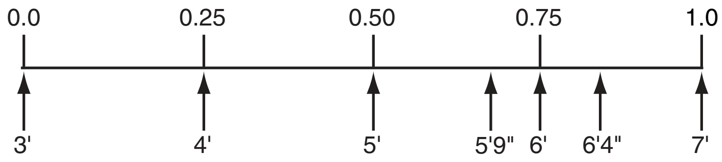


Figure 23.2 Tallness represented on a fuzzy value scale.

choose an upper limit for what 1.0 represents and a lower limit for what 0.0 represents. In Figure 23.2, we use 3 feet as a minimum height and 7 feet as a maximum height.

Fuzzy values, then, allow for an unlimited amount of distinction between the two absolutes of 0.0 (completely false) and 1.0 (completely true). With the freedom of fuzzy values, you can easily represent the following categories of information:

Degrees of a measurable attribute, such as tallness or speed

Degrees of belief, such as "environmental conditions are safe"

Concepts like *nearly*, *mostly*, *almost*, *possibly*, and *hardly*

Ranking on a scale, such as an exam grade or a temperature

Probabilities, such as the likelihood that a region will experience an earthquake

Combining Fuzzy Values

Fuzzy values can be combined into composite fuzzy values. For example, the composite value *largeness* could be a composite of two other values: *height* and *weight*.

There are many different ways to combine fuzzy values into a composite value. The simplest method is to add the individual values, then divide by the number of values so that the result is still within the standard fuzzy logic range of 0.0-1.0. For example, the composite value *should I see a movie tonight* might be derived from the individual values *something interesting is playing tonight*, *I have money*, and *tonight is not a school night*. If the latter three values were 0.7, 1.0, and 1.0, this would yield a value of (0.7+1.0+1.0)/3, or 0.9.

What if you want to combine terms with an unequal weighting? In this case, the method is to again sum each of the values, but first multiplying each value by its weighting factor. The total is divided by the sum of the weights rather than the number of terms. To revisit the movie example, assume we wish the condition *something interesting is playing tonight* to be three times more important than the other two factors. This would be evaluated as ((0.7*3)+(1.0*1)+(1.0*1))/5, or 0.82.

Logical Operations

In traditional Boolean algebra, true and false values can be combined using logical operations such as AND and OR. There are equivalent operations for fuzzy values.

- A fuzzy OR operation takes the maximum of two fuzzy values. Thus, 1.0 OR 0.5 equals 1.0, and 0.1 OR 0.44 equals 0.44.
- A fuzzy AND operation takes the minimum of two fuzzy values. Thus, 0.6 AND 0.4 equals 0.4, and 1.0 AND 0.9 equals 0.9.
- A fuzzy NOT operation is performed by subtracting the value from 1.0. Thus, NOT 1.0 equals 0.0, and NOT 0.20 equals 0.80.

Fuzzy Logic Support in NQL

You can choose to follow a fuzzy logic approach in just about any programming language, since fuzzy logic values can be represented as floating-point numbers. However, NQL has features expressly designed to make it natural and easy to apply fuzzy logic in your programs. NQL includes a fuzzy variable type, fuzzy logic operators, and the ability to specify fuzzy logic functions.

The NQL Fuzzy Logic Statements and Operators

NQL has both statements and operators for fuzzy logic. The NQL fuzzy logic statements are listed in Table 23.1.

The fuzzy operators are listed in Table 23.2.

The fuzzy logic statements and operators are described individually in the following sections.

Table 23.1 Fuzzy Logic Statements

STATEMENT	PARAMETERS	DESCRIPTION
condition	*name var-list*	Declares a fuzzy condition; followed by a code block
evaluate	*name var-list*	Computes a fuzzy value by executing its condition function
fuzzy	*var-list*	Declares one or more variables of type *fuzzy*
is	*result-expression*	Returns a fuzzy result in a condition function
is	*result-expression*, *condition-expression*	Returns a fuzzy result in a condition function if a logical condition is met
otherwise	*result-expression*	Returns a fuzzy result in a condition function if no other result has been specified

Table 23.2 Fuzzy Logic Operators

OPERATOR	DESCRIPTION
FUZZYOR	Performs a fuzzy OR operation on its two terms
^|	Performs a fuzzy OR operation on its two terms
FUZZYAND	Performs a fuzzy AND operation on its two terms
^&	Performs a fuzzy AND operation on its two terms

Fuzzy Variables

One of NQL's variable types is *fuzzy*. A fuzzy variable is similar to a floating-point variable, but enforces the range of 0.0–1.0. If you attempt to assign a negative value to a fuzzy variable, it will be set to 0.0. Likewise, attempting to set a fuzzy variable to a value greater than 1.0 will store 1.0.

Fuzzy variables are declared with the *fuzzy* keyword.

```
fuzzy Weight, Height, IsLarge
Weight = 185/300
Height = 6.4/7.0
IsLarge = Weight FUZZYOR Height
```

Fuzzy values otherwise act like the other numeric types. You can add, subtract, multiply, and divide them. You can intermix them with other numeric types in expressions. At all times, the range of 0.0–1.0 is enforced.

Fuzzy Operators

The FUZZYOR operator performs a fuzzy OR operation on its two terms. An alternate symbol for FUZZYOR is ^|. The following two expressions are functionally identical.

```
IsLarge = Weight FUZZYOR Height
IsLarge = Weight ^| Height
```

In addition, the standard OR function will perform a fuzzy OR if at least one of the two terms is of type fuzzy.

The FUZZYAND operator performs a fuzzy AND operation on its two terms. An alternate symbol for FUZZYAND is ^&. The following two expressions are functionally identical:

```
IsLarge = Weight FUZZYAND Height
IsLarge = Weight ^& Height
```

In addition, the standard AND function will perform a fuzzy AND if at least one of the two terms is of type fuzzy.

The NOT function performs a fuzzy NOT when applied to a fuzzy variable.

```
IsSmall = NOT(IsLarge)
```

Condition Functions

Fuzzy logic values are often created from one or more other values using specialized logic. For example, the fuzzy value *reasonable price* for a given price of a candy bar might be computed using rules like these:

1. If the price is less than 0.25, the result is 0.0 because the price is unacceptable (to the seller).
2. If the price is greater than 0.75, the result is 0.0 because the price is unacceptable (to the buyer).
3. Otherwise, the price is reasonable and the value is computed as *(price-0.25)*2*, which will map any price between 0.25 and 0.75 to the scale of 0.0 to 1.0.

Although you could represent this logic using standard language statements, NQL allows you to express it more naturally through the use of a *condition function*. A condition function resembles a standard function, but employs the *is* and *otherwise* keywords.

The *is* statement sets a fuzzy return value, given that its condition is true. A *result expression* and a *condition expression* are specified. If the condition expression is true, the result expression is the result that will be returned by the function.

```
is result-expression, condition-expression
```

The *otherwise* statement sets a fuzzy return value if no other value has been specified by an *is* statement. It is a catch-all, in case the other logic in the function failed to set a result.

```
otherwise result-expression
```

A sample condition function is shown. Given someone's age, U.S. citizenship, and conviction record, are they eligible to run for President of the United States? An age under 35, lack of citizenship, or record of conviction will make someone ineligible.

```
condition IsEligibleForPresident(float Age, boolean IsCitizen, boolean
IsConvict)
{
    is 0.0, Age < 35
    is 0.0, NOT IsCitizen
    is 0.0, NOT IsConvict
    otherwise 1.0
}
```

The function, which returns a fuzzy value, may be used in an expression.

```
Eligibility = IsEligibleForPresident(ApplicantAge,
ApplicantCitizenship, ApplicationConviction)
```

Another way to call the function is with the *evaluate* statement. This evaluates a condition function and sets a fuzzy variable with the same name as the function.

```
evaluate IsEligibleForPresident(ApplicantAge,
ApplicantCitizenship, ApplicationConviction)
show IsEligibleForPresident
```

Tutorial: Concerts

Now that you are familiar with fuzzy logic, let's put that knowledge to use in a script. In this tutorial, you will create a script that scans an XML document of concert opportunities, scoring each by weighing pros and cons, and recommends the best overall opportunity. The source will be Concerts.xml, a document included with the tutorial script on the companion CD.

The script will be biased to especially prefer Classical Music and to especially avoid Hip Hop, with a neutral stance toward all other genres of music. Another bias will be a venue preference for Irvine, California and a strong preference for Las Vegas, Nevada. Lastly, prices from $1 to $100 will be ranked by price, with a 1.0 score for $0.00 (the best price of all) and a 0.0 score for ticket prices over $100 (too expensive).

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Concerts script.
3. Run the Concerts script.
4. Understand the Concerts script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Work with fuzzy variables.
- Use direct comparisons to set fuzzy values.
- Use condition functions to set fuzzy values.
- Compute combined fuzzy scores from other fuzzy values.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program*

Files, Network Query Language, NQL Client from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the Concerts Script

In the NQL development environment, enter the script shown in Listing 23.1 and save it under the name Concerts.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Concerts.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Concerts.nql from the Tutorials\ch23 folder.

```
//Concerts - ranks concert events using fuzzy logic

fuzzy score, scoreGenre, scoreCity, scorePrice
string BestConcert
fuzzy BestScore

load "concerts.xml"
setinput

//read concerts from XML document

while inputrecord("Concert")
{
    input Show, Performer, Genre, Dates, City, State, Zip,
TicketPriceLow, TicketPriceHigh

    score = 0.0

    scoreGenre = 0.5
    if Genre=="Classical" { scoreGenre = 1.0 }
    if Genre=="Hip Hop" { scoreGenre = 0.0 }

    scoreCity = 0.25
    if Genre=="Irvine" { scoreCity = 0.75 }
    if Genre=="Las Vegas" { scoreCity = 1.0 }

    evaluate scorePrice

    score = (scoreGenre + scoreCity + scorePrice) / 3.0
```

Listing 23.1 Concerts script.

```
    if score > BestScore
    {
        BestScore = score
        BestConcert = Show
    }

    outputrecord "Concert"
    output Show, Performer, City, score
}
show "The best concert opportunity is\n\n{BestConcert}, with a score
of {BestScore}"
end

condition scorePrice
{
    is 0.0, TicketPriceLow > 100.00
    is 1.0, TicketPriceLow == 0.00
    otherwise TicketPriceLow/100.00
}
```

Listing 23.1 Concerts script (continued).

You will need to create or copy Concerts.xml, which is the data source for the script. Concerts.xml is shown in Listing 23.2. The Tutorials\ch23 folder contains this file, which needs to be in the same directory as your script.

```
<?xml version="1.0"?>
<Concerts>

  <Concert>
    <Show>Chamber Music</Show>
    <Performer>Yo Yo Ma</Performer>
    <Genre>Classical</Genre>
    <Dates>10/01/02 through 11/15/02</Dates>
    <City>Boston</City>
    <State>MA</State>
    <Zip>142024</Zip>
    <TicketPriceLow>$35.00</TicketPriceLow>
    <TicketPriceHigh>$75.00</TicketPriceHigh>
  </Concert>

  <Concert>
```

continues

Listing 23.2 Concerts.xml file.

```
      <Show>Hip Happenings</Show>
      <Performer>Thumper</Performer>
      <Genre>Hip Hop</Genre>
      <Dates>10/13/02 through 10/16/02</Dates>
      <City>Memphis</City>
      <State>TN</State>
      <Zip>35442</Zip>
      <TicketPriceLow>$27.00</TicketPriceLow>
      <TicketPriceHigh>$60.00</TicketPriceHigh>
    </Concert>

    <Concert>
      <Show>Goin' Home</Show>
      <Performer>The Rocking Chair Gang</Performer>
      <Genre>Country</Genre>
      <Dates>10/25/02 through 11/15/02</Dates>
      <City>Nashville</City>
      <State>TN</State>
      <Zip>37617</Zip>
      <TicketPriceLow>$30.00</TicketPriceLow>
      <TicketPriceHigh>$109.00</TicketPriceHigh>
    </Concert>

    <Concert>
      <Show>Orchestrations</Show>
      <Performer>Pacific Symphony Orchestra</Performer>
      <Genre>Classical</Genre>
      <Dates>10/01/02 through 10/31/02</Dates>
      <City>Irvine</City>
      <State>CA</State>
      <Zip>92720</Zip>
      <TicketPriceLow>$37.50</TicketPriceLow>
      <TicketPriceHigh>$90.00</TicketPriceHigh>
    </Concert>

    <Concert>
      <Show>Panoramas 2002</Show>
      <Performer>Panorama</Performer>
      <Genre>Country</Genre>
      <Dates>09/23/02 through 10/02/02</Dates>
      <City>Las Vegas</City>
      <State>NV</State>
      <Zip>89112</Zip>
      <TicketPriceLow>$29.00</TicketPriceLow>
      <TicketPriceHigh>$150.00</TicketPriceHigh>
    </Concert>

  </Concerts>
```

Listing 23.2 Concerts.xml file (continued).

At this point, the script in Listing 23.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. Your script and the Concerts.xml file shown in Listing 23.2 should be in the same directory.

You are now ready to run the script.

Step 3: Run the Concerts Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script completes, a message displays the best opportunity, and a summary of all opportunities examined—and their scores—is in the Output tab as XML.

If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Make sure Concerts.nql and Concerts.xml are present in the same directory.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Concerts script run, using fuzzy logic to score concert events and select the best opportunity.

Step 4: Understand the Concerts Script

We now want to make sure we understand every part of the Concerts script. The first line is a comment line.

```
//Concerts - ranks concert events using fuzzy logic
```

Variables are defined. The scoring variables include the total score, plus scores for genre, city, and price. They will hold fuzzy values and are thus declared to be of type *fuzzy*. A string and a fuzzy variable are declared to hold, respectively, the best concert name and the best score. As fuzzy scores are computed for each concert opportunity, every time a new high score is encountered these two variables will be updated.

```
fuzzy score, scoreGenre, scoreCity, scorePrice
string BestConcert
fuzzy BestScore
```

The concerts.xml file is loaded onto the stack and assigned to become the input stream. This permits the use of the *inputrecord* and *input* statements to extract data from the XML.

```
load "concerts.xml"
setinput
```

A *while* loop processes each record, which is identified by the <Concert> and </Concert> tags. The use of *inputrecord* searches for the next occurrence of these tags.

```
//read concerts from XML document

while inputrecord("Concert")
{
```

In the *while* loop, the individual fields of the concert are loaded into variables of the same name, with an *input* statement.

```
input Show, Performer, Genre, Dates, City, State, Zip, TicketPriceLow,
  TicketPriceHigh
```

The total score is initialized to 0.0. We will next be coming up with three subscores, which will be amalgamated into a final score shortly.

```
score = 0.0
```

First, the genre is examined and scored. The initial score is a neutral 5.0, pending special cases. A genre of “Classical” raises the score to 1.0. A genre of “Hip Hop” lowers it to 0.0.

```
scoreGenre = 0.5
if Genre=="Classical" { scoreGenre = 1.0 }
xif Genre=="Hip Hop" { scoreGenre = 0.0 }
```

Second, the city is examined and scored. The initial score is 0.25, indicating a dislike of travel to just about anywhere. If the city is Irvine, California, the score raises to a respectable 0.75. If the venue is Las Vegas, a favorite, a full 1.0 is scored.

```
scoreCity = 0.25
if Genre=="Irvine" { scoreCity = 0.75 }
if Genre=="Las Vegas" { scoreCity = 1.0 }
```

Third, the price is scored, this time with a condition function. The variable *scorePrice* is set by the function.

```
evaluate scorePrice
```

The total score can now be computed. The genre, city, and price scores are equal factors, so coming up with a total score is a simple matter of adding the three scores and dividing by three.

```
score = (scoreGenre + scoreCity + scorePrice) / 3.0
```

The score is now checked to see if it is a new best score. If so, *BestScore* is set to the new high score, and *BestConcert* is set to the name of the show.

```
if score > BestScore
{
    BestScore = score
    BestConcert = Show
}
```

A summary of the concert and its score are now output in XML. The *outputrecord* statement groups the output for this concert in a record. The *output* statement outputs some of the concert fields. The results will be visible in the *Output* tab of the development environment after the script ends.

```
    outputrecord "Concert"
    output Show, Performer, City, score
}
```

The *while* loop continues until there are no more concert opportunities to examine. A *show* statement declares the winning opportunity, the concert that scored the highest. The script ends.

```
show "The best concert opportunity is\n\n{BestConcert}, with a score of
{BestScore}"
end
```

Lastly, we have the definition of *scorePrice*, a fuzzy condition function. The low ticket price is checked against $100.00: If greater, the lowest possible score will be returned, 0.0. If equal to $0.00, the best possible score is returned, 0.0. Otherwise, the score is rated in the range $0.00 = 0.0 through $100.00 = 1.0. The result sets the variable *scorePrice*, which the main code uses in its computation of a total score.

```
condition scorePrice
{
    is 0.0, TicketPriceLow > 100.00
    is 1.0, TicketPriceLow == 0.00
    otherwise TicketPriceLow/100.00
}
```

At this point, you've seen how to perform fuzzy logic calculations in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Alter the likes and dislikes of the script to reflect different tastes.
- Change or extend the Concerts.xml file. See how the script handles mixtures of positive and negative characteristics in the data.

Chapter Summary

Fuzzy logic replaces the strict concept of true and false with infinite degrees in between the two extremes of 0.0 and 1.0. This makes it possible to better represent such things as the presence of characteristics, degrees of belief, and probabilities. The use of fuzzy values allows software to deal with real-world values and make better decisions, even with partial information.

Fuzzy values may be combined in a variety of ways to compute aggregate values, including summing individual values and dividing by the number of terms or weighting factors. Logical operations such as AND and OR have been defined for fuzzy logic.

Although fuzzy logic is primarily an approach the developer uses, NQL includes specific features to assist.

- There is a fuzzy variable type for holding fuzzy values.
- Fuzzy operators perform logical operations on fuzzy values.
- Fuzzy condition functions make it easy to express the logic for computing fuzzy values.

The NQL statements for fuzzy logic are *condition*, *evaluate*, *fuzzy*, *is*, and *otherwise*. The NQL fuzzy operators are *FUZZYAND* and *FUZZYOR*.

CHAPTER

24

Neural Networks

One of the most promising developments of artificial intelligence research is the neural network: software modeled after the workings of the human brain. Neural networks are fascinating in that they can learn, useful for their ability to match patterns and make predictions, and indispensable in that they can be taught tasks you may not have an algorithm for. Best of all, they are practical and can be applied to solve real-world problems. For all their power, neural networking has yet to become a standard item in the programmer's tool chest. Network Query Language includes support for four kinds of neural networks. This chapter begins with background information on neural networks, then goes on to describe how they are created, trained, and applied in NQL. In the tutorial at the end of the chapter you will create a neural network capable of recognizing letters, digits, and other symbols.

Introduction to Neural Networks

Most programmers have never worked with neural networks, so it is essential to introduce the topic before going into NQL specifics. Neural networks can't be used effectively without an understanding of their abilities and limitations. The following introduction puts neural networks into perspective.

Neural networks are a form of artificial intelligence, a field known for its failures as well as its successes. It is important to not confuse neural networks and artificial intelligence with actual thinking, which computers are not capable of. At the same time, dismissing neural networks out of hand would deprive you of a powerful tool.

What Is a Neural Network?

What exactly is a neural network? The term is unnerving to some people, evoking images of Frankenstein or robots. The term *neural networking* implies imitating the way the human brain works: simulating in software what we know about the workings of the human brain. We still know very little about how the brain functions, so neural networks are nowhere near being able to think. However, they are quite good at learning. That's really what a neural network is: software that can learn.

Neural networks are one of the few areas of computer science that don't require an algorithm, plan, or list of steps to follow. You can teach a neural network something by showing it examples of what you want. A learning system like this is tremendously valuable; there are some tasks we don't know how to approach using traditional programming methods. For example, imagine you were tasked with the job of writing software that matches fingerprints against fingerprint archives, a mammoth challenge. With a neural network, you have the option of training software by showing it fingerprint examples rather than writing complex, challenging code.

Neural networks must be used with care as they are not guaranteed to generate a correct answer. A well-trained network of the correct type and arrangement can make predictions that seem almost magical; a poorly-trained or poorly-designed network will not be very useful.

Applications for Neural Networks

What can you do with a neural network? Generally speaking, the applications are pattern matching, prediction, and analysis of information. Neural networks are being used a lot more than you might realize. Here are some examples of how neural networks are being put to use today:

- The wine industry has computers classify the grade of corks using neural networks.
- Analysts on Wall Street use neural networks to forecast how stocks are likely to perform.
- Economists in all major governments use neural networks to forecast the economy.
- Credit card systems use neural networks to detect when someone is likely to be using a card fraudulently.
- Petroleum companies use neural networks for process control.
- Neural networks grade pig carcasses in Danish slaughterhouses.
- Neural networks are used for cancer diagnosis in medical clinics.

- When an insurance claim is made, the likelihood of insurance fraud is checked by neural networks.
- Campaign committees use neural networks to identify the demographics of swing voters.
- Neural networks are used in the telecommunications industry to route long distance connections.
- The engine control of the Concorde is optimized with a neural network.
- Aerospace firms use neural networks to identify and retrieve 2-D and 3-D engineering designs—so they don't create a part they have already made for some other project.
- Electric companies use neural networks to reroute the power grid around trouble spots.

How Neural Networks Work

To understand how software neural networks work, we first need to know something about the genuine article: biological neurons in the human brain. Even though much of the brain is still a mystery, science has learned quite a bit about individual brain cells and how they interact with each other.

Biological Neurons

Biological neurons, or brain cells, seem fairly simple at the individual level. Figure 24.1 shows a neural brain cell, or neuron.

The neuron receives *inputs* on its *dendrites* in the form of electrical impulses. The core, or *soma*, performs some processing on those inputs. If the result of that processing

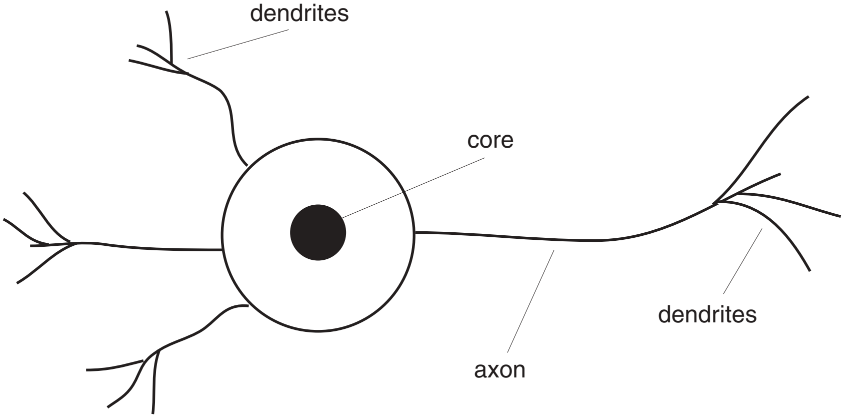


Figure 24.1 A biological neuron.

reaches a certain threshold, the *axon* generates *output* that is passed on to other neurons.

The Connections Are What Matter

Neurons are organized into layers, with a lot of complex connections between the neurons. The big thing about neurons is that the *connections* matter more than the neurons themselves do. As the brain learns, it alters the strength of the connections between neurons. This explains why the loss of a single brain cell is not catastrophic to the brain.

Artificial Neurons

So what does a software simulation of a neuron look like? Figure 24.2 shows the composition of an artificial neuron. Like the biological neuron, it must receive inputs, perform processing, and generate output. It must have connections to other neurons. Most importantly, it must be able to adjust the weights of those connections as it learns.

Whereas biological neurons pass electrical impulses over connections of varying strengths, artificial neurons pass numbers over connections of various weight.

The Artificial Neural Network (ANN)

When you group multiple artificial neurons into a network, you end up with an artificial neural network, or an ANN. When people use the popular term *neural network*, they really mean an ANN; after all, the brain is a neural network, too. Figure 24.3 shows an ANN. There are quite a few ways to arrange neurons, and a few different theories about what works best, but here are some typical characteristics of neural networks:

- The neurons are arranged in layers.
- The first layer is an *input layer*.
- The final layer is an *output layer*.
- There are frequently one or more *hidden layers*.
- Each neuron in a layer is connected to every other neuron in an adjacent layer.

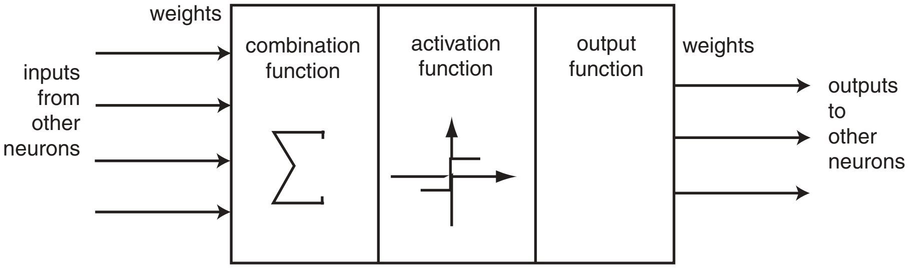


Figure 24.2 An Artificial Neuron.

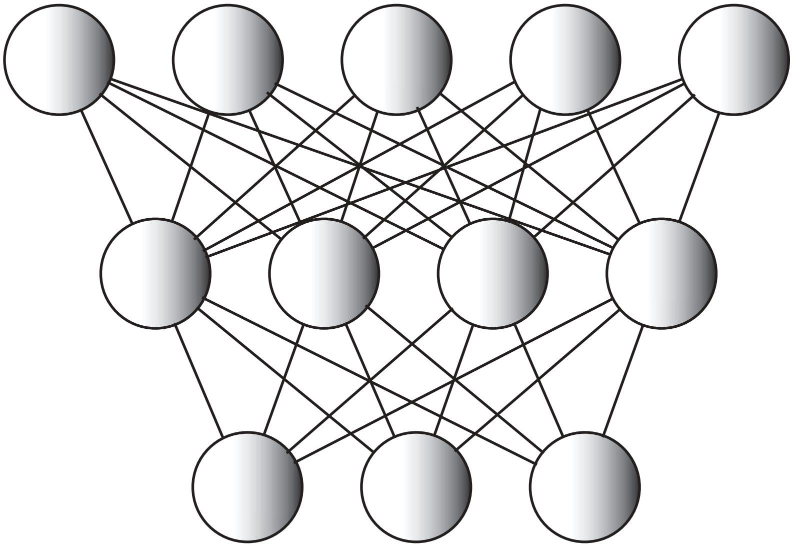


Figure 24.3 Artificial neural network.

Unlike the brain, artificial neural networks usually only connect neurons from one layer to another. The brain has more complex connections. There are different architectures for ANNs, reflected in the number of layers and the way in which they are connected. Some pass data in a forward direction only. Others feed backward and then forward as they learn.

The Learning Process

Neural networks have two modes of operation:

1. Training
2. Processing

You first train a neural network with training data, then put it to use processing live data.

Training the network is through example. Sets of input values and expected output values are supplied. In training mode, the network processes the input values and compares its own output values to the expected output values. At first they will be very different. The network then adjusts its weights and repeats the process, until it is producing output values acceptably close to the expected output values.

In processing mode, input values are supplied and the neural network generates output values in response. If the neural network has been well-trained, it will perform well even with input values it has never encountered before. What has happened is that the neural network has turned itself into the kind of device you need it to be by learning.

This learning ability can seem kind of spooky, but it's actually fairly simple under the hood. When you first create a neural network, it is full of random numbers and generates absolute gibberish. Its sole redeeming value is that you can show it training data—inputs and expected outputs—that it can learn from. It does this by processing the inputs and comparing its own outputs (nonsense at first) against the expected outputs. It then alters the weights between its neurons to better favor these results and repeats the entire process. This cycle of processing training inputs, comparing the actual output to the expected output, and adjusting weight continues as long as necessary until the neural network has literally turned itself into a machine able to generate correct values. In ideal situations, the neural network is then able to process new (previously unseen) inputs and generate accurate outputs. This is what makes a neural network useful for pattern matching and prediction applications.

What is really neat about this is that you can train a neural network to do something that you yourself have no idea how to go about doing. Let's say you want to do pattern recognition, so that your computer can identify certain kinds of objects from images. That's hard to do, and beyond the ability of many programmers. With a neural network, you don't have to know *how* to do it. You feed lots of pictures and describe the objects they represent to the network. Once it has learned enough, it gets pretty good at identifying new pictures of objects it has seen in the past.

The Catch

If neural networks sound too good to be true, that's because they sometimes are! Despite their utility, the actual job of choosing an architecture for a neural network (How many layers? How many nodes?) is something of a black art. Likewise, properly training a neural network is a mysterious discipline. One might design a neural network, carefully train it, and not find out how good or bad the system is until it is tried. With new inputs not previously encountered, you are trusting the neural network. Keep in mind that a neural network cannot tell you how it arrives at the conclusions it does.

Neural Network Architectures

There are different ways to arrange neural networks. An artificial neural network consists of nodes (artificial neurons) that are usually arranged in layers. The first layer of a neural network is known as the input layer. The final layer of a neural network is known as the output layer. Depending on the type and size of the network, there may or may not be one or more hidden layers between the input layer and the output layer. The nodes are connected with varying weights (connection strengths). These weights are altered as the network is being trained to learn a particular kind of task.

NQL supports the following four neural network architectures:

- Adaline network
- Backpropagation forward-feed network
- Kohonen self-organizing network
- Bidirectional associative memory (BAM)

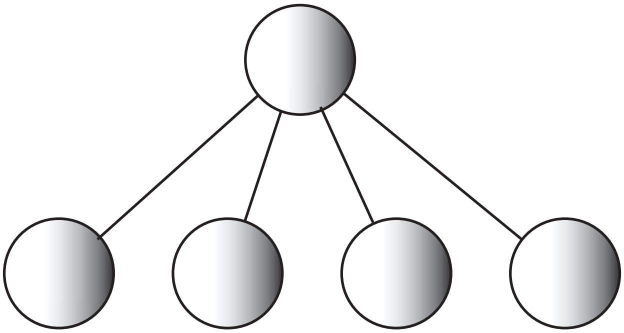


Figure 24.4 Adaline network with four input nodes.

Adaline Networks

A simple kind of neural network is the ADAptive LINear Element, or *Adaline* (Figure 24.4). This kind of network is also known as a *perceptron*. In an adaline network, there is an input layer and an output layer, but no hidden layers. The output layer has only one node, which means the adaline network is only useful for sorting its input data into one of two categories.

Adaline networks are trained using a supervised learning process. Data sets of input values and the desired output values are run through the network repeatedly, while the network adjusts its connection weights between nodes to reduce the degree of error. Eventually the network becomes a generalized system that can process new values.

Adaline networks can be trained to map their input values to one of two categories. For example, you could use an adaline network to learn to distinguish between points on a chart that are above or below a particular formula—perhaps indicating profit or loss.

In an adaline network, input values and the output value are in the range −1.0 to 1.0. Rounding the output value to −1 or +1 is common.

Although somewhat limited, adaline networks form the basis from which many of the more sophisticated kinds of neural networks are created.

Backpropagation Networks

A backpropagation forward-feed network has three (or more) layers (Figure 24.5). The input, hidden, and output layers can have any number of nodes. The greater complexity of the backpropagation network allows it to be used for more ambitious tasks than the primitive Adaline network. The middle layer of nodes provides an additional level of processing that permits multiple output nodes.

Backpropagation networks are trained using a supervised learning process. Data sets of input values and desired output values are run through the network repeatedly, while the network adjusts its connection weights between nodes to reduce the degree of error. Eventually the network becomes a generalized system that can process new values.

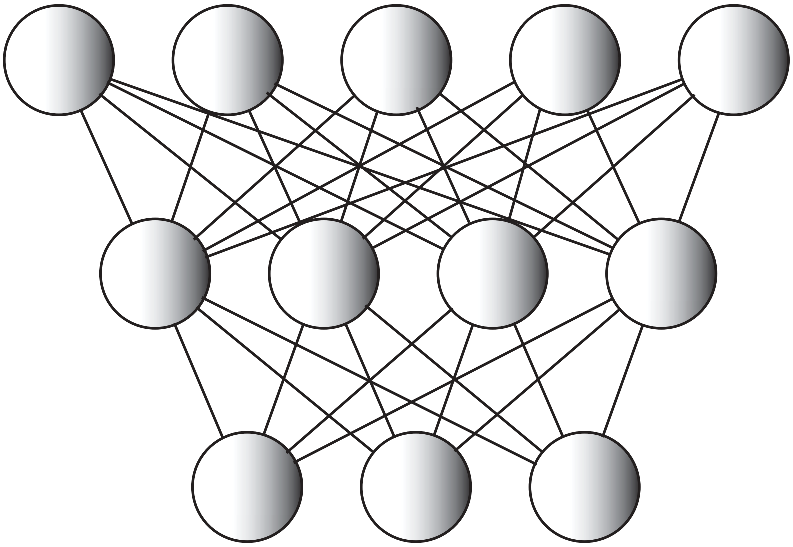


Figure 24.5 Backpropagation Network with five input nodes, four hidden nodes, and three output nodes.

Backpropagation networks are one of the more popular forms of neural networks in use today. They are able to learn to recognize patterns, such as handwriting, speech, or images. They can be used to make predictions, such as forecasting economic activity or indicating the likelihood that a credit card transaction is fraudulent based on the user's previous profile of use.

In a backpropagation network, input values and output values are in the range 0.0 to 1.0. Rounding the output value to 0 or 1 is common.

Kohonen Networks

Kohonen networks, also called self-organizing networks, map a set of input values onto a grid (Figure 24.6). Unlike the other kinds of neural networks, Kohonen networks don't have to be trained with a set of input and output values. As they process data, Kohonen networks cluster similar data.

These kinds of networks are called self-organizing because they do not need to undergo any kind of supervised training using known desired output values. Instead, they undergo unsupervised training: As a Kohonen network processes more and more input data, like values are grouped together. The results group together data with inherent similarities.

The other types of neural networks can be taught a particular task through the use of training data. Kohonen networks are suited for operating on a set of data where the kinds of results expected are not known in advance. For example, you might run customer and order profiles through a Kohonen network to discover new affinities between customer demographics and product types.

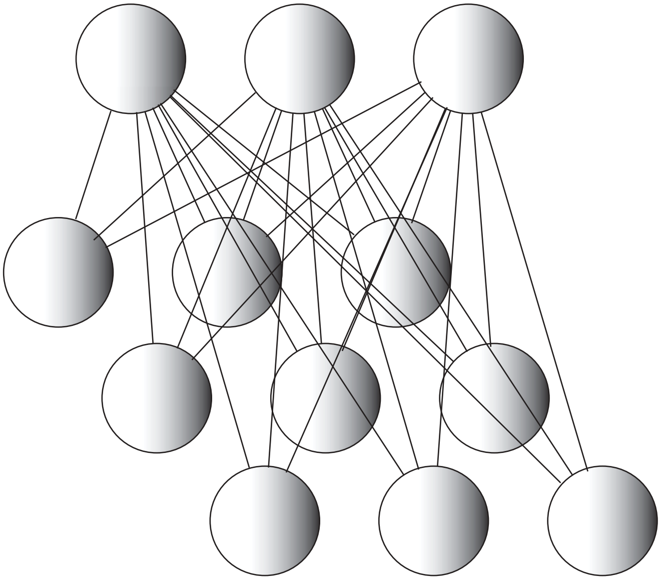


Figure 24.6 Kohonen Network with three input nodes and a 3 × 3 output grid.

In a Kohonen network, input values and output values are in the range 0.0 to 1.0. The output for a set of input patterns is a grid location, expressed as a row and column coordinate.

Bidirectional Associative Memory (BAM) Networks

A bidirectional associative memory (BAM) network functions like a random-access memory and is able to remember patterns (Figure 24.7). BAM networks have an input layer and an output layer.

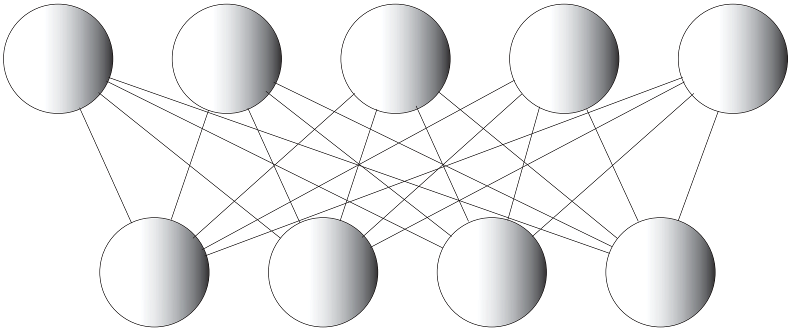


Figure 24.7 BAM Network with five input nodes and four output nodes.

A BAM network is trained using a supervised learning process. Data sets of input values and desired output values are run through the network repeatedly, while the network adjusts its connection weights between nodes to reduce the degree of error. Eventually the network memorizes the input value and the corresponding output values.

A BAM network is a good choice if you will be issuing input values to the network that you are sure were also a part of the training data set. A BAM network is not the best choice for a predictive system, in which new input values have to be handled. One good use for BAM networks is to memorize images.

In a BAM network, only −1.0 and 1.0 are valid input and output values (values in between are disallowed).

Neural Network Support in NQL

NQL includes about a dozen statements related to neural networks, but only a handful are needed on a regular basis. There are two basic sequences of operation:

- Create-and-train. Create, train, and save a new neural network.
- Open-and-run. Open a previously saved neural network and process live data with it.

The first sequence involves issuing *createneural*, *trainneural*, *saveneural*, and *closeneural* statements. For example:

```
createneural "backprop", 3, 4, 2
trainneural "net1.trn"
saveneural "net1.nnf"
closeneural
```

The second sequence involves issuing *openneural*, *processneural*, and *closeneural* statements. For example:

```
openneural "backprop", "net1.nnf"
processneural "net1.inp", "net1.out"
closeneural
```

The NQL Neural Network Statements

The NQL neural network statements are listed in Table 24.1. Of these, the five commonly used statements are *createneural*, *openneural*, *processneural*, *saveneural*, and *trainneural*.

There are function versions of some of the neural network statements. Table 24.2 lists the neural network functions.

The neural network statements and functions are described individually in the following sections.

Table 24.1 Neural Network Statements

STATEMENT	PARAMETERS	DESCRIPTION
closeneural		Closes the neural network
createneural	"adaline", *inputs*	Creates an adaline network
createneural	"backprop", *inputs*, *hidden*, *outputs*	Creates a backpropagation network
createneural	"kohonen", *inputs*, *x*, *y*	Creates a Kohonen network
createneural	"bam", *inputs*, *outputs*	Creates a bidirectional associative memory network
hideneural		Hides the neural network display
logneural	*filespec*	Logs neural network details to the specified text file
neuraltimeout	*seconds*	Causes training or processes to time-out after the specified interval
openneural	*type*, *filespec*	Opens a previously-saved neural network
processneural	*inputfile*, *outputfile*	Runs a neural network against the specified input data file
roundneural	*boolean*	Enables or disables rounding of output values
saveneural	*filespec*	Saves the neural network to disk
setneural	*learning-rate*	Sets parameters for an adaline network
setneural	*learning-rate*, *momentum*	Sets parameters for a backpropagation network
setneural	*initial-learning-rate*, *final-learning-rate*, *initial-neighborhood-size*, *neighborhood-decrement-level*, *max-iterations*	Sets parameters for a Kohonen network
showneural		Displays a neural network diagram with default title, size, and position
showneural	*title*	Displays a neural network diagram with the specified title
showneural	*title, x, y*	Displays a neural network diagram with the specified title and position

continues

Table 24.1 Neural Network Statements (continued)

STATEMENT	PARAMETERS	DESCRIPTION
showneural	*title, x, y, width, height*	Displays a neural network diagram with the specified title, position, and size
trainneural	*trainingfile*	Trains a neural network with the specified training file

Table 24.2 Neural Network Functions

FUNCTION	DESCRIPTION
boolean closeneural()	Closes the neural network
boolean openneural(*type*, *filespec*)	Opens a previously-saved neural network
boolean processneural(*inputfile*, *outputfile*)	Runs a neural network against the specified input data file

Creating a New Neural Network

The *createneural* statement creates a new neural network (in memory). The first parameter is the network type, which must be one of the following: *adaline*, *backprop*, *kohonen*, or *bam*.

The *createneural* statement requires additional parameters. The number and nature of those parameters vary depending on the network type, and are individually described in the four sections that follow.

Creating an Adaline Network

To create an adaline network, this form of the *createneural* statement is used:

```
createneural "adaline", inputs
```

The first parameter specifies the network type, *adaline*.

The second parameter specifies the number of input nodes, which must be greater than or equal to 1.

There is no need to specify hidden layer nodes or output nodes, because adaline networks have no hidden layer and there is always exactly one output node.

The following sample *createneural* statement creates an adaline network with four input nodes and one output node.

```
createneural "adaline", 4
```

Creating a Backpropagation Network

To create a backpropagation network, this form of the *createneural* statement is used:

```
createneural "backprop", inputs, hidden, outputs
```

The first parameter specifies the network type, *backprop*.

The second parameter specifies the number of input nodes, which must be greater than or equal to 1.

The third parameter specifies the number of nodes in the hidden layer, a number greater than or equal to 1.

The fourth parameter specifies the number of output nodes, which must be greater than or equal to 1.

The following sample *createneural* statement creates a backpropagation network with two input nodes, a hidden layer with three nodes, and one output node.

```
createneural "backprop", 2, 3, 1
```

Creating a Kohonen Network

To create a Kohonen self-organizing network, this form of the *createneural* statement is used:

```
createneural "kohonen", inputs, x, y
```

The first parameter specifies the network type, *kohonen*.

The second parameter specifies the number of input nodes, which must be greater than or equal to 1.

The third parameter specifies the X dimension of the results grid.

The fourth parameter specifies the Y dimension of the results grid.

The following sample *createneural* statement creates a Kohonen network with eight input nodes and a 3 × 3 results grid.

```
createneural "kohonen", 8, 3, 3
```

Creating a BAM Network

To create a BAM network, this form of the *createneural* statement is used:

```
createneural "bam", inputs, outputs
```

The first parameter specifies the network type, *bam*.

The second parameter specifies the number of input nodes, which must be greater than or equal to 1.

The third parameter specifies the number of output nodes, which must be greater than or equal to 1.

The following sample *createneural* statement creates a BAM network with 16 input nodes and three output nodes.

```
createneural "bam", 16, 3
```

Training a Neural Network

Once a neural network has been created or opened, it can be trained. The *trainneural* statement specifies the name of a training data file. The training file contains both input values and expected output values. The neural network repeatedly processes the data in the training file, adjusting connection weights between neurons, until it starts generating similar output.

```
trainneural "sample.trn"
```

Once a network has been trained, the usual next step is to save it to disk so that the network can be used in the future without repeating the training step.

What is in the training file? One data set per line, which includes both inputs and outputs. Numbers can be separated with spaces and/or commas. If you had a neural network with two inputs and one output, a training file might look like this:

```
0 0,0
0 1,1
1 0,1
1 1,0
```

This particular training file is very simple: It teaches a neural network how to perform a logical operation, the Exclusive-OR function. The preceding training file is interpreted as follows:

- Inputs 0 and 0 should yield an output of 0.
- Inputs 0 and 1 should yield an output of 1.
- Inputs 1 and 0 should yield an output of 1.
- Inputs 1 and 1 should yield an output of 0.

At times it is helpful to be able to express the input data on more than one line; this is especially true if the input data represents an image, where there are rows of pixels to be represented. In these cases, the plus sign (+) may be used to continue one line to the next. The following partial example training file teaches responses to board positions in a Tic Tac Toe game. The use of continuation lines allows the board positions to be expressed in a visually familiar 3 × 3 grid. There are nine inputs and four outputs for each record of training information. Although they could be coded into a single line, the use of continuation lines makes the information more meaningful for the person entering or modifying the training data.

```
0 0 0+
0 0 0+
0 0 0+
0 0 0 1

1 0 0+
0 0 0+
0 0 0+
1 0 0 1

0 1 0+
0 0 0+
0 0 0+
1 0 0 0
```

Saving a Neural Network

The *saveneural* statement saves the current open network to a disk file. A filename is specified. The file type .nnf (neural network file) is recommended as a convention.

```
saveneural "test.nnf"
```

After the network has been saved, it is possible to open it in the future to reuse the network.

Closing a Neural Network

When finished using a neural network, the *closeneural* statement closes the network and releases its resources.

```
closeneural
```

Opening a Neural Network

A previously saved neural network file (saved with *saveneural*) can be opened with the *openneural* statement. The network type and filename are specified.

```
openneural "backprop", "test.nnf"
```

After the network has been opened, it can process data.

Processing Data with a Neural Network

To process data with a neural network, first open it and then use the *processneural* statement. The two parameters are the names of an input file and an output file. The input file must be an existing file that contains input data, one set of values per line. Lines that end in a plus sign (+) are continued on to the next line. The output file is generated by the network and contains output values, one set per line, that correspond to the input data.

```
processneural "test1.inp", "test1.out"
```

The output results will be floating-point numbers or rounded to integer whole numbers, depending on a rounding setting. The default condition is to enable rounding. To change the setting, use the *roundneural* statement.

```
roundneural false
```

Displaying a Neural Network

You can see a visible diagram of a neural network while it is open. The *showneural* statement displays a window showing the network's topology (Figure 24.8).

The *hideneural* statement removes the diagram window.

```
hideneural
```

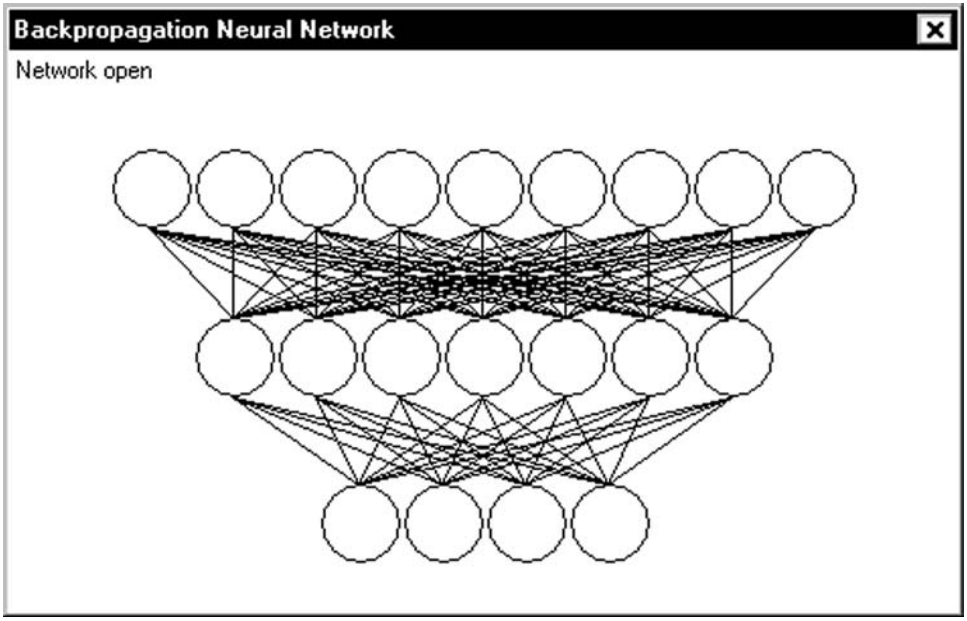


Figure 24.8 Example of the diagram window displayed by *showneural*.

Tutorial: Recognizing Symbols

Now it is time to see neural networks in action. In this tutorial, you will create a neural network that is able to recognize character symbols such as A, B, and C from a matrix of pixels. When finished, the network will examine characters such as the ones shown in Figure 24.9 and do a fairly good job of identifying them.

There are five steps in this tutorial:

1. Select a neural network design.
2. Create a training file.
3. Create and run a training script.
4. Create an input file.
5. Create and run a processing script.

Step 1: Select a Neural Network Design

We first need to consider the type and topology of the neural network that will be needed. Since the application is to supply training data and then process additional data, the appropriate choice is a backpropagation network. The topology of the network is primarily influenced by the number of inputs and outputs that will be needed. The number of inputs depends on the size of the character matrix (box of pixels) we will be processing. For the purposes of this tutorial, we will use a 5 × 7 pixel character matrix. Because we are assuming only two kinds of pixels (black and white), the total number of inputs will be 35 bits.

The number of outputs depends on the number of symbols we want to be able to accommodate. For this tutorial, we'll use four bits of output, which is enough to accommodate 16 symbols.

One final decision about our network design is the number of nodes in the center, hidden layer of neurons. Since there is no exact rule of thumb, we'll use a value of 24, somewhere between the 35 inputs and the four outputs.

You can easily change the number of inputs or outputs to reflect other sizes, and you should feel free to experiment after completing this tutorial; just make sure your training file and input file contain the expected number of inputs and outputs. Similarly, you can experiment with varying the number of nodes in the hidden layer of neurons.

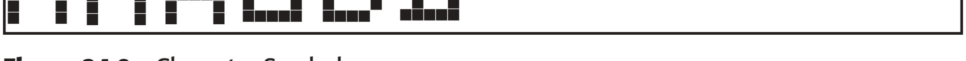


Figure 24.9 Character Symbols.

At this point, we know our neural network design. It will be a backpropagation network with three layers. The first layer is the input layer and has 35 inputs, reflecting a 5 × 7 matrix of pixels. The second layer is the output layer and has 24 neurons. The third and final layer is the output layer of four outputs.

Step 2: Create a Training File

Now it is time to come up with the training data for the neural network. This will require the creation of a text data file containing sequences of inputs and outputs. Because we want to be able to handle variations in the ways characters are expressed, it is important to have more than one example of each symbol in the training file. For example, you might include three or four variations of the letter A.

The training file we will use will be named symbols.trn. You can create this file using a text editor such as WordPad or NotePad, or use the symbols.trn file included on the companion CD.

Whether you create your own training file or use the one from the CD, the training information must be formatted in accordance with the neural network design: each training set is 35 input values followed by four output values. Listing 24.1 is a partial listing of the training data. In this case, we have arranged the input values on multiple lines to make it easy to see the symbol being described.

At this point you should have created a text file named symbols.trn with training data similar to that shown in Listing 24.1, or copied this file from the companion CD.

Step 3: Create and Run a Training Script

Now that the training file has been created, we can write and run the NQL script that will create, train, and save our neural network.

1. Launch the NQL development environment from your desktop.
2. Enter this script, or copy symbols1.nql from the companion CD.

   ```
   createneural "backprop", 35, 24, 4
   trainneural "symbols.trn"
   saveneural "symbols.nnf"
   closeneural
   ```

3. Save the script as symbols1.nql in the same directory where you saved the neural network training file, symbols.trn.
4. Now, run the script.

If the script runs successfully, a new file will be created named symbols.nnf in the same directory where you saved the NQL script. If the script does not run successfully, this is probably due to a typo in the script or the needed training file (symbols.trn) is missing or has invalid contents.

Let's be clear on how the statements in the script work. The first line of the script creates a backpropagation neural network in memory of 35 input nodes, 24 hidden

```
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0,0 0 0 0          ; 0 = blank

0 0 1 0 0+
0 1 0 1 0+
1 0 0 0 1+
1 1 1 1 1+
1 0 0 0 1+
1 0 0 0 1+
1 0 0 0 1,0 0 0 1          ; 1 = A

0 0 0 0 0+
0 0 1 0 0+
0 1 0 1 0+
0 1 1 1 0+
1 0 0 0 1+
1 0 0 0 1+
1 0 0 0 1,0 0 0 1          ; 1 = A

1 1 1 1 0+
1 0 0 0 1+
1 0 0 0 1+
1 1 1 1 0+
1 0 0 0 1+
1 0 0 0 1+
1 1 1 1 0,0 0 1 0          ; 2 = B

1 1 1 1 0+
1 0 0 0 1+
1 0 0 1 0+
1 1 1 1 1+
1 0 0 0 1+
1 0 0 0 1+
1 1 1 1 0,0 0 1 0          ; 2 = B
```

Listing 24.1 Neural network training file.

layer nodes, and four output nodes. The second line of the script trains the neural network by processing the input and output values from the training file, symbols.trn. The third line of the script saves the neural network to disk as a neural network file named symbols.nnf. Lastly, the fourth line of the script closes the neural network.

At this point you should have created, saved, and run an NQL script that looks identical to the one shown.

Step 4: Create an Input File

In step 2 we created a text file that contained training data. Now it is time to create a text file with the true input data we wish to process. This will again require creation of a text data file, this time containing only inputs—the outputs will be produced by the neural network itself in step 5.

The input file we will use will be named symbols.inp. You can create this file using a text editor such as WordPad or NotePad, or use the symbols.inp file included on the companion CD.

Whether you create your own input file or use the one from the CD, the input information must be formatted in accordance with the neural network's expectations: Each input set contains 35 inputs. Listing 24.2 is a partial listing of an appropriate input data file. In this case, we have arranged the input values on multiple lines to make it easy to see the symbol being described. Notice that we're free to vary from the original set of training data: Our neural network is intended to recognize new previously-unseen variations of symbols.

At this point you should have created a text file named symbols.inp with input data similar to that shown in Listing 24.2, or copied this file from the companion CD.

Step 5: Create and Run a Processing Script

Now that the input file has been created, we can write and run the NQL script that will open the previously-created neural network, process the input data, and generate results.

1. Launch the NQL development environment from your desktop.
2. Enter this script, or copy symbols2.nql from the companion CD.

   ```
   openneural "backprop", "symbols.nnf"
   processneural "symbols.inp", "synmbols.out"
   closeneural
   ```

3. Save the script as symbols2.nql in the same directory where you saved the other neural network files.
4. Now, run the script.

If the script runs successfully, a new file will be created named symbols.out in the same directory where you saved the NQL script. If the script does not run successfully, this is probably due to a typo in the script or the needed input file (symbols.inp) is missing or has invalid contents.

To view the results, open symbols.out in a text editor. Each line contains the results from processing a set of input bits. The values in this case indicate which symbol the network identified with each input character: 1 for A, 2 for B, and so on. Listing 24.3 shows a partial listing of the output results for the input data provided on the CD.

```
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0+
0 0 0 0 0

1 1 1 0 0+
1 0 0 1 1+
1 0 0 0 1+
1 1 1 1 0+
1 0 0 0 1+
1 0 0 0 1+
1 1 1 1 0

1 1 1 1 0+
1 0 0 0 1+
1 0 0 1 0+
1 1 1 0 1+
1 0 0 1 1+
1 0 0 0 1+
1 1 1 1 0

0 1 1 1 0+
0 1 0 1 0+
1 0 0 0 1+
1 1 1 1 1+
1 0 0 0 1+
1 0 0 0 1+
1 0 0 0 1

0 0 0 0 0+
0 0 1 0 0+
0 1 0 1 0+
1 1 1 1 1+
1 0 0 0 1+
1 0 0 0 1+
1 0 0 0 1
```

Listing 24.2 Neural network input file.

Let's be clear on how the statements in the script work. The first line of the script opens the neural network we created back in step 3, which has been trained for symbol recognition. The second line of the script puts the neural network to work, specifying the existing input file (symbols.inp) and the name of an output file to be generated (symbols.out). The third line of the script closes the neural network.

```
0,0,0,0
0,0,1,0
0,0,1,0
0,0,0,1
0,0,0,1
```

Listing 24.3 Output data from a neural network.

At this point you should have created, saved, and run an NQL script and experienced firsthand the results of symbol recognition by a neural network.

Further Exercises

You could amend this example in a number of ways, either to learn more about neural networks or to apply the scripts to an application need. Here are some interesting modifications that can be made:

- Change the number of inputs to reflect a different size character matrix, such as 7 × 9 characters. Of course, this means changing the scripts, training file, and input files to match the new number of input bits.
- Change the number of outputs to reflect a larger symbol set. The number of bits in the output should be determined by the binary number that can hold the total number of symbols you want to deal with. For example, to support 26 letters of the alphabet you need five output bits (0-32), because four bits (0-15) are too few.
- Change the number of nodes in the middle layer to see if there is a perceptible difference in how well the neural network performs. Too few nodes may result in a network that retains too little and guesses wrongly much of the time. Too many nodes may result in a network that memorizes the training patterns rather than learning how to generalize.
- Add additional code to the NQL scripts so that the training data and input data do not have to come from text data files. For example, your script could read data from another source, such as a database table, and generate the input files at runtime for the *processneural* statement.

Chapter Summary

Neural networks simulate the learning ability of the human brain in software. They provide the developer with a powerful tool: Software that can be taught to do a job through example. The seductive appeal of neural networks must be balanced with an appreciation for their varying reliability. Practice and experience in designing and training neural networks is a necessary ingredient in applying neural networks successfully.

NQL supports four neural network architectures: *Adaline*, *backpropagation*, *Kohonen self-organizing*, and *bidirectional associative memory*. A small collection of statements is all that is needed to create, train, and run neural networks.

In NQL, neural networks are initially created, trained, and saved. They may then be opened to process live data. Training data, input data, and output data are all stored in text data files.

The NQL statements for neural networking are *closeneural*, *createneural*, *hideneural*, *logneural*, *neuraltimeout*, *openneural*, *processneural*, *roundneural*, *saveneural*, *setneural*, *showneural*, and *trainneural*.

CHAPTER

25

Bayesian Inference

Network Query Language includes the ability to compute probabilities using Bayesian inference. Bayesian inference is a useful tool in some applications of artificial intelligence, probability, and statistics. This chapter provides some background on Bayes' theorem, then explores how it is used in NQL programming. The tutorial at the end of the chapter computes the likelihood of having a disease, given an imperfect test for it.

Introduction to Bayesian Inference

Bayesian inference is a form of statistical analysis that enables one to infer causes from effects, rather than the usual inferring of effects from causes. Thomas Bayes published his ideas in 1763. They were used successfully and improved by Laplace (1812), but they were later discredited and forgotten for many years. Bayesian inference was rediscovered in the twentieth century and has been found to have great application in statistics and computing applications. Unlike other methods, Bayesian inference takes prior knowledge into account when computing a probability.

Bayesian inferences generally involve two variables: A and B are generally used in this manual. There is a specific notation and terminology normally used in Bayesian inference. The symbol **|** means *given*; **~** means *not*; **,** means *and*; **+** means *or*. Table 25.1 shows some Bayesian terms and how they are understood.

Table 25.1 Bayesian Notation

NOTATION	PRONOUNCED	MEANING
P(A)	Probability of A	The probability that A is true
P(B)	Probability of B	The probability that B is true
P(~A)	Probability of NOT A	The probability that A is false
P(~B)	Probability of NOT B	The probability that B is false
P(A|B)	Probability of A, given B	The probability that A is true, given the fact that B is true
P(B|A)	Probability of B, given A	The probability that B is true, given the fact that A is true
P(A,B)	Probability of A AND B	The probability that both A and B are true
P(B,A)	Probability of B AND A	The probability that both B and A are true
P(A+B)	Probability of A OR B	The probability that either A or B are true
P(B+A)	Probability of B OR A	The probability that either B or A are true

The Sum Rule

In Bayesian inference, 1.0 represents completely true and 0.0 represents completely false. An important concept is that the true and false possibilities of a probability add up to 1.0. In Bayesian terminology, P(A) + P(~A) = 1, and P(~A) = 1.0 − P(A) .

```
P(A) + P(~A) = 1
P(~A) = 1 - P(A)
```

This is known as the *sum rule*.

The Product Rule

In Bayesian inference, the probability of two probabilities both being true can be computed by the following formula:

```
P(A,B) = P(A|B) x P(B)
```

This is known as the *product rule*.

Bayes' Theorem

The central Bayes' theorem is shown in Figure 25.1, from which there are many derivatives.

$$P(A|B) = \frac{P(B|A) \text{ x } P(A)}{P(B)}$$

Figure 25.1 Bayes' theorem,

Table 25.2 Derivations

IF YOU KNOW	THEN YOU CAN INFER	FORMULA
P(A), P(B|A), P(B|~A)	P(B)	P(B) = P(A) × P(B|A) + P(~A) × P(B|~A)
P(A), P(B|A), P(B)	P(A|B)	(P(B|A) × P(A)) / P(B)
P(A), P(B|A), P(B|~A)	P(A|B)	P(B) = P(A) × P(B|A) + P(~A) × P(B|~A), P(B) = P(A) × P(B|A) + P(~A) × P(B|~A)
P(A), P(B|A)	P(A,B)	P(A,B) = P(B|A) × P(A)
P(A), P(B), P(A,B)	P(A+B)	P(A+B) = P(A) + P(B) − P(A,B)

The implications of Bayes' theorem are shown in Table 25.2. Given certain known probabilities, other values can be inferred.

Bayesian Computations in NQL

When using NQL to express Bayesian terms like those listed in the previous section, the notation is similar, but square brackets are used in place of P( ). For example: [B | A] is how you express P(B | A) in an NQL script.

The NQL statements for Bayesian inference operations are listed in Table 25.3.

There is also a function version of the *infer* statement. The Bayesian inference statements and functions are described individually in the following sections.

Table 25.3 Bayesian Inference Statements

STATEMENT	PARAMETERS	DESCRIPTION
bayes	*var1*, *var2*	Declares Bayesian variables
prob	*term*, *value*	Declares a known value
probability	*term*, *value*	Declares a known value
infer	*term*, *var*	Computes a term using Bayesian inference

Performing Bayesian Inferences

To perform a Bayesian inference in NQL, the following sequence is used:

1. Use a *bayes* statement to indicate that a new Bayesian inference problem is being solved. This serves to identify the two variable names you will be using and clears away any prior result.
2. Use one or more *probability* statements to declare known values.
3. Use an *infer* statement to compute a desired value and store it in a result variable.

The following code computes a Bayesian inference.

```
bayes A, B
probability [A], 0.1
prob [B], 0.5
prob [B|A], 0.8
infer [A|B], result
show "The likelihood is " result
```

The preceding code is understood as follows: The probability of A is 0.1; the probability of B is 0.5; the probability of B given A is 0.8. Given what has been stated, compute the probability of A given B and store it in the variable *result*. Display *result*.

Starting a Bayesian Inference Problem

The first statement to use in a Bayesian inference problem is *bayes*. The two parameters are the names of the variables you will be using in the problem. These can be as generic as *A* and *B* or *X* and *Y*, or very specific such as *WhichCandidateWins* and *EconomyImproves*. The *bayes* statement also clears away the results of any prior Bayesian computations in order to start with a clean slate.

```
bayes var1, var2
```

Declaring Known Values

Next, one needs to declare the probability values that are known. The *probability* (or *prob*) statement declares a Bayesian reference and a value for it. The term is a Bayesian term such as [A] or [A | B]. The value can be any expression that evaluates to a floating-point number.

```
probability [term], value
```

The following code declares two Bayesian variables, then states three known values for the terms [X], [Y], and [Y | X].

```
bayes X, Y
probability [X], 0.1
```

```
prob [Y], 0.5
prob [Y|X], 0.8
```

Which Bayesian references do you specify? The ones that you already know and that are germane to the problem you are trying to solve. Network Query Language can compute any of the terms listed in Table 25.2.

Performing a Computation

The *infer* statement computes a desired Bayesian value. The parameters are the term to compute and a variable to receive the result.

```
infer term, var
bool infer(term, var)
```

The term is computed using Bayes' theorem. The following code shows the use of *infer* to perform a computation.

```
bayes X, Y
probability [X], 0.1
prob [Y], 0.5
prob [Y|X], 0.8
infer [X|Y], result
```

Tutorial: HealthTest

In this tutorial, we will apply Bayesian statistics to predict the likelihood of having a rare disease if you test positive for it, given some known information about the disease and the accuracy of the test.

Let's say we know the following facts:

- A rare disease exists, and one person in 1000 has this disease.
- There is a test for this disease, but it isn't completely accurate.
- If someone who has the disease is tested, 95 percent of the time the test will show positive, but 5 percent of the time it will erroneously indicate negative.
- If someone who does not have the disease is tested, 90 percent of the time the test will show negative, but 10 percent of the time it will erroneously show positive.

Here's the problem we want to solve: If you take the test for this disease and test positive, how worried should you be? What is the probability that you actually have the disease?

Our NQL script will compute the probability using Bayesian inference. In Bayesian terms, the information we already know can be represented as follows:

P(Disease) = 1/1000 (0.001)
P(No Disease) = 0.999
P(Positive Test | Disease) = 0.95
P(Negative Test | Disease) = 0.05
P(Negative Test | No Disease) = 0.90
P(Positive Test | No Disease) = 0.10

What we want to compute is P(Have Disease | Positive Test). We can compute this with Bayes' theorem, as shown in Figure 25.2.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the HealthTest script.
3. Run the HealthTest script.
4. Understand the HealthTest script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Set up a Bayesian problem.
- Compute Bayes' theorem.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the HealthTest Script

In the NQL development environment, enter the script shown in Listing 25.1 and save it under the name HealthTest.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

$$\text{P(Disease | Positive Test)} = \frac{\text{P(Positive Test | Disease) x}}{\text{P(Positive Test)}}$$

Figure 25.2 Bayesian solution.

```
//HealthTest - applies Bayes' theorem to rare disease test problem.

//Declare Bayesian variables

bayes HaveDisease, PositiveTest

//Define known information

probability [HaveDisease], 0.001
probability [PositiveTest|HaveDisease], 0.95
probability [PositiveTest|~HaveDisease], 0.10

//Infer the probability we want to solve.

infer [HaveDisease|PositiveTest], Likelihood
show "If you test positive, your likelihood of having the disease is:
" Likelihood
```

Listing 25.1 HealthTest script.

If you prefer, you may copy HealthTest.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select HealthTest.nql from the Tutorials\ch25 folder.

At this point, the script in Listing 25.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD.

Step 3: Run the HealthTest Script

You are now ready to run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. The script displays the answer:

```
If you test positive, your likelihood of having the disease is: 0.0094
```

You can breathe a sigh of relief, because your likelihood of actually being ill is quite low—less than 1 percent.

If the script does not run as expected, check the following:

- Check your script for typographical errors.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the HealthTest script run, performing a simple Bayesian computation. In the next step, we'll dissect the script and explain exactly how it works.

Step 4: Understand the HealthTest Script

We now want to make sure we understand every part of the HealthTest script. The first line is a comment line.

```
//HealthTest - applies Bayes' theorem to rare disease test problem.
```

The two Bayesian variables for this problem are defined: *HaveDisease* and *PositiveTest*.

```
//Declare Bayesian variables

bayes HaveDisease, PositiveTest
```

Next, *probability* statements define the information that is already known: the incidence of the disease in the population, the accuracy of the test when the result is positive, and the accuracy of the test when the result is negative.

```
//Define known information

probability [HaveDisease], 0.001
probability [PositiveTest|HaveDisease], 0.95
probability [PositiveTest|~HaveDisease], 0.10
```

We can now compute the desired value, the likelihood of having the disease if you test positive. The *infer* statement performs the computation, leaving the result in *Likelihood*. A *show* statement displays the result.

```
//Infer the probability we want to solve.

infer [HaveDisease|PositiveTest], Likelihood
show "If you test positive, your likelihood of having the disease is: "
Likelihood
```

At this point, you have seen an NQL script perform a Bayesian inference.

Chapter Summary

Bayes’ theorem is a theorem about conditional probabilities that takes prior information into account. Bayesian inference allows causes to be inferred from effects, and is useful in artificial intelligence applications.

- The *bayes* statement sets up a Bayesian problem and declares variables.
- The *probability* statement declares known information.
- The *infer* statement performs a Bayesian computation.

Bayesian inference is a useful tool for making software more intelligent.

CHAPTER

26

Interacting with the Desktop

When NQL applications are deployed on desktop computers, interaction with the desktop and users may be desirable. Network Query Language has features for opening documents, displaying console windows, prompting users for input, speaking, and playing multimedia files. This chapter covers each of the aforementioned areas. The tutorial at the end of the chapter plays MP3 music files from a Web site and displays status information in a console window.

Desktop Operations

The NQL desktop statements are summarized in Table 26.1.

Some of the desktop statements are also available as functions. Table 26.2 lists the desktop functions.

The desktop statements and functions are described individually in the following sections.

Opening Documents

Many operating systems associate applications with documents. For example, in Microsoft Windows a .doc file is associated with Microsoft Word and a .xls file is associated with Microsoft Excel. Double-clicking on a file type from Windows Explorer opens

Table 26.1 Desktop Statements

STATEMENT	PARAMETERS	DESCRIPTION
ask	*caption*, *message*	Displays a message and retrieves a *yes* or *no* response
closeconsole		Closes a console window
closespeech		Closes a text-to-speech session
getallvoices		Returns a list of available speech engine voices
getcurrentvoice	*var1*, *var2*	Returns the current voice name and speaking speed
isspeaking		Reports on whether text-to-speech is still in progress
openconsole	[*caption* [, *stay-on-top*]]	Opens a console window
opendoc	*file* [, *verb*]	Opens a document on the desktop
openspeech		Opens a text-to-speech session
play	*filespec*	Plays a multimedia (audio or video) file
prompt	*caption*, *message*, [*default* [, *allow-blank*]]	Prompts for an input value
sendkeys	*characters*	Sends keystrokes to the current active application
setconsole	*control*, *value*	Updates a control of a console window
setvoice	*voice-name* [, *speed*]	Selects a voice and speaking speed
show	[*value-list*]	Displays data in a message window
speak	[*text*]	Speaks text and waits for completion
speakcontinue	[*text*]	Speaks text without waiting for completion
Status	*type* [, *value*]	Updates the display of an agent's desktop skin
stopspeaking		Stops speaking
waitconsole		Waits for a user to close a console window

the specified file in its default application. Your NQL scripts can similarly open documents and, through association, their applications. This capability is useful for displaying the results of a task. For example, you might use the image statements to create a graphic file, then wish to have the graphic file opened on the desktop for presentation.

Table 26.2 Desktop Functions

FUNCTION	DESCRIPTION
boolean closespeech( )	Closes a text-to-speech session
boolean ask(*caption message*)	Displays a message and retrieves a *yes* or *no* response
string getallvoices( )	Returns a list of available speech engine voices
boolean getcurrentvoice(*var1*, *var2*)	Returns the current voice name and speaking speed
boolean isspeaking( )	Reports on whether text-to-speech is still in progress
boolean openspeech( )	Opens a text-to-speech session
boolean prompt(*caption*, *message*, [*default* [, *allow-blank*]])	Prompts for an input value
boolean setvoice(*voice-name* [, *speed*])	Selects a voice and speaking speed
boolean speak([*text*])	Speaks text and waits for completion
boolean speakcontinue([*text*])	Speaks text without waiting for completion
boolean stopspeaking( )	Stops speaking

The *opendoc* statement opens a document on the desktop. The parameters are the file specification of a document (file) to open, and an optional verb. The verb defaults to the operating system standard open verb ("Open" in Windows).

```
opendoc filespec [, verb]
```

The *opendoc* statement does not change the success/failure condition.

Sending Keystrokes

To control another application on the desktop, you can send it keystrokes as if a user were entering keystrokes from a keyboard. The *sendkeys* statement takes one parameter, a string of character values to send to the current active application on the desktop.

```
sendkeys characters
```

For non-printing characters, the names listed in Table 26.3 may be embedded in the character text.

The specified character(s) are sent to the current active application. The following code opens a document in Microsoft Word, then uses *sendkeys* to close the application by selecting *File, Exit* from the menu.

Table 26.3 Special Character Names for Sendkeys

NOTATION	MEANING
{CTRL+c}	Sends a character with Ctrl pressed
{SHIFT+c}	Sends a character with Shift pressed
{ALT+c}	Sends a character with Alt pressed
{ENTER}	Sends the Enter key
{ESC}	Sends the Escape key
{TAB}	Sends the Tab key
{SPACE}	Sends the Space key
{RIGHT}	Sends the Right Arrow key
{LEFT}	Sends the Left Arrow key
{UP}	Sends the Up Arrow key
{DOWN}	Sends the Down Arrow key
{PAGEUP}	Sends the Page Up key
{PAGEDOWN}	Sends the Page Down key
{INSERT}	Sends the Insert key
{DELETE}	Sends the Delete key
{BACK}	Sends the Backspace key
{F1}...{F12}	Sends the corresponding Function key

```
opendoc "c:\\letter.doc"
sendkeys "{ALT+F}x"
```

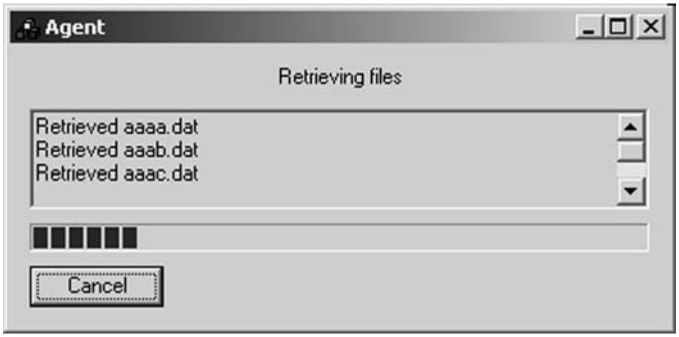
The *sendkeys* statement does not change the success/failure condition.

Console Windows

NQL scripts can display a desktop console window that can be used to communicate general status, percentage of progress, and messages. Figure 26.1 shows a console window.

The sequence for using console windows is as follows:

1. Open a console window with *openconsole*.
2. During the course of script execution, update the console with *setconsole*.



Retrieved aaaa.dat
Retrieved aaab.dat
Retrieved aaac.dat

Figure 26.1 Console window.

3. If you want to wait for the user to explicitly close the console window, issue a *waitconsole* statement.
4. Close the window with *closeconsole*.

Opening a Console Window

The *openconsole* statement opens a console window. The parameters are an optional window caption and a *stay-on-top* flag. The *stay-on-top* flag, if set to true, keeps the console window top-most on the desktop. It defaults to false.

```
openconsole [caption [, stay-on-top]]
```

The console window displays, managed by its own user-interface thread, and vanishes when one of the following occurs: The user closes the window; the user clicks *Cancel*; a *closeconsole* statement executes; or the script ends.

If a user closes the console window or clicks its *Cancel* button, the NQL script halts after it completes the statement it is currently executing. The following code opens a console window using *openconsole*.

```
openconsole "Mail Agent Status", true
```

After a console is opened, use *setconsole* to update its display.

Updating a Console Window

The *setconsole* statement updates the display of a console window. The parameters are a control name (“action”, “log”, or “progress”) and a value for the control.

```
setconsole control, value
```

The console window updates according to the parameters. If a control of “action” is specified, the action label of the console window is set to the text of the value parameter. If a control of “log” is specified, the text of the value parameter is added to a growing list box on the console window. If a control of “progress” is specified, the progress bar on the window is set to the percentage of the value parameter, which should be a number between 0 and 100. The following code shows all three forms of *setconsole*.

```
openconsole "Agent", true
setconsole "action", "Retrieving files"
setconsole "log", "Retrieved aaaa.dat"
setconsole "log", "Retrieved aaab.dat"
setconsole "log", "Retrieved aaac.dat"
setconsole "log", "Retrieved aaad.dat"
setconsole "log", "Retrieved aaae.dat"
setconsole "progress", 15
closeconsole
```

Waiting for a Console Window

The *waitconsole* statement waits for a user to click a *Close* button on the console. There are no parameters. Use *waitconsole* when you want the user to be aware of and acknowledge that a script has completed.

```
waitconsole
```

The caption of the *Close* button is changed from “Cancel” to “Close”. The script waits until the console window has been closed by the user before proceeding with execution. The following code shows the use of *waitconsole*.

```
openconsole "Agent", true
setconsole "action", "Retrieving files"
setconsole "progress", 10
    ....
setconsole "action", "Completed"
setconsole "progress", 100
waitconsole
closeconsole
```

Closing a Console Window

The *closeconsole* statement closes a console window. There are no parameters.

```
closeconsole
```

The console window is closed. The following code shows the use of *closeconsole*.

```
openconsole "Agent", true
    ....
```

```
setconsole "action", "Completed"
setconsole "progress", 100
closeconsole
```

Dialogs

NQL has features for using dialog windows to display messages, ask questions, and accept input.

Displaying Messages

The *show* statement displays text in a message window. The parameters are one or more values to display. If no values are listed, the top data item on the stack is displayed.

```
show [value-list]
```

Figure 26.2 shows the appearance of the *show* dialog window.

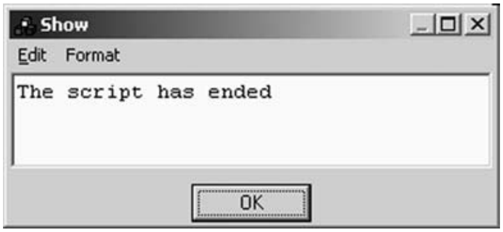
The following code shows the use of *show* to display stack data, computed values, and a text message.

```
push "test"
show

a = 5
b = 3.5
show a, b, a * b

show "The End"
```

If multiple values are specified, each value is displayed serially in its own window.

Accepting Input

The *prompt* statement or function displays a message and accepts input from the keyboard. The parameters are a dialog window caption, a message to display, and optionally a default input value and an *allow-blank* flag which defaults to true. An empty input will only be accepted if the *allow-blank* parameter is true.



The script has ended

Figure 26.2 *Show* window.

```
prompt caption, message, [default [, allow-blank]]
boolean prompt(caption, message, [default [, allow-blank]])
```

The caption and message are displayed in a dialog window which has a text box for data entry. If a default value is specified, it is pre-loaded into the text box. The user enters input and clicks *OK*. If the user enters input, a success code results (the function returns true) and the input data is pushed onto the stack. If the user cancels the dialog, a failure condition results (the function returns false). Figure 26.3 shows the appearance of the *show* dialog window.

The following code shows the use of *prompt* to collect input fields.

```
prompt "Data Entry", "What is your first name?"
pop first
prompt "Data Entry", "What is your last name?"
pop last
show first, last
```

Asking Yes/No Questions

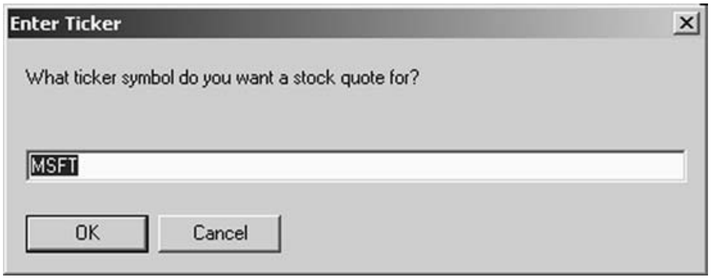
The *ask* statement or function displays a message and waits for a *yes* or *no* response. The parameters are a caption and a message for the dialog window.

```
ask caption, message
boolean ask(caption, message)
```

The dialog is displayed with *Yes* and *No* buttons. If *Yes* is selected, a success condition results (the function returns true). If *No* is selected, a failure condition results (the function returns false). Figure 26.4 shows the appearance of the *ask* dialog window.

The following code shows the use of *ask* to collect input fields.

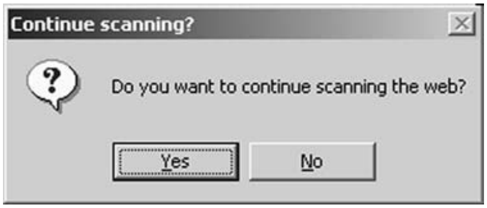
```
if !ask("Continue scanning?", "Do you want to continue scanning the web?")
{
    end
}
```


Enter Ticker

What ticker symbol do you want a stock quote for?

MSFT

Figure 26.3 Prompt window.



Continue scanning?

Do you want to continue scanning the web?

Figure 26.4 Ask window.

Text-to-Speech

NQL can speak text, if your computer has the appropriate hardware and software. For a Windows PC, this means speakers, a sound card, and a Microsoft SAPI speech engine. The common sequence for text-to-speech operations is as follows:

1. Open a text-to-speech session with *openspeech*.
2. Optionally select a voice and a speaking speed with *setvoice*.
3. Speak text using *speak* or *speakcontinue*.
4. Close the text-to-speech session with *closespeech*.

Opening a Text-to-Speech Session

The *openspeech* statement opens a text-to-speech session. There are no parameters.

```
openspeech
boolean openspeech()
```

If a text-to-speech session is successfully opened, a success condition results (the function returns true). The following code shows the use of *openspeech*.

```
openspeech
speak "It's 5:00, you can go home now."
closespeech
```

Note that the use of *openspeech* is an optional formality because most of the speech statements perform an automatic *openspeech* if necessary. However, *openspeech* is useful for checking the availability of a speech engine early in your program.

Speaking

The *speak* statement or function speaks text. The optional parameter is the text to speak; if omitted, the top data item on the stack is used as the text to speak.

```
speak [text]
boolean speak([text])
```

The text is spoken through the computer's sound card and speakers. The NQL script does not continue execution until the speech rendering completes. The following code shows the use of *speak*.

```
speak "Attention! Your stock is at an all-time high!"
```

To speak without waiting, use *speakcontinue* instead of *speak*.

Speaking without Waiting

The *speakcontinue* statement or function speaks text. It is identical to *speak* except that it does not wait for the speaking to complete. The text-to-speech occurs in parallel, allowing your script to race ahead on to other statements while the text is being spoken. The optional parameter is the text to speak; if omitted, the top data item on the stack is used as the text to speak.

```
speakcontinue [text]
boolean speakcontinue([text])
```

The text is spoken through the computer's sound card and speakers. The following code shows the use of *speakcontinue*.

```
speakcontinue "As we speak, your computer is being rebooted."
reboot
```

To speak and wait for speech to be rendered before continuing, use *speak* instead of *speakcontinue*.

Stopping Speech

If speech is rendered using *speakcontinue*, you may wish to stop it at a later point even if it has not finished, as in the case of a fatal error. The *stopspeaking* statement or function halts text-to-speech in progress. There are no parameters.

```
stopspeaking
boolean stopspeaking()
```

Any text-to-speech in progress is halted. If the speech engine could not be initialized, a failure condition results (the function returns false). The following code shows the use of *stopspeaking* to break off a narrative after a certain amount of time has passed.

```
get sURL
speakcontinue
at "12:00p"
stopspeaking
show "Time for lunch"
```

Checking for Speech in Progress

The *isspeaking* statement or function checks if any text-to-speech is still in progress. If the *speakcontinue* statement is used to begin speech in parallel, *isspeaking* may be used to determine if it is finished or not. There are no parameters.

```
isspeaking
boolean isspeaking()
```

If text-to-speech is still in progress, a success condition results (the function returns true). The following code shows the use of *isspeaking*.

```
load "filibuster.txt"
speakcontinue
while isspeaking()
{
     sleep 5
}
show "Done at last!"
```

Selecting a Voice

The *setvoice* statement selects a voice and a speaking speed. The parameters are a voice name and, optionally, a speaking speed.

```
setvoice voice-name [, speed]
boolean setvoice(voice-name [, speed])
```

If the specified voice is available, it is loaded and a success condition results (the function returns true). The following code shows the use of *setvoice*.

```
openspeech
setvoice "Sam"
speak "It's 5:00, you can go home now."
setvoice "Mary", 200
speak "Don't forget to get some milk on the way home."
closespeech
```

Declaring a voice is not mandatory when using text-to-speech; a default voice is selected when the speech session is opened. You can discover the available voice names with the *getallvoices* statement.

Determining Available Voices

The *getallvoices* statement or function returns a list of available speech engine voices. There are no parameters.

```
getallvoices
string getallvoices()
```

The available voices are returned as a single string value with a newline separating each voice in the list. In the case of the statement, the string is pushed onto the stack. In the case of the function, the string is the function result. If a speech engine could not be initialized, a failure condition results. The following code shows *getallvoices* in action.

```
getallvoices
while nextline(voice)
{
    show voice()
}
```

To determine the current voice in use by the speech engine, use *getcurrentvoice*.

Determining the Current Voice

The *gecurrentvoice* statement or function returns the name and speed of the current voice in use by the speech engine. The parameters are a variable name to receive the voice name and a variable name to receive the speaking speed.

```
getcurrentvoice var1, var2
boolean getcurrentvoice(var1, var2)
```

The current voice name and speed are returned in the specified variables. If a speech engine could not be initialized, a failure condition results (the function returns false). The following code shows the use of *getcurrentvoice*.

```
string VoiceName, VoiceSpeed
getcurrentvoice VoiceName, VoiceSpeed
```

To determine all voices available to the speech engine, use *getallvoices*.

Playing Audio and Video

NQL can display images and play audio and video files. Images can be displayed with the *opendoc* statement, described earlier in this chapter. Audio and video files can be played with the *play* statement. The parameter is the file specification of an audio or video file.

```
play filespec
```

If the specified audio or video file is found and is of a type supported on your computer, a success condition results and it is played on the desktop. The following code shows the use of *play* to play audio.

Table 26.4 Supported Audio and Video Types

FILE TYPE	DESCRIPTION
.mid	MIDI audio
.mp3	MP3 audio
.wav	Wave audio
.mpg	MPEG video
.mpeg	MPEG video
.mov	QuickTime video

```
play "http://www.my-music-site.com/oldies/videostar.mp3"
```

The supported file types for Windows PCs are shown in Table 26.4. Your computer must also have the necessary drivers and software for playing these kinds of files.

Desktop Skins

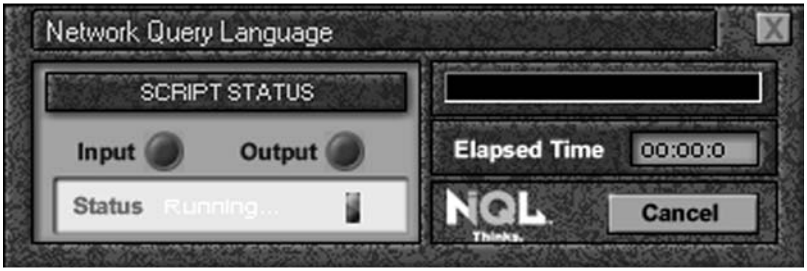
One of the options available in the Windows edition of NQL is the ability to generate an .exe file for a script that is destined to run on a desktop. These generated .exe files make use of *desktop skins* for a customized appearance that resembles a modern MP3 player. Figure 26.5 shows the appearance of a desktop .exe file.

While the script is running, the *status* statement may be used to update the display. The parameters are the function to perform and a value to go with it. Not all functions require a value.

```
status type [, value]
```

With a few status statements, your script can show ongoing status, display errors, indicate progress, and turn on and off lights indicating input/output is in progress.

The available types and values are listed in Table 26.5. The desktop skin facility gives your NQL scripts a professional, cutting-edge look.



Status Running...

Elapsed Time 00:00:0

Figure 26.5 Desktop agent .exe file.

Table 26.5 Status Types and Values

TYPE	VALUE	DESCRIPTION
start		Indicates that the script is running
input	“on”	Turns on the input indicator
input	“off”	Turns off the input indicator
output	“on”	Turns on the output indicator
output	“off”	Turns off the output indicator
message	text	Displays a short message (script continues)
messagebox	text	Displays a message in a window (script waits)
errorbox	text	Displays an error message in a window (script waits)
progress		Increments the progress bar
done		Indicates that the script is finished

Tutorial: MP3

Now it’s time to put some of this information to work in NQL. In this tutorial, you will create a script that reads a Web page from a music site and plays all of the MP3 music files on the page. The script will also display a console window to show its progress.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the MP3 script.
3. Run the MP3 script.
4. Understand the MP3 script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Retrieve and play MP3 files.
- Display and update a console window.

Let’s begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the MP3 Script

In the NQL development environment, enter the script shown in Listing 26.1 and save it under the name MP3.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//MP3 - plays all MP3 files on a web page.

GetWebSite:
    prompt "MP3 Player - Select Web Site",
        "Enter a web page that contains MP3 (.mp3) music files",
        "http://www.my-music-site.com"
    else Exit
    pop sURL

    if sURL = ""
    then Exit

    push sURL
    absurl sURL
    pop sURL
    sAbsolute = sURL

    //open a console window on the desktop and keep it topmost

    openconsole "MP3 Player", true

    // Get the page containing the music files

    get sURL
    else Exit
    setconsole "log", sURL

    //count how many music files there are

    count '.mp3'
    pop total
    played = 0
```

continues

Listing 26.1 Client script.

```
    //find each link on the page, and play the ones that are .mp3
(MP3) files

    match '<a href="{MP3File}"'
    while
    {
        //make the link absolute and check if it contains a .mp3
extension

        push MP3File
        absurl sAbsolute
        store MP3File
        played = played + 1

        contains ".mp3"
        then
        {
            parseurl protocol, server, object, port, path, filename

            //update the title and progress bar on the console window

            setconsole "action", "Playing " | filename
            if played > total
            {
                total = played
            }
            n = (played * 100) / total
            setconsole "progress", n

            play MP3File
        }
        pop
        match '<a href="{MP3File}"'
    }

    ask "Play More Music?", "Do you want to play selections from
another music site?"
    else
    {
        closeconsole
        end
    }
    goto GetWebSite

Exit:    end
```

Listing 26.1 Client script (continued).

If you prefer, you may copy MP3.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select MP3.nql from the Tutorials\ch26 folder.

At this point, the script in Listing 26.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD.

Step 3: Run the MP3 Script

Now run the server script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. A window prompts for the name of a music site. Enter the URL of a Web page where you know there are links to MP3 files. The script plays the MP3 files on the page in sequence and displays progress information in a console window. If this does not happen, check the following, or try a different music site.

- Check your script for typographical errors.
- Make sure your PC is equipped to play MP3 files.
- Check the NQL client's *Errors* tab for error messages.

Step 4: Understand the MP3 Script

Let's go through the MP3 script and see how it works. The first line is a comment line.

```
//MP3 - plays all MP3 files on a web page.
```

A *prompt* statement gets a Web site from the user, where there are hopefully MP3 files to be found. If the user cancels or enters an empty string, the script ends. The URL is saved in *sURL*.

```
GetWebSite:
    prompt "MP3 Player - Select Web Site",
        "Enter a web page that contains MP3 (.mp3) music files",
        "http://www.my-music-site.com"
    else Exit
    pop sURL

    if sURL = ""
    then Exit
```

The URL is made absolute and stored in *sAbsolute*. The absolute URL will be needed later.

```
push sURL
absurl sURL
```

```
pop sURL
sAbsolute = sURL
```

A console window is displayed on the screen with *openconsole*, which will be used to display progress as music files are played.

```
//open a console window on the desktop and keep it topmost

openconsole "MP3 Player", true
```

Next, the music Web page is retrieved with *get*, which leaves the HTML on the stack. The URL is listed in the console window's log area.

```
// Get the page containing the music files

get sURL
else Exit
setconsole "log", sURL
```

The *count* statement counts the occurrences of .mp3 in the HTML on the stack, which should be the number of links to .mp3 files on the page. The count is popped into *total*. The variable *played*, which will track the number of .mp3 files played, is initialized to zero.

```
//count how many music files there are

count '.mp3'
pop total
played = 0
```

The *match* statement locates the first link, setting the filename into MP3File. However, we don't know if the link is to an .mp3 file yet. A *while* loop is entered if a match is found.

```
//find each link on the page, and play the ones that are .mp3 (MP3) files

match '<a href="{MP3File}"'
while
{
```

The link may be relative, which is no good if we want to retrieve the page; *get* needs an absolute URL. The *sAbsolute* variable now comes into play with an *absurl* statement to make the *MP3File* variable a complete URL. We leave the URL on the stack because we will reference it later.

```
//make the link absolute and check if it contains a .mp3 extension

push MP3File
absurl sAbsolute
```

```
store MP3File
played = played + 1
```

If the link contains .mp3, we want to play it. The URL is parsed with *parseurl* so that the filename portion of the URL can be displayed in the console window with *setconsole*. The progress bar is updated, also with *setconsole*.

```
contains ".mp3"
then
{
    parseurl protocol, server, object, port, path, filename

    //update the title and progress bar on the console window

    setconsole "action", "Playing " | filename
    if played > total
    {
        total = played
    }
    n = (played * 100) / total
    setconsole "progress", n
```

At last, we can play the .mp3 file. The *play* statement plays the music.

```
    play MP3File
}
```

Now that the .mp3 file has been played, it's on to the next one. The URL is popped off of the stack. Still on the stack is the HTML of the Web page, where the previous *match* left off. Another *match* statement finds the next link, and if there is one the *while* loop will continue. When all .mp3 files have been played, the *while* loop exits.

```
    pop
    match '<a href="{MP3File}"'
}
```

The user is prompted with *ask* to continue on to another page. A *yes* response causes a jump to the top of the script. A *no* response closes the console window and ends the script.

```
    ask "Play More Music?", "Do you want to play selections from another
music site?"
    else
    {
        closeconsole
        end
    }
    goto GetWebSite

Exit:    end
```

At this point, you've seen an NQL script play .mp3 music files and display a console window.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Change the script to play other kinds of multimedia files, such as .mid (MIDI) sound files.
- Change the script to not only play song files, but also to download them, if this is legal for the song material in question.

Chapter Summary

Network Query Language can perform a variety of operations on the desktop.

- Multimedia audio and video files can be played with the *play* statement
- Documents and applications can be opened on the desktop with the *opendoc* statement.
- The *sendkeys* statement sends keystrokes to the active desktop application.

Dialog windows can interact with desktop users to display information, ask questions, or retrieve input data.

- The *show* statement displays a message
- The *ask* statement asks a yes-or-no question.
- The *prompt* statement retrieves an input value.

Console windows can display windows with a status label, scrolling text area, and progress bar.

- The *openconsole* statement opens a console window.
- The *setconsole* statement updates a control on the window.
- The *waitconsole* statement waits for a console window to be closed.
- The *closeconsole* statement closes a console window.

Text-to-speech operations allow NQL scripts to speak.

- The *openspeech* statement opens a text-to-speech session.
- The *setvoice* statement selects a voice and speed.

- The *speak* statement speaks text.
- The *speakcontinue* statement speaks text without waiting.
- The *stopspeaking* statement halts speech started with *speakcontinue*.
- The *isspeaking* statement checks to see if a *speakcontinue* is still in progress.
- The *closespeech* statement closes a text-to-speech session.
- The *getallvoices* statement returns a list of available voice names.
- The *getcurrentvoice* returns the current voice name and speed.

The desktop capabilities allow NQL scripts to communicate with users and integrate with desktops in a variety of ways.

CHAPTER

27

Network Monitoring

Modern networks have a diverse collection of devices attached to them, many of which are smart enough to be interrogated and controlled programmatically. Network Query Language has facilities for monitoring and controlling devices through the Simple Network Management Protocol (SNMP). This chapter briefly introduces SNMP and covers NQL's SNMP capabilities. The tutorial at the end of this chapter scans a network and locates devices that respond to SNMP inquiries, showing the results in a Web page.

Introduction to Network Monitoring

The Simple Network Management Protocol allows network devices to be monitored and controlled. It is very common for modern network devices such as computers, routers, hubs, and phone systems to support SNMP.

The two kinds of parties one will find in the world of SNMP are *managers* and *agents*. Managers interrogate devices or send them commands, as would be the case with a console program for monitoring network status. Agents respond to commands from managers and do what they are told; thus, each SNMP-compliant network device contains an agent.

SNMP devices tend to have a lot of information fields. These are logically arranged into a Management Information Base (MIB). The MIB may or may not be stored in a physical device; the important thing is that SNMP-compliant devices support the logical idea of a MIB. All MIBs fit into an overall hierarchy which can be represented with a dotted number notation, such as 1.3.6.1.2.1.1.4.0.

SNMP also uses the idea of community names. Different devices can be assigned to different communities (logical groupings). A common community name is "public." A community name can serve as a kind of password.

SNMP commands specify Object IDs (OIDs) when reading or writing information to SNMP devices. An OID can be a dotted numeric reference to a MIB field, or it can be a logical name. Some logical names are listed in Table 27.1.

The use of object names simplifies MIB specifications.

Network Monitoring Operations

Network Query Language contains features that allow scripts to act as SNMP managers, discovering, interrogating, and controlling network devices. The NQL SNMP statements are summarized in Table 27.2.

Some of the network monitoring statements are also available as functions. Table 27.3 lists the HTML pattern-matching functions.

The network monitoring statements and functions are described individually in the following sections.

Pinging a Network Device

The *ping* statement pings a network device, accessing an IP address to determine its availability and response time. The parameter is a system name or IP address.

Table 27.1 Object Names

NAME	DESCRIPTION
system	same as 1.3.6.1.2.1.1
interfaces	same as 1.3.6.1.2.1.2
at	same as 1.3.6.1.2.1.3
ip	same as 1.3.6.1.2.1.4
icmp	same as 1.3.6.1.2.1.5
tcp	same as 1.3.6.1.2.1.6
udp	same as 1.3.6.1.2.1.7
egp	same as 1.3.6.1.2.1.8
snmp	same as 1.3.6.1.2.1.11

Table 27.2 Network Monitoring Statements

STATEMENT	PARAMETERS	DESCRIPTION
ping	*address*	Accesses a remote TCP/IP address to check availability and response time
pingoptions	*timeout*, *packet-count*, *packet-delay*	Sets options for the *ping* statement
snmpdiscover	*ip-range*, *community*	Scans a range of IP addresses to discover responding devices
snmpget	*address*, *community*, *OID*	Retrieves information from a network device
snmpoptions	*timeout*, *retries*, *polling-interval*, *poll-count*	Sets options for the *snmp* statements
snmpset	*address*, *community*, *OID, value*	Sets a network device value
snmpwalk		Retrieves the next information field from a network device

Table 27.3 Network Monitoring Functions

FUNCTION	DESCRIPTION
boolean ping(*address*)	Accesses a remote TCP/IP address to check availability and response time
boolean snmpdiscover(*ip-range*, *community*)	Scans a range of IP addresses to discover responding devices
boolean snmpget(*address*, *community*, *OID*)	Retrieves information from a network device
boolean snmpoptions(*timeout*, *retries*, *polling-interval*, *poll-count*)	Sets options for the *snmp* statements
boolean snmpset(*address*, *community*, *OID, value*)	Sets a network device value
boolean snmpwalk( )	Retrieves the next information field from a network device

```
ping     address
boolean ping(address)
```

If the network device does not respond, a failure condition results (the function returns false). If the network device does respond, a success condition results (the function returns true) and two variables are created automatically: *ResolvedIP* and

PingTotalTime. *ResolvedIP* contains the IP address that was accessed; *PingTotalTime* contains the access time, in milliseconds. The following code shows the use of *ping*.

```
ping "www.nqli.com"
then
{
    show "the site is down"
}
else
{
    show "the site is up"
}
dumpvars
```

The period of time *ping* is willing to wait is affected by its options. The *pingoptions* statement may be used to amend these options.

Setting Ping Options

The *pingoptions* statement changes the settings used by the *ping* statement. The parameters are a time-out interval (in milliseconds), the number of packets to wait for (two or more), and a packet delay between echo tries (in milliseconds).

```
pingoptions timeout, packet-count, packet-delay
```

The following code shows the use of *pingoptions*.

```
pingoptions 60, 5, 10
ping sURL
```

The NQL default values for *ping* are 200, 3, and 1000.

Locating Network Devices

The *snmpdiscover* statement or function scans a range of IP addresses and returns a list of devices that respond. This makes it useful for discovery of available network devices. The parameters are an IP range and a community name.

```
snmpdiscover ip-range, community
boolean snmpdiscover(ip-range, community)
```

IP ranges may be phrased in two ways. The first format, "n.n.n.n-n.n.n.n," is two full IP addresses separated by a hyphen, such as "172.16.100.1-172.16.100.255." The second format, "n.n.n.n-n," assumes the first three IP numbers are the same for the starting and ending values of the range, as in "172.16.100.1-255."

If the parameters are accepted and a network scan is successful, a success condition results (the function returns true) and a list of responding addresses is pushed onto the stack as a newline-separated string. The following code shows the use of *snmpdiscover* to locate and display all responding network devices within an IP range.

```
snmpdiscover "172.16.100.1-255", "public"
nextline
while
{
    pop address
    show address
    nextline
}
```

Retrieving Values from a Network Device

The *snmpget* statement retrieves a value from a network device. The parameters are an IP address, a community name, and an OID. Community names and object identifiers are part of the SNMP organizational system. A commonly used community name is “public.” The object identifier is either a dotted formal OID number such as “1.3.6.1.2.1.1.4.0” or an MIB node name such as “system.”

```
snmpget address, community, OID
boolean snmpget(address, community, OID)
```

If the requested information is successfully received from the specified device, a success condition results (the function returns true) and two variables are created automatically: *MIBName* and *MIBValue*. *MIBName* contains the MIB field name; *MIBValue* contains the field value. The following code uses *snmpget* to retrieve a value from a network device.

```
snmpget "172.16.100.10", "public", "system.sysContact.0"
show MIBName, MIBValue
```

Once an *snmpget* statement has been issued, an easy way to traverse the subordinate fields of the MIB is to use the *snmpwalk* statement.

Walking Through MIB Values

The *snmpwalk* statement retrieves a value from a network device, picking up from where the last *snmpget* or *snmpwalk* statement left off. There are no parameters.

```
snmpwalk
boolean snmpwalk()
```

If the requested information is successfully received from the specified device, a success condition results (the function returns true) and two variables are created automatically: *MIBName* and *MIBValue*. *MIBName* contains the MIB field name; *MIBValue* contains the field value. The following code uses *snmpwalk* to retrieve values from network devices.

```
snmpget "172.16.100.10", "public", "system.sysContact.0"
show MIBName, MIBValue
snmpwalk
show MIBName, MIBValue
snmpwalk
show MIBName, MIBValue
```

Setting Network Device Values

The *snmpset* statement sets a network device value. The parameters are an IP address, a community name, an OID, and a value. Community names and object identifiers are part of the SNMP organizational system. A commonly used community name is "public." The object identifier is either a dotted formal OID number such as "1.3.6.1.2.1.1.4.0" or an MIB node name such as "system."

```
snmpset address, community, OID, value
boolean snmpset(address, community, OID, value)
```

If the field value is successfully passed to the specified device, a success condition results (the function returns true) and two variables are created automatically: *MIBName* and *MIBValue*. *MIBName* contains the MIB field name; *MIBValue* contains the field value. The following code uses *snmpset* to set a network device value.

```
snmpset "172.16.100.10", "public", "system.sysContact.0", "jsmith"
show MIBName, MIBValue
```

Setting SNMP Options

The *snmpoptions* statement sets options that affect the SNMP statements. The parameters are a timeout interval (in milliseconds), a retry count, a polling interval (in milliseconds), and a poll count.

```
snmpoptions timeout, retries, polling-interval, poll-count
```

The following code shows the use of *snmpoptions*.

```
snmpoptions 60, 10, 10, 5
```

The NQL default *snmpoptions* values are 200, 1, 15, and 4.

Tutorial: Discover

Let's apply the SNMP statements in a script. In this tutorial, you will create a script that scans a network, searching for devices that respond to SNMP inquiries. The result will be a Web page listing the responding devices, such as the one shown in Figure 27.1.

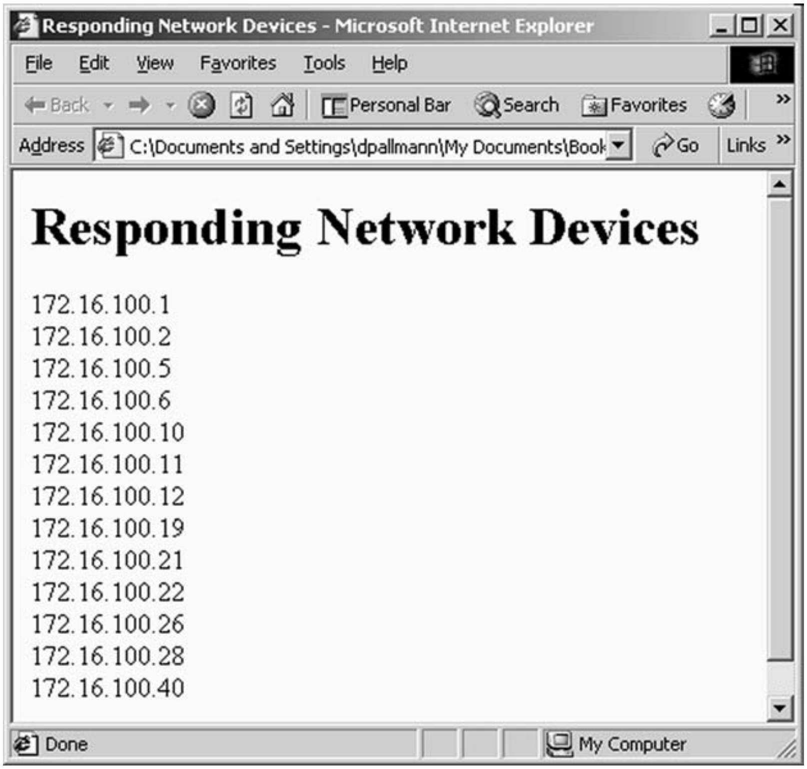
There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Discover script.
3. Run the Discover script.
4. Understand the Discover script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Scan the SNMP devices on a network.

Let's begin!


Responding Network Devices - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Address C:\Documents and Settings\dpallmann\My Documents\Book
Go Links
Done
My Computer

Responding Network Devices

172.16.100.1
172.16.100.2
172.16.100.5
172.16.100.6
172.16.100.10
172.16.100.11
172.16.100.12
172.16.100.19
172.16.100.21
172.16.100.22
172.16.100.26
172.16.100.28
172.16.100.40

Figure 27.1 Web page created by the Discover script.

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the Discover Script

In the NQL development environment, enter the script shown in Listing 27.1 and save it under the name Discover.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Discover.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial

```
//Discover - lists responding network devices in a web page.

create "discover.htm"

write "<HTML>
<HEAD>
<TITLE>Responding Network Devices</TITLE>
<BODY>
<H1>Responding Network Devices</H1>
"

if snmpdiscover ("172.16.100.1-120", "public")
{
    while nextline()
    {
        pop ip
        write ip "<BR>"
    }
}

write "<P>
</BODY>
</HTML>"
close
opendoc "discover.htm"
```

Listing 27.1 Discover script.

materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Discover.nql from the Tutorials\ch27 folder.

You will need to change the network range and community name to match your network.

At this point, the script in Listing 27.1 should be in your code window, either because you entered it by hand, or because you opened the script from the companion CD. And, you should have replaced the network range and community name with appropriate values for your environment. You are now ready to run the script.

Step 3: Run the Discover Script

Run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. When the script completes, a Web page opens on the desktop showing the captured network IP addresses. If the script does not seem to be working, check the following:

- Check for typographical errors in your script.
- Check that you are using a valid IP range and community name.
- Check the NQL client's Errors tab for error messages.

At this point you should have seen the Discover script run, polling network devices and outputting the responding IP addresses to a Web page.

Step 4: Understand the Discover Script

We now want to make sure we understand every part of the Discover script. The first line is a comment line.

```
//Discover - lists responding network devices in a web page.
```

The Web page is created with *create*. A *write* statement outputs the initial HTML for the page.

```
create "discover.htm"

write "<HTML>
<HEAD>
<TITLE>Responding Network Devices</TITLE>
<BODY>
<H1>Responding Network Devices</H1>
"
```

The network is polled with an *snmpdiscover* statement. As shown, IP addresses in the range 172.16.100.1 through 172.16.100.120 are scanned for an SNMP response if they belong to the community "public." The result is a string value on the stack that consists of IP addresses of responding systems, separated by newlines.

```
if snmpdiscover ("172.16.100.1-120", "public")
{
```

A *while* loop and the *newline* function iterate through each line (address) in the results. In the body of the *while* loop, each IP address is popped into the variable *ip* and output to the Web page output file. The loop continues until there are no more IP addresses to process.

```
    while nextline()
    {
        pop ip
        write ip "<BR>"
    }
}
```

A *write* statement writes out the final HTML of the page. The file is closed, then displayed on the desktop with *opendoc*, ending the script.

```
write "<P>
</BODY>
</HTML>"
close
opendoc "discover.htm"
```

At this point, you've seen how to scan network devices via SNMP in an NQL script.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Do more than discover network devices: Capture information from them using snmpget. In order to do this, you'll need to know what to expect from various devices on your network.
- Amend the script to ping the devices that are discovered and record how long it takes them to respond.
- Interrogate each device as to its operational status. In the generated Web page, reference images of green or red lights to reflect how well each device is functioning.

Chapter Summary

Network Query Language supports Simple Network Management Protocol (SNMP), permitting scripts to monitor and control network devices.

- The *ping* statement accesses an IP address to check availability and response time.
- The *pingoptions* statement changes time-out and other settings for the *ping* statement.
- The *snmpdiscover* statement scans a range of IP addresses to discover responding devices.
- The *snmpget* statement retrieves information from a network device.
- The *snmpoptions* statement sets options for the SNMP statements.
- The *snmpset* statement sets a network device value.
- The *snmpwalk* statement retrieves the next information field from a network device.

SNMP support in NQL allows your scripts to sense, monitor, and control many kinds of devices on the network.

CHAPTER

28

Web Applications

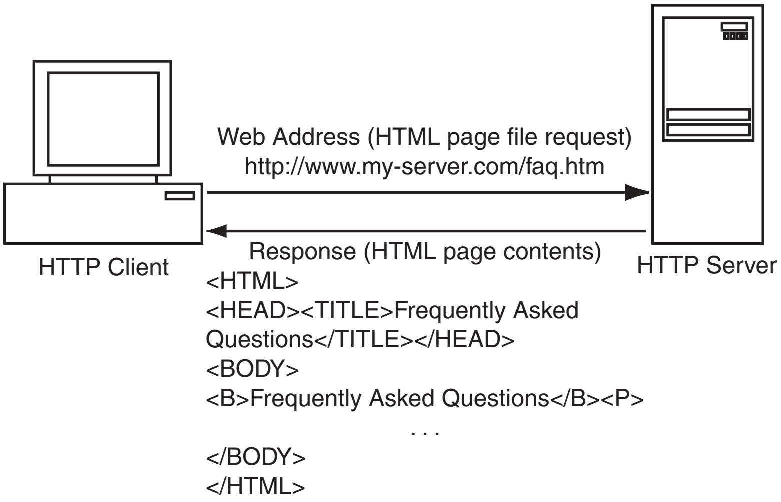
Network Query Language scripts can be run in many contexts, such as the desktop, on servers, as middleware, and as Web applications. As Web applications, NQL scripts play a role similar to that of Perl scripts, Active Server Pages scripts, and Java Server Pages scripts. Web requests cause scripts to run and the script responses are usually HTML pages. This chapter explains how to write Web applications in NQL. The tutorial at the end of the chapter creates a Web application in NQL that provides a custom interface for searching a university library.

Background on Web Applications

Web applications are based on a client/server model, in which a client (such as a Web browser) makes a request of a Web server. In the early days of the World Wide Web, requests were primarily the names of static HTML pages that the Web server would return to the client verbatim. Figure 28.1 illustrates a static request and a response.

Over the years, dynamic Web sites powered by applications have become common. A Web application follows the same client/server model, but when a request comes in to the Web server, a script executes and dynamically decides what response to send back to the client. Figure 28.2 shows a dynamic request to a Web application.

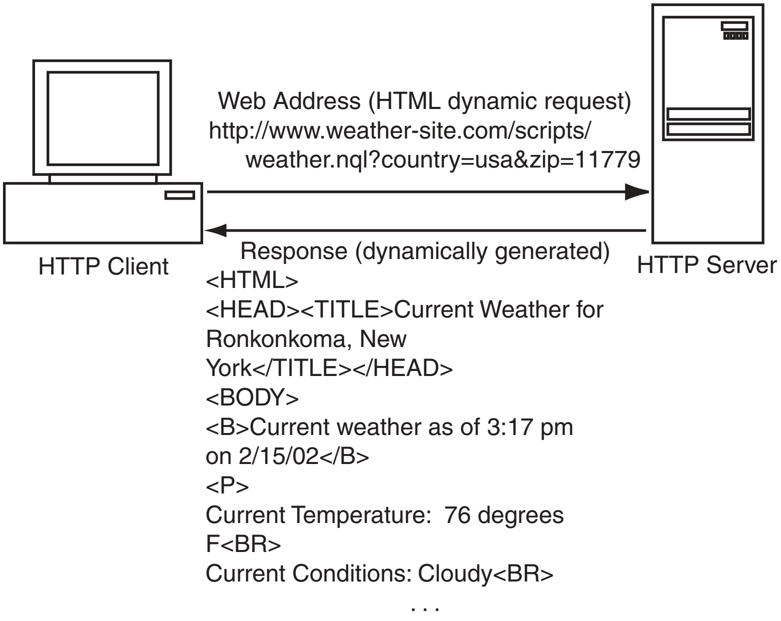
The input to a Web server is a Uniform Resource Locator (URL), also known as a Web address. Web addresses generally contain some or all of the following parts: a


Web Address (HTML page file request)
http://www.my-server.com/faq.htm
Response (HTML page contents)
HTTP Client
HTTP Server

```
<HTML>
<HEAD><TITLE>Frequently Asked
Questions</TITLE></HEAD>
<BODY>
<B>Frequently Asked Questions</B><P>
                    . . .
</BODY>
</HTML>
```

Figure 28.1 Static Web request.

protocol identifier (http:// or https://), a server name (such as www.wiley.com), a path (such as /scripts/), and a filename (such as default.htm). Some standard URLs are shown.


Web Address (HTML dynamic request)
http://www.weather-site.com/scripts/
weather.nql?country=usa&zip=11779
Response (dynamically generated)
HTTP Client
HTTP Server

```
<HTML>
<HEAD><TITLE>Current Weather for
Ronkonkoma, New
York</TITLE></HEAD>
<BODY>
<B>Current weather as of 3:17 pm
on 2/15/02</B>
<P>
Current Temperature:  76 degrees
F<BR>
Current Conditions: Cloudy<BR>
                    . . .
```

Figure 28.2 Dynamic Web request.

```
http://www.my-news-site.com
http://store.my-retail-site.com/products/1033.htm
https://secure.my-bank.com/customers/ca/securelogon.html
```

When a Web application is being accessed, the filename references a program file rather than an HTML page, and there is more to the URL. After the filename comes a question mark, and some number of parameters. Each parameter is of the format *name=value*, and multiple parameters are separated from each other by ampersands. Some sample dynamic URLs are shown.

```
http://www.my-company.com/scripts/purchase.dll?prodid=1035
http://www.my-federal.gov/publications/application.pl?pubno=562&state=NY
http://www.my-taxes.com/compute.asp?subtotal=2500.00&taxrate=0.75&state=MA
```

In Web application URLs, the file name indicates a type of application. In the preceding examples, DLL means a Windows Dynamic Link Library (DLL), PL means a Perl script, and ASP means an Active Server Pages script. When you program Web applications in NQL, the same principle holds true. Web addresses contain the name of an NQL script to execute, such as the following URL.

```
http://www.my-spider.com/search/search.nql?query=camping&page=1
```

Web application scripts produce output in a content language, most commonly HTML, the standard language of Web pages. In addition to HTML, Web applications might also produce XML, the newer standard for Internet content, or content formatted for various mobile devices such as PDAs and Internet phones.

Web Programming in NQL

An NQL script that is also a Web application can employ the usual NQL statements and functions, but must also concern itself with inputs (parameters from the URL) and outputs (the response to the Web browser or client). NQL has specific statements and functions to aid in Web programming.

The NQL Web programming statements are summarized in Table 28.1.

The Web programming statements are described individually in the following sections.

Retrieving Input

The input to a Web application script is the parameter(s) in the URL. In the context of a Web application, the *input* statement retrieves Web parameters into variables. For example, consider the following Web address:

```
http://www.my-server.com/scripts/findperson.nql?ss=091-62-
1958&bday=02/03/61
```

Table 28.1 Web Programming Statements

STATEMENT	PARAMETERS	DESCRIPTION
closewebsession		Closes a Web session
input	*var-list*	Retrieves parameters from the URL
openwebsession	*minutes*	Opens a Web session
output	*value*	Outputs response data to the browser
websessionclear	[*minutes*]	Clears some or all Web sessions
websessiongetcookie	*cookie-name*, *var*	Retrieves a cookie from the client system
websessionlist		Reports session information
websessionload	*var-list*	Retrieves one or more session variables
websessionsave	*var-list*	Sets one or more session variables
websessionsetcookie	*cookie-name*, *value*	Sets a cookie on the client system

The parameters in this URL are named *ss* and *bday*. To retrieve them in an NQL script, the *input* statement is specified with these same names.

```
input ss, bday
```

The result is that variables named *ss* and *bday* are available to the script, containing the values specified in the URL.

Generating Output

In the context of a Web application, an NQL script can generate response data (to be sent to the Web browser or client) using the *output* statement. For example, the following code is a complete NQL Web application that reports the time of day to a Web browser.

```
sNow = time()
output "<HTML><BODY>"
output "The current time is " sNow
output "</BODY></HTML>"
end
```

If accessed at 9:15 A.M., the preceding script produces the following response to a Web browser:

```
The current time is 9:15:00
```

For large amounts of HTML, the use of multiple *output* statements can be inconvenient. An alternative way to generate HTML output is to use the <% and %> characters, which switch you between NQL and HTML code in your script. For example, the script we just looked at can be rewritten as follows, and is functionally identical.

```
sNow = time()
<%HTML><BODY>
The current time is {sNow}
BODY></HTML>%>
end
```

Notice in the preceding example how the time, *sNow*, is output. When you are in HTML output mode (within the <% and %> symbols), you may embed variable values by specifying their names within curly braces. Text such as the following will be substituted with current data when the script responds to a Web request.

```
<%
Your customer ID is {CustID} and your representative is {CustRep}.
%>
```

For example, if the variable *CustID* contained 104545 and *CustRep* contained "Kevin Elofson", the preceding line would generate the following to a Web browser:

```
Your customer ID is 104545 and your representative is Kevin Elofson.
```

Cookies

Web applications can save information on users' computers. This allows your applications to remember that a user visited you previously, or what someone's preferences are. Network Query Language scripts can set cookies and can read them back. A cookie is nothing more than a field with a name and a text value, for instance name *ID* and value "104545".

Setting a Cookie

The *websessionsetcookie* statement sets a cookie on the client system. The parameters are a cookie name and a value. Cookies have unique names and you may set as many as you like.

```
websessionsetcookie cookie-name, value
```

The cookie is included in the response that the script generates, and the client's application (usually a Web browser) is responsible for saving the cookie on the client computer. When future accesses are made to your Web server, the cookie is available and can be read with the *websessiongetcookie* statement. The following code sets a cookie.

```
CookieName = "UserID"
CookieValue = "JKQ78711"
websessionsetcookie CookieName, CookieValue
<%
<HTML>
...
</HTML>
%>
```

The value parameter can contain just a cookie value or may contain additional information that controls the lifetime of the cookie. The value may be followed by standard cookie fields such as those shown in Table 28.2. The value and each of the cookie fields are separated by semicolons.

Bear in mind that the supported cookie fields, their format, and their implementation are all dependent on the Web browser software at the client side. A sample cookie value with cookie fields is shown.

```
websessionsetcookie "mycookie", "myvalue; expires=Mon, 18-Sep-2002
18:30:26 GMT; path=/; domain=mydomain.com; secure"
```

The preceding statement sets a cookie named *mycookie* with value *myvalue*. The cookie expires at a specific time and date, is issued for all URL paths (for this Web server), reports a specific domain, and is only to be issued over a secure connection.

When you don't specify any cookie fields, as in the following code, the default conditions are that the cookie will expire when the client closes the Web browser.

```
websessionsetcookie "mycookie", "myvalue"
```

Retrieving a Cookie

Once you have created a cookie for a user, you can retrieve it with the *websessiongetcookie* statement. The parameters are a cookie name and a variable to receive the value.

Table 28.2 Cookie Fields

FIELD NAME	DESCRIPTION	EXAMPLE
expires	Indicates the expiration date/time of the cookie	expires=Mon, 18-Sep-2002 18:30:26 GMT
path	Indicates the Web server path the cookie applies to	path=/
domain	Indicates the domain the cookie applies to	domain=mydomain.com
secure	Indicates the cookie is only to be used over a secure connection	secure

```
websessiongetcookie cookie-name, var
```

The variable is set to the value of the specified cookie. If no such cookie exists, or if it has expired, the value returned will be an empty string. The following code shows the use of *websessiongetcookie*.

```
CookieName = "UserID"
websessiongetcookie CookieName, CookieValue
show CookieValue
```

Web Sessions

Web applications often need to preserve information between accesses. For example, a retail Web site needs to remember selected items in a shopping cart. One method for saving information between Web accesses is cookies; another is Web session variables. The difference is that cookies are stored on disk on the client's computer, while session variables are stored in memory on the Web server. Another difference is that cookies can last a long time or be permanent, whereas session variables last only for the duration of the Web session. Cookies and session variables may be combined if desired.

A Web session is specific to a single user. They allow your scripts to serve many simultaneous users, yet keep separate variable data for each user across multiple accesses to your Web site.

Opening a Web Session

A Web session is opened with *openwebsession*. The parameter is a timeout interval in minutes.

```
openwebsession 15
```

The Web session begins for the current user. The Web session will be removed from the server after the specified time, or if a script executes a *closewebsession* statement. The following code opens a Web session for 20 minutes.

```
openwebsession 20
```

Since the intended function of Web sessions is to preserve state between multiple Web accesses, the script that specifies *openwebsession* and the script that specifies *closewebsession* don't have to be the same (and probably would not be).

Setting Session Variables

If a script wishes to set session variables (to be retrieved from another script later in the Web session), *websessionsave* is used. The parameters are a list of variables.

```
websessionsave var-list
```

Each specified variable is saved for later retrieval. The following code saves a user ID and password as session variables using *websessionsave*.

```
openwebsession 15
<%<HTML><HEAD></HEAD><BODY>%>
input userid, password
if ...valid login...
{
    websessionsave userid, password
    <%Logged in successfully.<P>%>
}
else
{
    <%Invalid login, try again.<P>%>
}
...
<%</BODY></HTML>%>
```

Once variables are saved in a Web session, they can be retrieved until the Web session times out or is explicitly closed.

Retrieving Session Variables

Session variables are retrieved with the *websessionload* statement. The parameters are a list of variables.

```
websessionload var-list
```

Each specified variable is retrieved (or is set to an empty value if the variable had not been stored previously by *websessionsave*). The following code retrieves a user ID and password from session variables using *websessionload*.

```
openwebsession 15
<%<HTML><HEAD></HEAD><BODY>%>
input userid, password
websessionload userid, password
<%Welcome back, {userid}!<P>%>
...
<%</BODY></HTML>%>
```

Closing a Web Session

The *websessionclose* statement terminates a Web session, something you might want to do if a user logs off your Web site in some way, or if processing has come to a logical conclusion such as completion of an online order. There are no parameters.

```
websessionclose
```

The Web session is terminated, and any session variables are lost. The following code closes a Web session with *websessionclose*.

```
<%<HTML>
<HEAD><TITLE>Thank you for your order</TITLE>
</HEAD>
<BODY>
Thank you for your order. We value your business.
<P>
</BODY>
</HTML>%>
websessionclose
end
```

Retrieving Session Information

The *websessionlist* statement returns information about the current Web session. There are no parameters.

```
websessionlist
```

A string is pushed onto the stack that contains the following values, separated by newlines: the session ID, the length of stored variables, and the number of minutes since the session was last accessed. The following code uses *websessionlist* to determine session information.

```
websessionlist
nextline sSessionID
nextline nVarLength
nextline nMinutes
```

Clearing Web Sessions

The *websessionclear* statement clears Web sessions (server-wide). The optional parameter is an age in minutes.

```
websessionclear [minutes]
```

All Web sessions on the server that are more than the specified number of minutes old are cleared (the sessions are closed and session variables are no longer stored). If no time parameter was specified, all Web sessions on the server are cleared. The following code clears all Web sessions more than 30 minutes old.

```
websessionclear 30
```

Tutorial: Library Search

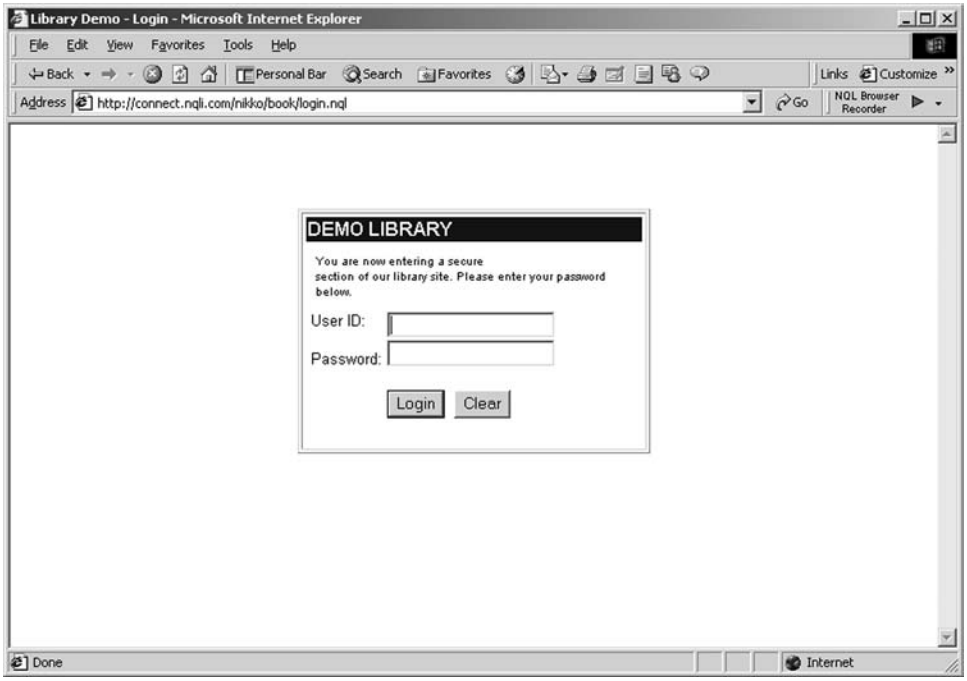
Let's create a Web application in NQL to put all of this into practice. In this tutorial, you will create a Web interface that requires a user to log in, then front-ends a search of a university library Web site. This tutorial will demonstrate the use of NQL as a Web application, and will make use of Web sessions and cookies.

There will be three distinct parts to Library Search: a login page, a search page, and a results page. The login page is shown in Figure 28.3. The user will have to enter a valid ID and password in order to progress through the site. For this tutorial, *nql* is a valid ID and password.

The second page is the search page. If the user logs in with a valid ID and password, the page in Figure 28.4 is displayed. This is a search form, allowing the user to enter search text and indicate if it is a keyword, author, or title. When the user clicks the *Search* button, the script will submit a university library search to another Web site behind the scenes.

The third page is the results page. The results from the library search are displayed on this page, which is shown in Figure 28.5.

In order to run this tutorial, you will need access to a Web server, and NQL needs to be installed on that Web server. For example, if you are using the Windows edition of NQL, a Windows NT/2000/XP server with IIS may be used. When installing NQL on



Library Demo - Login - Microsoft Internet Explorer

Address http://connect.nqli.com/nikko/book/login.nql

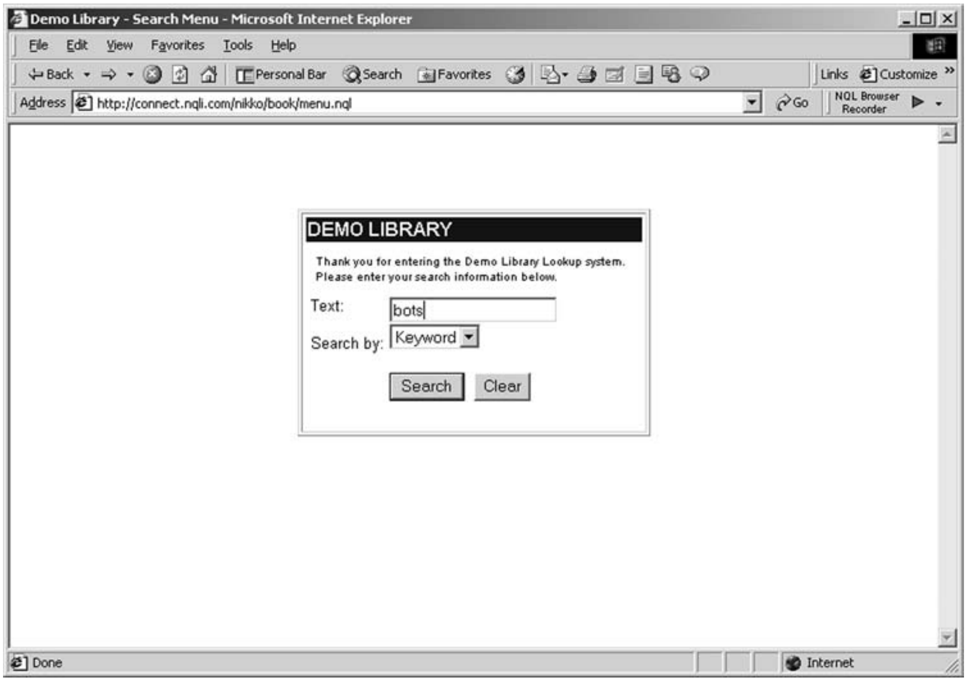
DEMO LIBRARY

You are now entering a secure section of our library site. Please enter your password below.

User ID:

Password:

Figure 28.3 Login page.


Demo Library - Search Menu - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites Links Customize
Address http://connect.nqli.com/nikko/book/menu.nql
Go NQL Browser Recorder
Search Clear
Done
Internet

DEMO LIBRARY

Thank you for entering the Demo Library Lookup system. Please enter your search information below.

Text: bots

Search by: Keyword

Figure 28.4 Search page.

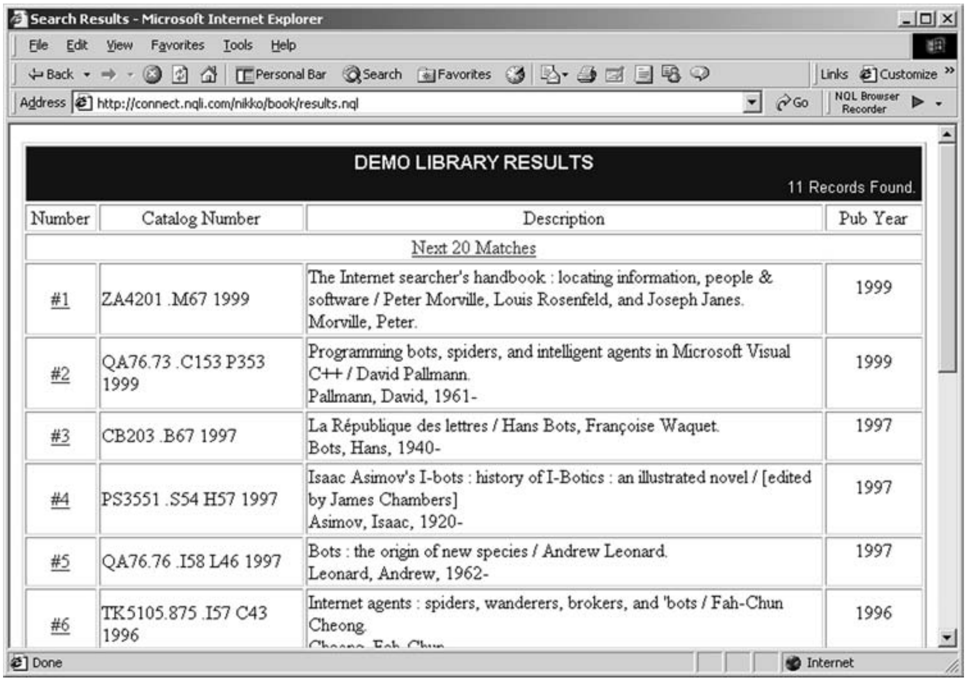
the server, the installation program may ask if you want the IIS settings configured for NQL as a Web application. If so, respond in the affirmative.

There are eight steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Login Page script.
3. Enter the Search Page script.
4. Enter the Results Page script.
5. Access the Library Search application.
6. Understand the Login Page script.
7. Understand the Search Page script.
8. Understand the Results Page script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Create a Web application in NQL.
- Use cookies.
- Maintain state in a Web session.


Search Results - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Personal Bar Search Favorites
Links Customize
Address http://connect.nqli.com/nikko/book/results.nql
Go
NQL Browser Recorder
Done
Internet

DEMO LIBRARY RESULTS

11 Records Found.

Number	Catalog Number	Description	Pub Year
		Next 20 Matches	
#1	ZA4201 .M67 1999	The Internet searcher's handbook : locating information, people & software / Peter Morville, Louis Rosenfeld, and Joseph Janes. Morville, Peter.	1999
#2	QA76.73 .C153 P353 1999	Programming bots, spiders, and intelligent agents in Microsoft Visual C++ / David Pallmann. Pallmann, David, 1961-	1999
#3	CB203 .B67 1997	La République des lettres / Hans Bots, Françoise Waquet. Bots, Hans, 1940-	1997
#4	PS3551 .S54 H57 1997	Isaac Asimov's I-bots : history of I-Botics : an illustrated novel / [edited by James Chambers] Asimov, Isaac, 1920-	1997
#5	QA76.76 .I58 L46 1997	Bots : the origin of new species / Andrew Leonard. Leonard, Andrew, 1962-	1997
#6	TK5105.875 .I57 C43 1996	Internet agents : spiders, wanderers, brokers, and 'bots / Fah-Chun Cheong.	1996

Figure 28.5 Results page.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

You will be entering three scripts, so either launch the development environment three times, or start a new script with *File, New* as you follow steps 2, 3, and 4.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter your three NQL scripts.

Step 2: Enter the Login Page Script

The first script we will enter is the login page for the Library Search application. In the NQL development environment, enter the script shown in Listing 28.1 and save it under the name login.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//login - Login page of the Library Search application.

websessionload Error

websessiongetcookie "loggedin", sLoggedIn
if sLoggedIn != ""
{
    websessionredirect "menu.nql"
}

<%
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR"
content="Microsoft FrontPage 4.0">
<title>Library Demo - Login</title>
</head>

<body bgcolor="#FFFFFF" vlink="#0000FF" alink="#0000FF">

<p align="center"><br>
<br>
<br>
</p>
<div align="center"><center>

<table border="1" width="300">
    <tr>
        <td align="center"><table border="0" width="100%">
            <tr>
                <td bgcolor="#000080"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY</b>
                </font></td>
            </tr>
            <tr>
                <td><table border="0" cellpadding="5"
                width="100%">
                    <tr>
                        <td valign="top"><font size="1"
                        face="ARIAL, HELVETICA">You are now
                        entering a secure <br>
                        section of our library site. Please
```

continues

Listing 28.1 Login page script.

```
                        enter your password below. </font></td>

                    </tr>
                </table>
             %>
                if Error = true
                {
                    output '<font face="Arial" color="#FF0000"
size="2"><i>UserID or Password incorrect, please try again</i></font>
<br><br>'
                    Error = false
                    websessionsave Error
                }
            <%

                <table border="0">
                    <tr>
                        <td valign="top">


                <font size="2"
                        face="ARIAL, HELVETICA">User ID:<br>
                        <br>
                        Password:</font></td>
                        <td valign="top"><form name=loginForm action=
"menu.nql"
                        method="POST">
                <p><input type="text" size="20"
                            name="UserID"><br>
                            <input type="password" size="20"
                            name="Password"></p>
                            <p><input type="submit" name="B1"
                            value=”Login” tabindex="1">&nbsp;&nbsp;
<input type="reset"
                            name="B2" value="Clear" tabindex="2">
                            <br>
                            </p>
                        </form>
                        </td>
                    </tr>
                </table>
                </td>
            </tr>
        </table>
        </td>
    </tr>
</table>
```

Listing 28.1 Login page script (continued).

```
</center></div>

<script language="javascript">
    document.loginForm.UserID.focus();
</script>

<p>&nbsp;</p>

<p>&nbsp;</p>
</body>
</html>
%>
```

Listing 28.1 Login page script (continued).

If you prefer, you may copy login.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select login.nql from the Tutorials\ch28 folder.

You need to copy this and the other two scripts we will create onto a Web server in a publicly accessible directory, such as c:\InetPub\Scripts in the case of a Windows Web server.

At this point, the script in Listing 28.1 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. The script should be saved on disk as login.nql. The script should be placed on a Web server.

Step 3: Enter the Search Page Script

The second script we will enter is the search page for the Library Search application. In the NQL development environment, enter the script shown in Listing 28.2 and save it under the name menu.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy menu.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select menu.nql from the Tutorials\ch28 folder.

You need to copy this and the other scripts we will create onto a Web server in a publicly accessible directory, such as c:\InetPub\Scripts in the case of a Windows Web server.

At this point, the script in Listing 28.2 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. The script should be saved on disk as menu.nql.

```
//menu - Search page of the Library Search application.

input UserID, Password

Error = False

websessiongetcookie "loggedin", sLoggedIn
if sLoggedIn = ""
{
    if UserID == "nql"
    {
        if Password != "nql"
        {
            Error = true
            websessionsave Error
            websessionredirect "login.nql"
        }
    }
    else
    {
        Error = true
        websessionsave Error
        websessionredirect "login.nql"
    }
    date sDate
    call FormatDate
    sCookieDate = sDayofWeek & ", " & sDay & "-" & sMonth & "-" &
sYear

    sCookie = "yes; expires=" & sCookieDate & " 23:59:59 GMT; path=/;
domain=nqli.com"

    websessionsetcookie "loggedin", sCookie
    websessiongetcookie "loggedin", sLoggedIn

}
<%
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR"
content="Microsoft FrontPage 4.0">
<title>Demo Library - Search Menu</title>
/</head>
```

Listing 28.2 Search page script.

```
<body bgcolor="#FFFFFF" vlink="#0000FF" alink="#0000FF">
<p align="center"><br>
<br>
<br>
</p>
<div align="center"><center>

<table border="1" width="300">
    <tr>
        <td align="center"><table border="0" width="100%">
            <tr>
                <td bgcolor="#000080"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY</b>
                </font></td>
            </tr>
            <tr>
                <td><table border="0" cellpadding="5"
                width="100%">
                    <tr>
                        <td valign="top"><font size="1" face="ARIAL,
HELVETICA">Thank
                          you for entering the Demo Library Lookup
system.
                          Please enter your search information
below.</font></td>

                    </tr>
                </table>
                <table border="0">
                    <tr>
                        <td valign="top"><font size="2"
                        face="ARIAL, HELVETICA">Text:<br>
                        <br>
                          Search by:</font></td>
                        <td valign="top"><form name=loginForm
action="results.nql"
                        method="POST">
                            <p><input type="text" size="20"
                            name="terms"><br>
                            <select size="1" name="type">
                              <option value="keyword">Keyword</option>
                              <option value="author">Author</option>
                              <option value="title">Title</option>
                            </select></p>
```

continues

Listing 28.2 Search page script.

```
                    <!--<p><input type="radio" name="searchtype"
value="standard"> Standard <input type="radio" name="searchtype"
value="complex"> Complex-->
                            <p><input type="submit" name="B1"
value="Search">&nbsp;&nbsp;<input type="reset" name="B2"
value="Clear">
                            <br>
                            </p>
                        </form>
                        </td>
                    </tr>
                </table>
                </td>
            </tr>
        </table>
        </td>
    </tr>
</table>
</center></div>

<p>&nbsp;</p>

<p>&nbsp;</p>
</body>
</html>
%>
end

FormatDate:
    push sDate
    parsedate intMonth, sDay, sYear, intDayofWeek
    if sDay < 10
    { sDay = "0" & sDay }

    if intMonth = "1"
    { sMonth = "Jan" }
    if intMonth = "2"
    { sMonth = "Feb" }
    if intMonth = "3"
    { sMonth = "Mar" }
    if intMonth = "4"
    { sMonth = "Apr" }
    if intMonth = "5"
    { sMonth = "May" }
    if intMonth = "6"
    { sMonth = "Jun" }
    if intMonth = "7"
```

Listing 28.2 Search page script (continued).

```
      { sMonth = "Jul" }
      if intMonth = "8"
      { sMonth = "Aug" }
      if intMonth = "9"
      { sMonth = "Sep" }
      if intMonth = "10"
      { sMonth = "Oct" }
      if intMonth = "11"
      { sMonth = "Nov" }
      if intMonth = "12"
      { sMonth = "Dec" }

      if intDayofWeek = "1"
      { sDayofWeek = "Sun" }
      if intDayofWeek = "2"
      { sDayofWeek = "Mon" }
      if intDayofWeek = "3"
      { sDayofWeek = "Tue" }
      if intDayofWeek = "4"
      { sDayofWeek = "Wed" }
      if intDayofWeek = "5"
      { sDayofWeek = "Thu" }
      if intDayofWeek = "6"
      { sDayofWeek = "Fri" }
      if intDayofWeek = "7"
      { sDayofWeek = "Sat" }
      return
```

Listing 28.2 Search page script (continued).

Step 4: Enter the Results Page Script

The third script we will enter is the results page for the Library Search application. In the NQL development environment, enter the script shown in Listing 28.3 and save it under the name results.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy results.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select results.nql from the Tutorials\ch28 folder.

You need to copy this script onto a Web server in a publicly accessible directory, such as c:\InetPub\Scripts in the case of a Windows Web server.

At this point, the script in Listing 28.3 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. The script should be saved on disk as results.nql.

You should now have created or copied all three scripts onto a Web server directory: login.nql, menu.nql, and results.nql. In the next step you will run the Web application.

```
//results - Results page of the Library Search application.

input terms, type, searchtype, firsthit, lasthit

openwebsession

//handle search by keyword

if type = "keyword"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5EGENERAL%
5ESUBJECT%5EGENERAL%5E%5Ewords%2Bor%2Bphrase.x=80&S_ICON%5EGENERAL%
5ESUBJECT%5EGENERAL%5E%5Ewords%2Bor%2Bphrase.y=4"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL ="http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y
=10"

    }
}

//handle search by author

if type = "author"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5EAU%
```

Listing 28.3 Results page script.

```
5EAUTHOR%5EAUTHORS%5EAuthor+Processing%5Eauthor.x=44&S_ICON%5EAU%
5EAUTHOR%5EAUTHORS%5EAuthor+Processing%5Eauthor.y=21"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y
=10"

    }

}

//handle search by title

if type = "title"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5ETI%
5ETITLE%5ESERIES%5ETitle+Processing%5Etitle.x=53&S_ICON%5ETI%5ETITLE%
5ESERIES%5ETitle+Processing%5Etitle.y=22"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y
=10"

    }
}

//submit search to web site and retrieve results
```

continues

Listing 28.3 Results page script (continued).

```
eval sURL
get sURL

//extract a link session number from the results

match '<FORM NAME="hitlist" METHOD="POST"
      ACTION="{LinkSession}">'
else
{
    LinkSession = "none"
}
websessionsave LinkSession

//extract the number of records found

match '<B>{RecordsFound}</B>
records were found.'
i = 0

//output results page HTML

<%
    <html><head><title>Search Results</title></head><body>
    <table border="1" width="100%" height="38">
  <tr>
    <td width="100%" colspan="4" bgcolor="#000080" height="5">
          <table width="100%"><tr><td width="33%">&nbsp;</td>
                <td align="center" width="34%"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY
RESULTS</b></font><font color="#FFFFFF"
                </b></font><p></td>
            <td width="33%" valign="bottom" align="right"><font
color="#FFFFFF" size="2" face="ARIAL, HELVETICA">{RecordsFound}
Records Found.</td></tr></table>
    </td>
  </tr>
    <tr>
    <td width="8%" height="19" align="center">Number</td>
    <td width="23%" height="19" align="center">Catalog Number</td>
    <td width="58%" height="19" align="center">Description</td>
    <td width="11%" height="19" align="center">Pub Year</td>
  </tr>
%>
while
{
    push Number
    replace "#", ""
```

Listing 28.3 Results page script (continued).

```
    pop Number

    if firsthit = "" { firsthit = "1" }
    if lasthit = "" { lasthit = "20" }

    //Use LinkSession in sViewURL
    sViewURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&VIEW%5E{Number}.x=31&VIEW%
5E{Number}.y=8"
    if i > 0
    {
<%

  <tr>
    <td width="8%" height="19" align="center"><a href="{sViewURL}">
#{Number}</a></td>
    <td width="23%" height="19">{CatalogNumber}</td>
    <td width="58%" height="19">{Description}</td>
    <td width="11%" height="19"
align="center">{PubYear}&nbsp;</td>
  </tr>
%>
    }
    else
    {
<%
    <tr>
        <td colspan=4 align="center"><a href="results.nql?firsthit=
{firsthit}&lasthit={lasthit}&type={type}&terms={terms}">Next 20
Matches</a></td>
    </tr>
%>
    }

    match '
<TR>
    <TD VALIGN = "TOP" ALIGN = "left" NOWRAP>
      <!-- print list number, checkbox, VIEW icon in first column -->
      <TABLE>
        <TR>
          <TD><B>{Number}</B>
              <INPUT TYPE=checkbox*</TD>
        </TR>

        <TR>
          <TD><INPUT TYPE=image*</TD>
```

continues

Listing 28.3 Results page script (continued).

```
        </TR>


      </TABLE>
    </TD>

    <TD VALIGN = "TOP" ALIGN = "left">*<B>{CatalogNumber}</B><BR>
{Description}</TD>*<TD NOWRAP VALIGN = "TOP" ALIGN =
"left">{Copies}<BR>{WhereAt}<BR>{PubYear}</TD>
</TR>'
    i = i + 1
}

closesession

if firsthit = "" { firsthit = 1 }
if lasthit = "" { lasthit = 20 }
<%
    </table>
<center>    <a href="results.nql?firsthit={firsthit}&lasthit={lasthit}
&type={type}&terms={terms}">Next 20 Matches</a>
</body>
</html>
%>
```

Listing 28.3 Results page script (continued).

Step 5: Run the Library Search Application

Now, we are ready to run the Library Search Web application. You can do this by accessing login.nql on your Web server. For example, if your Web server is named www.my-server.com and you have placed the three scripts in a /scripts/ directory, the URL is

```
http://www.my-server.com/scripts/login.nql
```

In response, you should see the login page appear that was shown earlier in Figure 28.3. If this does not occur, check the following:

- Check for typographical errors in your scripts.
- Check that the Web server address is correct.
- Check that NQL is installed on your Web server.

If the login page displays, go ahead and try a login. A user ID of *nql* and a password of *nql* should take you to the search page. Anything else should be rejected.

At the search page, enter a search word or phrase and select *Keyword*, *Author*, or *Title* from the option list. Click *Search* to submit a search. The search page was shown earlier in Figure 28.4.

The results from the search are displayed on the results page, which was shown earlier in Figure 28.5.

At this point you should have seen the execution of Library Search, an NQL-driven Web application.

Step 6: Understand the Login Page Script

We now want to make sure we understand every part of the Library Search scripts, beginning with the login page script, login.nql. The first line is a comment line.

```
//login - Login page of the Library Search application.
```

A Web session variable named *Error* is retrieved with *websessionload*. If this login page is being redisplayed because of a login error, *Error* will contain true. If *Error* is false or empty, this is the first time the login page is being accessed.

```
websessionload Error
```

A cookie named "loggedin" is retrieved with *websessiongetcookie*. If the value is not blank, the user is already logged in and the browser is redirected to the search page (menu.nql) with *websessionredirect*.

```
websessiongetcookie "loggedin", sLoggedIn
if sLoggedIn != ""
{
    websessionredirect "menu.nql"
}
```

The code that follows outputs the HTML of the login page.

```
//output start of page HTML

<%
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR"
content="Microsoft FrontPage 4.0">
<title>Library Demo - Login</title>
</head>

<body bgcolor="#FFFFFF" vlink="#0000FF" alink="#0000FF">
```

```
<p align="center"><br>
<br>
<br>
</p>
<div align="center"><center>

<table border="1" width="300">
    <tr>
        <td align="center"><table border="0" width="100%">
            <tr>
                <td bgcolor="#000080"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY</b>
                </font></td>
            </tr>
            <tr>
                <td><table border="0" cellpadding="5"
                width="100%">
                    <tr>
                        <td valign="top"><font size="1"
                        face="ARIAL, HELVETICA">You are now
                        entering a secure <br>
                        section of our library site. Please
                        enter your password below. </font></td>

                    </tr>
                </table>
             %>
```

If the page needs to report an incorrect login error, *Error* will be true. When this is the case, an appropriate error message is displayed on the page, in red.

```
if Error = true
{
    output '<font face="Arial" color="#FF0000" size="2"><i>UserID or
Password incorrect, please try again</i></font><br><br>'
    Error = false
    websessionsave Error
}
```

The output of HTML continues, and the script is complete.

```
            <%

                <table border="0">
                    <tr>
                        <td valign="top">

                <font size="2"
                        face="ARIAL, HELVETICA">User ID:<br>
                        <br>
                        Password:</font></td>
                        <td valign="top"><form name=loginForm
action="menu.nql"
                        method="POST">
                <p><input type="text" size="20"
                            name="UserID"><br>
                            <input type="password" size="20"
                            name="Password"></p>
                            <p><input type="submit" name="B1"
                            value="Login"
tabindex="1">&nbsp;&nbsp; <input type="reset"
                            name="B2" value="Clear"
tabindex="2">
                            <br>
                            </p>
                        </form>
                        </td>
                    </tr>
                </table>
                </td>
            </tr>
        </table>
        </td>
    </tr>
</table>

</center></div>

<script language="javascript">
    document.loginForm.UserID.focus();
</script>

<p>&nbsp;</p>

<p>&nbsp;</p>
</body>
</html>
%>
```

At this point, you've seen how to create a login page whose purpose is to accept a user ID and password and submit them to the search page.

Step 7: Understand the Search Page Script

We now want to understand the script for the search page, menu.nql. The first line is a comment line.

```
//menu - Search page of the Library Search application.
```

Next, two parameters are retrieved from the URL: *UserID* and *password*. The login page submits these form fields in the Web address that causes menu.nql to be activated. They are retrieved into variables with the *input* statement.

```
input UserID, Password
```

A check for the logged in cookie is made with *websessiongetcookie*. Unless there is a logged-in cookie already set, the user ID and password are now validated. If the script dislikes the user ID and password, the variable *Error* is set to true and stored in a Web session variable with *websessionsave*, and the browser is redirected to the login page, login.nql. The login script will redisplay the login page, and seeing the value of *Error* it will display an error message about an invalid login.

```
Error = False
websessiongetcookie "loggedin", sLoggedIn
if sLoggedIn = ""
{
    if UserID == "nql"
    {
        if Password != "nql"
        {
            Error = true
            websessionsave Error
            websessionredirect "login.nql"
        }
    }
    else
    {
        Error = true
        websessionsave Error
        websessionredirect "login.nql"
    }
```

After a successful login, a cookie is set that will mark the user as logged in for the remainder of the day.

```
    date sDate
    call FormatDate
    sCookieDate = sDayofWeek & ", " & sDay & "-" & sMonth & "-" & sYear

    sCookie = "yes; expires=" & sCookieDate & " 23:59:59 GMT; path=/;
domain=nqli.com"

    websessionsetcookie "loggedin", sCookie
    websessiongetcookie "loggedin", sLoggedIn
}
```

The HTML of the search page is now displayed, which includes a form for entering search text and selection of keyword, author, or title.

```
<%
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="GENERATOR"
content="Microsoft FrontPage 4.0">
<title>Demo Library - Search Menu</title>
</head>

<body bgcolor="#FFFFFF" vlink="#0000FF" alink="#0000FF">
<p align="center"><br>
<br>
<br>
</p>
<div align="center"><center>

<table border="1" width="300">
    <tr>
        <td align="center"><table border="0" width="100%">
            <tr>
                <td bgcolor="#000080"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY</b>
                </font></td>
            </tr>
            <tr>
                <td><table border="0" cellpadding="5"
                width="100%">
                    <tr>
                        <td valign="top"><font size="1" face="ARIAL,
HELVETICA">Thank
                          you for entering the Demo Library Lookup
system.
                          Please enter your search information
below.</font></td>

                    </tr>
                </table>
                <table border="0">
                    <tr>
                        <td valign="top"><font size="2"
                        face="ARIAL, HELVETICA">Text:<br>
                        <br>
                          Search by:</font></td>
                        <td valign="top"><form name=loginForm
action="results.nql"
                        method="POST">
                            <p><input type="text" size="20"
                            name="terms"><br>
                            <select size="1" name="type">
                              <option
value="keyword">Keyword</option>
                              <option
value="author">Author</option>
                              <option
value="title">Title</option>
                            </select></p>
                    <!--<p><input type="radio" name="searchtype"
value="standard"> Standard <input type="radio" name="searchtype"
value="complex"> Complex-->
                            <p><input type="submit" name="B1"
value="Search">&nbsp;&nbsp;<input type="reset" name="B2" value="Clear">
                            <br>
                            </p>
                        </form>
                        </td>
                    </tr>
                </table>
                </td>
            </tr>
        </table>
        </td>
    </tr>
</table>
</center></div>

<p>&nbsp;</p>

<p>&nbsp;</p>
</body>
</html>
%>
end
```

At the end of the script is a subroutine for formatting the date the way a cookie needs it expressed.

```
FormatDate:
    push sDate
    parsedate intMonth, sDay, sYear, intDayofWeek
    if sDay < 10
    { sDay = "0" & sDay }
```

```
if intMonth = "1"
{ sMonth = "Jan" }
if intMonth = "2"
{ sMonth = "Feb" }
if intMonth = "3"
{ sMonth = "Mar" }
if intMonth = "4"
{ sMonth = "Apr" }
if intMonth = "5"
{ sMonth = "May" }
if intMonth = "6"
{ sMonth = "Jun" }
if intMonth = "7"
{ sMonth = "Jul" }
if intMonth = "8"
{ sMonth = "Aug" }
if intMonth = "9"
{ sMonth = "Sep" }
if intMonth = "10"
{ sMonth = "Oct" }
if intMonth = "11"
{ sMonth = "Nov" }
if intMonth = "12"
{ sMonth = "Dec" }

if intDayofWeek = "1"
{ sDayofWeek = "Sun" }
if intDayofWeek = "2"
{ sDayofWeek = "Mon" }
if intDayofWeek = "3"
{ sDayofWeek = "Tue" }
if intDayofWeek = "4"
{ sDayofWeek = "Wed" }
if intDayofWeek = "5"
{ sDayofWeek = "Thu" }
if intDayofWeek = "6"
{ sDayofWeek = "Fri" }
if intDayofWeek = "7"
{ sDayofWeek = "Sat" }
return
```

At this point, you've seen how to create a search page that validates a user login and displays a form for library searching.

Step 8: Understand the Results Page Script

Finally, we want to understand the script for the results page, results.nql. The first line is a comment line.

```
//results - Results page of the Library Search application.
```

As in the previous script, form fields are retrieved from the URL and mapped into variables with the versatile *input* statement.

```
input terms, type, searchtype, firsthit, lasthit
```

A Web session is started with *openwebsession*, because the site we are accessing behind the scenes requires session continuity as you browse through results pages.

```
openwebsession
```

The search type (keyword, author, or title) is checked so that the appropriate URL for a search request is issued to the library site. First, the university library's main page is accessed to obtain some information. The library site returns several values we must retain for future access, including a session number and a link number. These are stored in session variables for later recall. When the user looks at the results page and clicks a link to get more results, this same script will re-execute.

```
//handle search by keyword

if type = "keyword"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5EGENERAL%
5ESUBJECT%5EGENERAL%5E%5Ewords%2Bor%2Bphrase.x=80&S_ICON%5EGENERAL%
5ESUBJECT%5EGENERAL%5E%5Ewords%2Bor%2Bphrase.y=4"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y=10"

    }
}

//handle search by author

if type = "author"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5EAU%5EAUTHOR%
5EAUTHORS%5EAuthor+Processing%5Eauthor.x=44&S_ICON%5EAU%5EAUTHOR%
5EAUTHORS%5EAuthor+Processing%5Eauthor.y=21"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y=10"

    }
}

//handle search by title

if type = "title"
{
    if firsthit = ""
    {
        get "http://www.library.emory.edu"
        match '<FORM NAME=searchform METHOD=POST
                ACTION={SiteSession}>'
        websessionsave SiteSession

        sURL = "http://www.library.emory.edu{SiteSession}?search_type=
search&searchdata1={terms}&library=ALL&sort_by=ANY&S_ICON%5ETI%5ETITLE%
5ESERIES%5ETitle+Processing%5Etitle.x=53&S_ICON%5ETI%5ETITLE%5ESERIES%
5ETitle+Processing%5Etitle.y=22"
    }
    else
    {
        websessionload SiteSession
        websessionload LinkSession

        sURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&SCROLL%5EF.x=61&SCROLL%5EF.y=10"

    }
}
```

With the search URL computed, it can now be issued to the library Web site. An *eval* statement resolves the variable references in the URL, and the *get* statement submits the search request and receives the first results page.

```
//submit search to web site and retrieve results

eval sURL
get sURL
```

A link session number is extracted from the results that will be needed when visiting continuation pages of the results. It is saved in a session variable named *LinkSession*.

```
//extract a link session number from the results

match '<FORM NAME="hitlist" METHOD="POST"
      ACTION="{LinkSession}">'
else
{
    LinkSession = "none"
}
websessionsave LinkSession
```

The number of records found is taken from the page using HTML pattern matching (the *match* statement) and stored in *RecordsFound*. The success condition is set to true by *match* if this is found. A loop control variable, *i*, is initialized to 0. The *i* variable will be used to count the results as they are extracted from the Web page.

```
//extract the number of records found

match '<B>{RecordsFound}</B>
records were found.'
i = 0
```

Some HTML code is output to begin a table on the results page. Note the use of *RecordsFound* as an embedded variable.

```
//output results page HTML

<%
    <html><head><title>Search Results</title></head><body>
    <table border="1" width="100%" height="38">
  <tr>
    <td width="100%" colspan="4" bgcolor="#000080" height="5">
          <table width="100%"><tr><td width="33%">&nbsp;</td>
                <td align="center" width="34%"><font color="#FFFFFF"
                size="3" face="ARIAL, HELVETICA"><b>DEMO LIBRARY
RESULTS</b></font><font color="#FFFFFF"
```

```
                </b></font><p></td>
            <td width="33%" valign="bottom" align="right"><font
color="#FFFFFF" size="2" face="ARIAL, HELVETICA">{RecordsFound} Records
Found.</td></tr></table>
    </td>
  </tr>
    <tr>
    <td width="8%" height="19" align="center">Number</td>
    <td width="23%" height="19" align="center">Catalog Number</td>
    <td width="58%" height="19" align="center">Description</td>
    <td width="11%" height="19" align="center">Pub Year</td>
  </tr>
%>
```

A *while* loop now iterates through all of the results. The success condition is still true from the last *match* statement (none of the intervening statements affect the success/failure condition). The body of the *while* loop displays a result and uses a *match* statement to find the next result, but the first time through we have not yet extracted any results. The test for *i > 0* causes results to be displayed only from the second iteration on. For each result, the appropriate HTML with embedded variables is output. The *match* statement at the bottom of the loop loads the next result data and sets the success/failure code. The loop iterates until there are no more results to display.

```
while
{
    push Number
    replace "#", ""
    pop Number

    if firsthit = "" { firsthit = "1" }
    if lasthit = "" { lasthit = "20" }

    //Use LinkSession in sViewURL
    sViewURL = "http://www.library.emory.edu{LinkSession}?first_hit=
{firsthit}&last_hit={lasthit}&form_type=&VIEW%5E{Number}.x=31&VIEW%
5E{Number}.y=8"
    if i > 0
    {
<%

  <tr>
    <td width="8%" height="19" align="center"><a
href="{sViewURL}">#{Number}</a></td>
    <td width="23%" height="19">{CatalogNumber}</td>
    <td width="58%" height="19">{Description}</td>
    <td width="11%" height="19"
align="center">{PubYear}&nbsp;</td>
  </tr>
%>
```

```
    }
    else
    {
<%
    <tr>
        <td colspan=4 align="center"><a
href="results.nql?firsthit= {firsthit}&lasthit={lasthit}&type={type}
&terms={terms}">Next 20 Matches</a></td>
    </tr>
%>
    }

    match '
<TR>
    <TD VALIGN = "TOP" ALIGN = "left" NOWRAP>
      <!-- print list number, checkbox, VIEW icon in first column -->
      <TABLE>
        <TR>
          <TD><B>{Number}</B>
              <INPUT TYPE=checkbox*</TD>
        </TR>

        <TR>
          <TD><INPUT TYPE=image*</TD>
        </TR>

      </TABLE>
    </TD>

    <TD VALIGN = "TOP" ALIGN = "left">*<B>{CatalogNumber}</B><BR>
{Description}</TD>*<TD NOWRAP VALIGN = "TOP" ALIGN = "left">{Copies}<BR>
{WhereAt}<BR>{PubYear}</TD>
</TR>'
    i = i + 1
}
```

The Web session is now closed with *closwebesession*.

```
closesession
```

Starting and ending numbers are computed because they are needed in a URL for continuation pages. The final HTML is written out, and the script is complete.

```
if firsthit = "" { firsthit = 1 }
if lasthit = "" { lasthit = 20 }
<%
    </table>
<center>    <a
href="results.nql?firsthit={firsthit}&lasthit={lasthit}
&type={type}&terms={terms}">Next 20 Matches</a>
</body>
```

```
</html>
%>
```

At this point, you’ve seen how to issue a search request against a Web site and retrieve results, redisplaying the data in an NQL-driven Web application.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- Expand the accepted user IDs and passwords. To be really sophisticated, use a database or encrypted data file to hold this information.
- Instead of merely displaying the results, export them to a document or spreadsheet using the NQL export statements.

Chapter Summary

NQL scripts can run as Web applications, just as Perl, ASP, and JSP scripts can.

- URL parameters are mapped to the script’s input stream. The input statement retrieves URL parameters into variables.
- The script’s output stream becomes the HTML response sent to a browser. The output statement may be used to generate the HTML output, or the more convenient <% and %> characters may enclose HTML.

Web sessions allow multiple NQL scripts to share variables, allowing state to be maintained in a Web application.

- Web sessions are opened with *openwebsession*.
- Session variables are set with *websessionsave*.
- Session variables are retrieved with *websessionload*.
- Web sessions are closed with *closewebsession*.

Web sessions can examine or set cookies.

- Cookies are read with *websessiongetcookie*.
- Cookies are set with *websessionsetcookie*.

To create Web applications that support mobile devices rather than desktop Web browsers, refer to Chapter 29, “Supporting Mobile Devices.”

CHAPTER 29

Supporting Mobile Devices

A variety of mobile devices can be supported by NQL on a Web server. This chapter explains how to write mobile applications in NQL. The tutorial at the end of the chapter is a Web script that recognizes the device attempting to access it.

Mobile Applications

Mobile applications are simply Web applications with some interesting twists. A standard Web application runs on a Web server and produces HTML output intended for Web browsers. A mobile application also runs on a Web server, but produces output intended for a specific type of mobile device or perhaps several types of devices. It is also possible to write applications that run directly on mobile devices, but this is more difficult and requires a more sophisticated level of programming experience. In this chapter, we are only concerned with mobile applications that run on Web servers.

There are many kinds of mobile devices, and new ones are continually being introduced. At the time of this writing, NQL has features that directly support the following devices: Palm devices, Windows CE devices, WAP phones, and RIM Blackberry devices.

Palm Devices

Palm devices run the PalmOS operating system, and are manufactured by Palm Inc. as well as licensed manufacturers such as Handspring. Some PalmOS devices are connected, such as the Palm VII and (with the appropriate modem hardware) the Palm V. Connected Palm devices can access Web applications using a system known as Web clipping. You can test Palm Web applications with actual Palm devices or with a software emulator that you can run on your PC desktop, available from Palm. Figure 29.1 shows the appearance of a Palm device.

Web applications customized for Palm devices must generate a subset of HTML known as *Palm-friendly HTML* and target a small 190 × 190-pixel monochrome display area.

Windows CE Devices/Pocket PCs

Pocket PCs and many other devices run the Microsoft Windows CE operating system. Many of these devices are connected and can access Web applications using browser software such as Pocket Explorer. You can test Windows CE Web applications with actual Windows CE devices or with a software emulator that you can run on your PC desktop, available from Microsoft.

Web applications customized for Windows CE devices generate HTML and target a small display area. The availability of color and display size vary among manufacturers.

WAP Devices/Web-enabled Mobile Phones

Web-enabled mobile phones and other devices support Wireless Application Protocol (WAP), a suite of protocols for mobile device communication featuring a content language named Wireless Markup Language (WML). Older phones support a slightly different language called Handheld Device Markup Language (HDML). You can test WAP Web applications with actual WAP devices or with a software emulator that you can run on your PC desktop. Figure 29.2 shows the typical appearance of a WAP device.

Web applications customized for WAP devices generate WML or HDML rather than HTML and target a small display area. The availability of color and display size vary among manufacturers.

RIM Blackberry Devices

Devices manufactured by Research In Motion, Inc., run the RIM OS operating system. The more sophisticated devices, such as the Blackberry 950, include Web-browsing capabilities using browser software known as Go.Web. You can test RIM Web applications with actual RIM devices or with a software emulator that you can run on your PC desktop, available from Research in Motion. Figure 29.3 shows the typical appearance of a RIM device.

Web applications customized for RIM Blackberry devices must generate a subset of HTML supported by Go.Web and target a small 20-line monochrome display area.

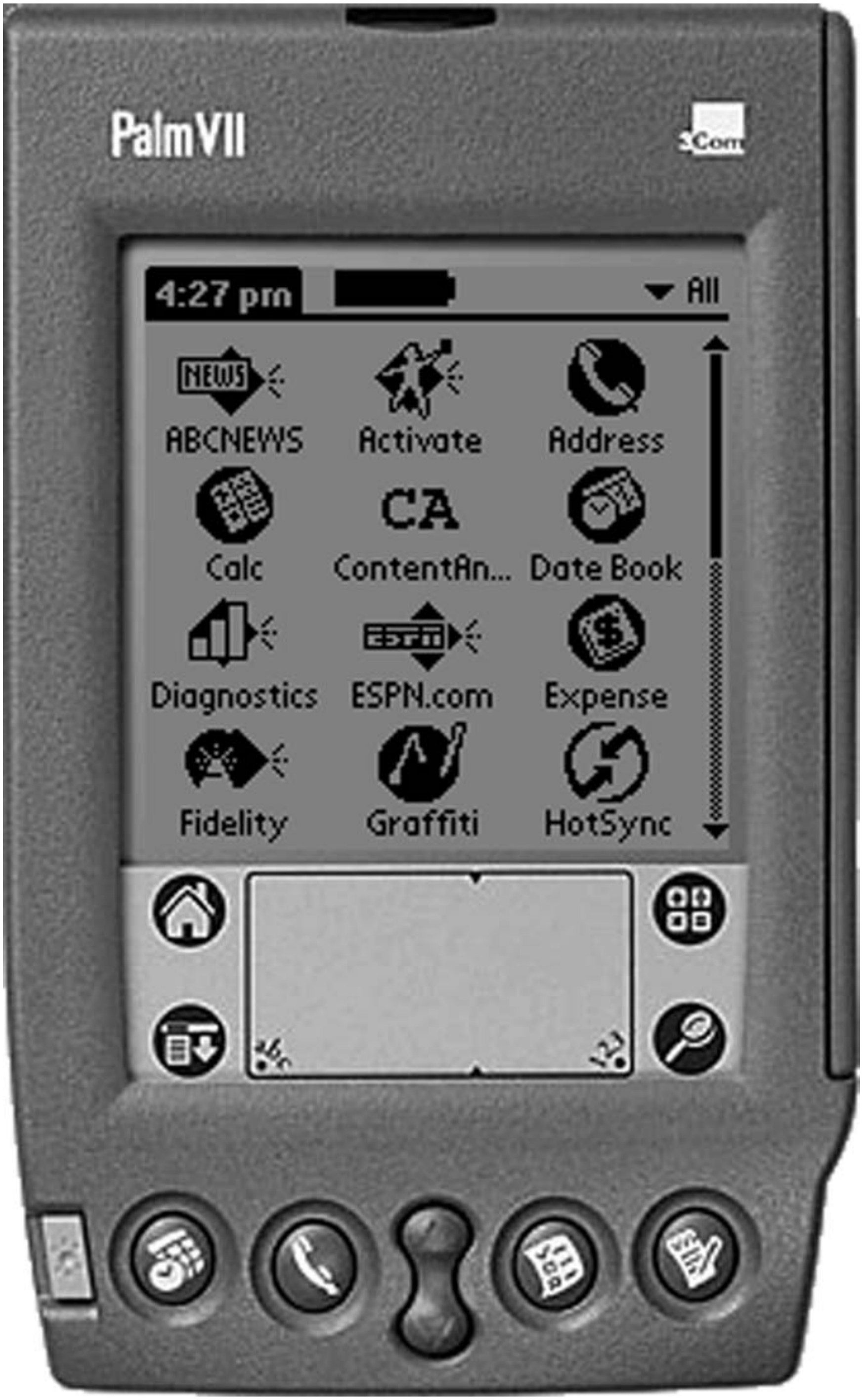


Figure 29.1 Palm device.

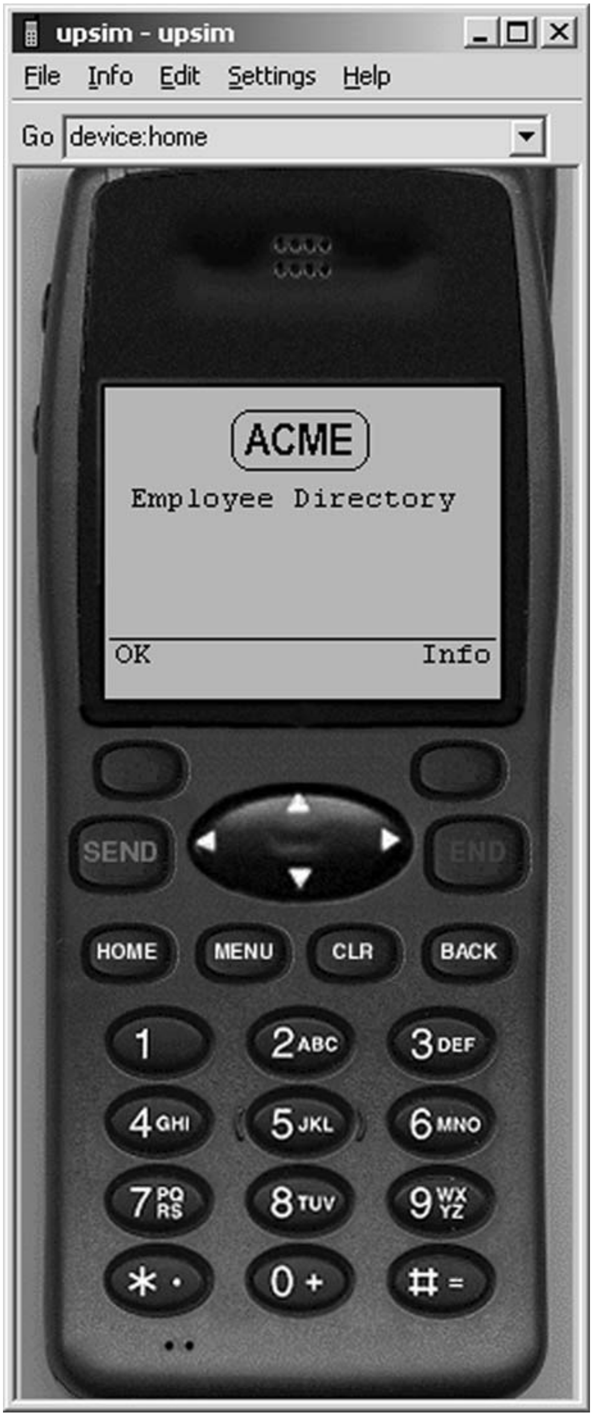


Figure 29.2 WAP device.

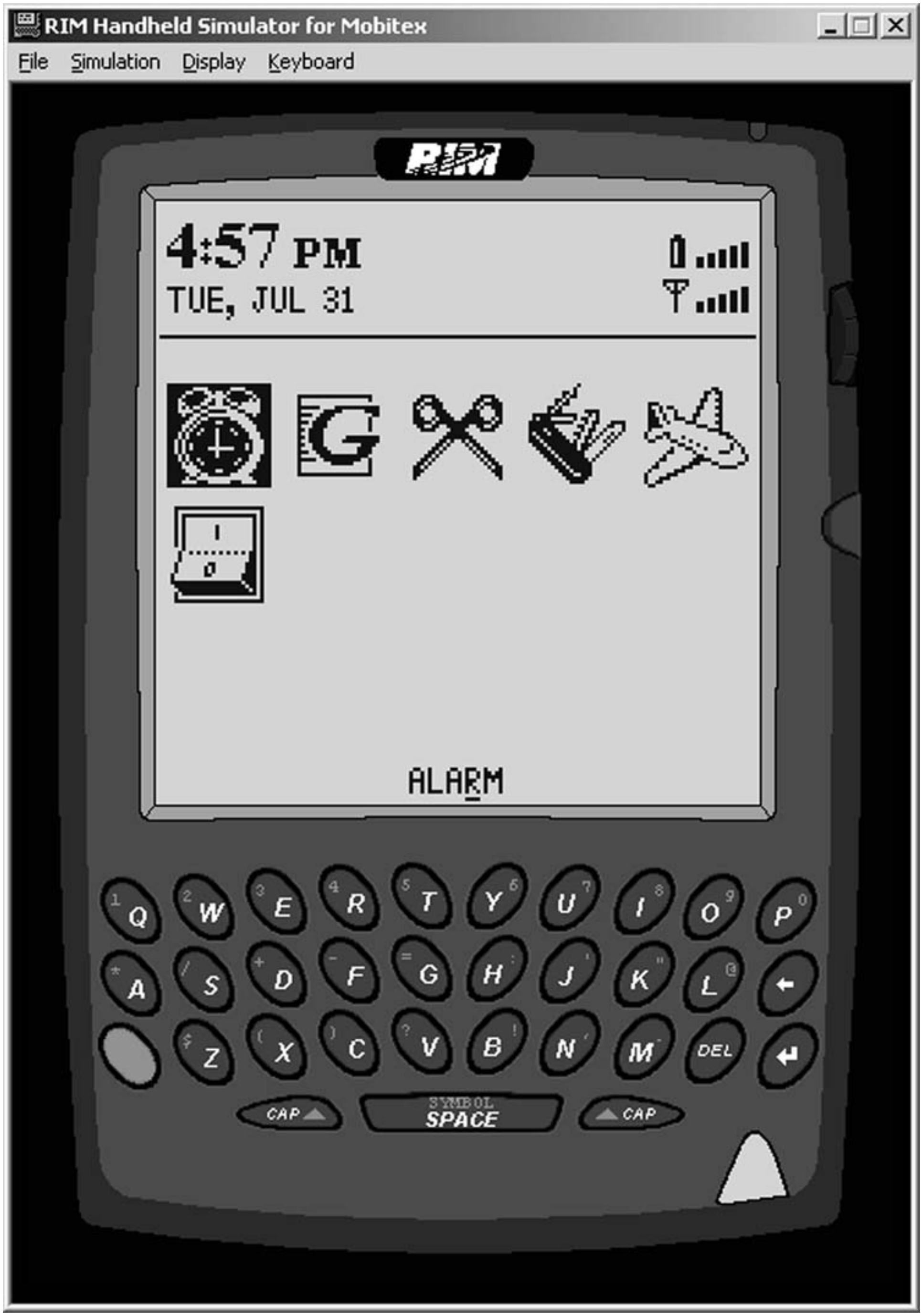


Figure 29.3 RIM device.

Mobile Programming in NQL

A mobile application NQL script follows all of the rules presented for Web applications in Chapter 28, but must output the appropriate format and size of content for the mobile device(s) it is intended to support. Although you can write NQL scripts that support just one kind of device, it is actually possible to use conditional logic so that a single script can generate appropriate output for a range of mobile devices as well as a standard Web browser.

The NQL mobile device statements are summarized in Table 29.1. The function versions of the mobile device statements are listed in Table 29.2. The mobile device statements are described individually in the following sections.

Detecting Context

The *getcontext* statement determines what type of device is accessing the Web server. There are no parameters.

```
getcontext
```

The Web headers from the client accessing the Web server are examined to determine the device type. Once *getcontext* executes, the statements *browser*, *hdml*, *hpc*, *html*, *large*, *palm*, *rim*, *small*, *wap*, *WinCE*, and *wml* may be used in the script to conditionally execute code. The following code uses *getcontext* to determine if the script is being accessed by a browser or by some other device.

```
getcontext
if browser()
{
    <HTML><BODY>This application is meant for mobile devices,
    not web browsers.</HTML>
}
```

If *getcontext* does not recognize a specific type of device, it sets the context to that of a Web browser.

When testing mobile device scripts, it is sometimes useful to force the device type. The *setcontext* statement is used for this purpose.

Setting Context

To force the device type, the *setcontext* statement is used. You might want to force the device type when debugging with a standard Web browser so that NQL believes it is talking to a specific kind of device. The parameter is a device type string.

```
setcontext type
```

The supported device types are listed in Table 29.3.

Table 29.1 Mobile Device Statements

STATEMENT	PARAMETERS	DESCRIPTION
browser		Conditionally executes a code block if the current device is a Web browser
closecontent		Closes a content session
getcontext		Determines the type of device accessing the script
hdml		Conditionally executes a code block if the current device supports HDML
hpc		Conditionally executes a code block if the current device is a Handheld PC
html		Conditionally executes a code block if the current device supports HTML
large		Conditionally executes a code block if the current device has a large display area
makepqa	*file.htm*, *pqa-filespec* [,*qab.exe* [, *palmuser* [, *large-iconfilespec* [, *small-iconfilespec*]]]]	Generates a Palm Portable Query Application (PQA)
opencontent	[*title* [, *code-output-flag*]]	Opens a content session
palm		Conditionally executes a code block if the current device is a Palm device
palmid	[*var*]	Retrieves the device ID of a Palm device
rim		Conditionally executes a code block if the current device is a RIM device
rimid	[*var*]	Retrieves the device ID of a RIM device
setcontext	*type*	Declares a device type
small		Conditionally executes a code block if the current device has a small display area
wap		Conditionally executes a code block if the current device is a WAP device
wapid	[*var*]	Retrieves the device ID of a WAP device
WinCE		Conditionally executes a code block if the current device is a Windows CE device
wml		Conditionally executes a code block if the current device supports WML

Table 29.2 Mobile Device Functions

FUNCTION	DESCRIPTION
boolean browser( )	Conditionally executes a code block if the current device is a Web browser
boolean hdml( )	Conditionally executes a code block if the current device supports HDML
boolean hpc( )	Conditionally executes a code block if the current device is a Handheld PC
boolean html( )	Conditionally executes a code block if the current device supports HTML
boolean large( )	Conditionally executes a code block if the current device has a large display area
boolean makepqa(*file.htm*, *pqa-filespec* [, *qab.exe* [, *palmuser* [, *large-iconfilespec* [, *small-iconfilespec*]]]])	Generates a Palm Portable Query Application (PQA)
boolean palm( )	Conditionally executes a code block if the current device is a Palm device
string palmid([*var*])	Retrieves the device ID of a Palm device
boolean rim( )	Conditionally executes a code block if the current device is a RIM device
string rimid([*var*])	Retrieves the device ID of a RIM device
boolean small( )	Conditionally executes a code block if the current device has a small display area
boolean wap( )	Conditionally executes a code block if the current device is a WAP device
boolean wapid([*var*])	Retrieves the device ID of a WAP device
boolean WinCE( )	Conditionally executes a code block if the current device is a Windows CE device
boolean wml( )	Conditionally executes a code block if the current device supports WML

The following code forces a Palm context in order to test the Palm-specific code in the script.

```
setcontext "palm"
    ...
```

Table 29.3 Setcontext Values

VALUE	DESCRIPTION
browser	A standard desktop PC with a Web browser that accepts HTML
palm	A connected Palm PDA that accepts Palm-friendly HTML, such as a Palm VII
hdml	A wireless device that accepts HDML, such as older smart phones
hpc	A Windows CE Handheld PC (a PDA with a larger display area and a keyboard)
pocketpc	A Windows CE Pocket PC (a PDA with a smaller display area and no keyboard)
rim	A RIM OS device that accepts Go.Web-friendly HTML
wml	A wireless device that accepts WML, such as newer smart phones

```
palm
{
    <HTML><BODY>
        ...
    </HTML></BODY>
}
```

Generally, *setcontext* is used for testing. For live use where you want to detect the device accessing the Web server, use the *getcontext* statement.

Content Sessions

You can generate the opening prefix code and closing suffix code of the response page for mobile devices with the *opencontent* and *closecontent* statements. A *getcontext* statement must appear earlier in the script to determine the device type.

The prefix code is generated with the *opencontent* statement, which takes no parameters. This may generate HTML, PDA-friendly HTML, HDML, or WML, depending on the device type.

```
opencontext
```

The suffix code is generated with the *closecontent* statement, which takes no parameters. Like *opencontent*, this generates the appropriate tags in the appropriate content language for the device type.

```
closecontent
```

The following code shows the use of *getcontext*, *opencontent*, and *closecontent*.

```
getcontext
opencontent
<%Your order has been processed. Thank you.
%>
closecontent
```

You are not required to use *opencontent* and *closecontent*; they are merely convenient for generating the typical opening and closing tags for a response page.

Conditional Blocks

NQL supplies statements and functions that test for a specific device type or device characteristic and conditionally execute a code block. This allows you to have a script that is capable of supporting multiple devices as well as Web browsers. A *getcontext* statement must have been executed in order for these statements and functions to work.

Checking for a Browser

The *browser* statement executes a code block if the Web server is being accessed by a desktop Web browser such as Microsoft Internet Explorer or Netscape Navigator.

```
browser
{
    ...statements...
}
```

There is also a function version of *browser* that returns true if the access device is a desktop PC with a Web browser.

```
boolean browser()
```

For browser output, you can generally assume a large display area, HTML as a content language, and the ability to accept rich documents that contain color, fonts, images, and multimedia.

Checking for an HDML Device

The *hdml* statement executes a code block if the Web server is being accessed by a WAP device that supports HDML, such as older Web-enabled mobile phones.

```
hdml
{
    ...statements...
}
```

The function version of *hdml* returns true if the access device speaks HDML.

```
boolean hdml()
```

For HDML output, you can generally assume a small display area, HDML as a content language, and a requirement for small, simple documents.

Checking for a Handheld PC

The *hpc* statement executes a code block if the Web server is being accessed by a Windows CE device that qualifies as a Handheld PC. In contrast to Pocket PCs, Handheld PCs have keyboards and larger displays.

```
hpc
{
    ...statements...
}
```

The function version of *hpc* returns true if the access device is a Handheld PC.

```
boolean hpc()
```

For Handheld PC output, you can generally assume a small display area that may not support color, limited HTML as a content language, and a requirement for small, simple documents.

Checking for an HTML Device

The *html* statement executes a code block if the Web server is being accessed by a Web browser or mobile device that supports HyperText Markup Language (HTML), such as Pocket PCs and Palm devices.

```
html
{
    ...statements...
}
```

The function version of *html* returns true if the access device speaks HTML.

```
boolean html()
```

Checking for a Large-size Device

The *large* statement executes a code block if the Web server is being accessed by a device or application with a large display size, such as a desktop Web browser.

```
large
{
    ...statements...
}
```

The function version of *large* returns true if the access device has a large display area.

```
boolean large()
```

For large displays, you can generally assume the ability to send large, rich content and a greater likelihood of supporting color and fonts.

Checking for a Palm Device

The *palm* statement executes a code block if the Web server is being accessed by a Palm OS device, such as a Palm VII.

```
palm
{
    ...statements...
}
```

The function version of *palm* returns true if the access device is a Palm device.

```
boolean palm()
```

For Palm output, you can generally assume a small-size display that may not support color, Palm-friendly HTML as a content language, and a requirement for small, simple documents.

Checking for a RIM Blackberry Device

The *rim* statement executes a code block if the Web server is being accessed by a RIM device, such as a RIM Blackberry 950.

```
rim
{
    ...statements...
}
```

The function version of *rim* returns true if the access device is a RIM device.

```
boolean rim()
```

For RIM output, you can generally assume a small-size display that probably does not support color, Palm-friendly HTML as a content language, and a requirement for small, simple documents.

Checking for a Small-size Device

The *small* statement executes a code block if the Web server is being accessed by device or application with a small display area, such as a mobile device.

```
small
{
    ...statements...
}
```

The function version of *small* returns true if the access device has a large display area.

```
boolean small()
```

For small displays, you can generally assume the ability to send short, simple content and a lesser likelihood of supporting color or fonts.

Checking for a WAP Device

The *wap* statement executes a code block if the Web server is being accessed by a WAP device, such as a Web-enabled mobile phone.

```
wap
{
    ...statements...
}
```

The function version of *wap* returns true if the access device is a WAP device.

```
boolean wap()
```

Checking for a Windows CE Device

The *WinCE* statement executes a code block if the Web server is being accessed by a Windows CE device, such as a Pocket PC or Handheld PC.

```
WinCE
{
    ...statements...
}
```

The function version of *WinCE* returns true if the access device is a Windows CE device.

```
boolean WinCE()
```

Checking for a WML Device

The *wml* statement executes a code block if the Web server is being accessed by a WAP device that supports Wireless Markup Language (WML), such as newer Web-enabled mobile phones.

```
wml
{
    ...statements...
}
```

The function version of *wml* returns true if the access device speaks WML.

```
boolean wml()
```

For WML output, you can generally assume a small display area, WML as a content language, and a requirement for small, simple documents.

Obtaining a Palm Device ID

When an NQL script is accessed by a Palm device, the *palmid* statement or function retrieves the Palm's device ID. The *palmid* statement sets the device ID into the specified variable, or onto the stack if no variable is specified. The *palmid* function returns the device ID as its result.

```
palmid [var]
string palmid()
```

The Palm device ID is retrieved from the Web request headers sent by the device. If the device accessing the Web server is not a Palm device, the *palmid* statement sets a failure condition and does not set a value. The following code uses *palmid* to record the device ID.

```
getcontext
    ...
if palm()
{
    string sDeviceID = palmid()
}
```

The device ID can be used for mapping devices to accounts, for logging activity, and for recognizing repeat visitors to a Web site.

Obtaining a RIM Device ID

When an NQL script is accessed by a RIM Blackberry device, the *rimid* statement or function retrieves the Blackberry's device ID. The *rimid* statement sets the device ID into the specified variable, or onto the stack if no variable is specified. The *rimid* function returns the device ID as its result.

```
rimid [var]
string rimid()
```

The RIM device ID is retrieved from the Web request headers sent by the device. If the device accessing the Web server is not a RIM device, the *rimid* statement sets a failure condition and does not set a value. The following code uses *rimid* to record the device ID.

```
getcontext
    ...
if rim()
{
    string sDeviceID = rimid()
}
```

The device ID can be used for mapping devices to accounts, for logging activity, and for recognizing repeat visitors to a Web site.

Generating a Palm Query Application

NQL has the ability to generate a Palm Query Application (PQA). A PQA is a starting Web page form and graphics for a Palm Web application that gets installed on the Palm device itself. Normally, PQAs are built interactively using an application from Palm known as Query Builder, which has the file name qab.exe.

The *makepqa* statement or function invokes Query Builder and generates a PQA from an HTML document. The parameters are the file specification of an HTML page, the name of the PQA file to generate, the path (location) of Query Builder, the Palm user the PQA is intended for, a large icon file, and a small icon file. Only the HTML page and the PQA file specification are mandatory.

```
makepqa file.htm, pqa-filespec [, qab.exe [, palmuser [, large-
confilespec [, small-iconfilespec]]]]
boolean makepqa(file.htm, pqa-filespec [, qab.exe [, palmuser [, large-
confilespec [, small-iconfilespec]]]])
```

The Query Builder application is launched and commanded to generate a PQA file using the specified parameters. If a PQA file is generated successfully, a success condition results (the function returns true). The following code uses *makepqa* to generate a PQA file.

```
if !makepqa("stockvue.htm", "stockvue.pqa")
{
    show "The PQA could not be created."
}
```

Tutorial: DeviceType

Now that we have a foundation in mobile device programming, let's put it to work in an NQL script. In this tutorial, you will create a Web application that can be accessed by a variety of mobile devices as well as Web browsers. The application does nothing more than identify the type of device it believes it is being accessed by and display some of the device's characteristics.

In order to run this tutorial, you will need access to a Web server, and NQL needs to be installed on that Web server. For example, if you are using the Windows edition of NQL, a Windows NT/2000/XP server with IIS may be used. When installing NQL on the server, the installation program may ask if you want the IIS settings configured for NQL as a Web application. If so, respond in the affirmative.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the DeviceType script.
3. Run the DeviceType script.
4. Understand the DeviceType script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Create a mobile device Web application.
- Detect the device type.
- Conditionally execute code based on device type and device attributes.

Let's begin!

Step 1: Launch the NQL Development Environment

On a Windows system, launching the NQL development environment can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the DeviceType Script

In the NQL development environment, enter the script shown in Listing 29.1 and save it under the name DeviceType.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

```
//DeviceType - recognizes device type and displays the device and its
characteristics

getcontext

//output beginning page content

opencontent "test"

//display device type

browser
{
    <% You are using a browser!<P> %>
}
palm
{
    <% You are using a Palm!<P> %>
}
pocketpc
{
    <% You are using a Pocket PC!<P> %>
}
wap
{
    <% This device is classified as a WAP device. %>
}
rim
{
    <% You are using a Rim Blackberry!<P> %>
}

//display device language

html
{
    <% This device is classified as an HTML device.<P> %>
}
hdml
{
    <% <DISPLAY>You are using an HDML wireless device! %>
}
wml
{
    <% <card><p>You are using a WML wireless device! %>
}
```

continues

Listing 29.1 DeviceType script.

```
//display device characteristics

large
{
    <% This device has a large-size display.<P>%>
}
small
{
    <% This device has a small-size display. %>
}

//output closing page content

hdml
{
    <% </DISPLAY> %>
}
wml
{
    <% </p></card> %>
}

closecontent
```

Listing 29.1 DeviceType script (continued).

If you prefer, you may copy DeviceType.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select DeviceType.nql from the Tutorials\ch29 folder.

You need to copy this script onto a Web server in a publicly accessible directory, such as c:\InetPub\Scripts in the case of a Windows Web server.

At this point, the script in Listing 29.1 should be entered and saved, either because you entered it by hand, or because you opened the script from the companion CD. The script should have been placed on a Web server.

Step 3: Run the DeviceType Script

Now, we are ready to run the DeviceType Web application. You can do this by accessing DeviceType.nql on your Web server using a Web browser. For example, if your Web server is named www.my-server.com and you have placed the script in a /scripts/ directory, the URL is

```
http://www.my-server.com/scripts/DeviceType.nql
```

In response, you should see the following displayed:

```
You are using a browser!
This device is classified as an HTML device.
This device has a large-size display.
```

If this doesn't happen, check the following:

- Check for typographical errors in your scripts.
- Check that the Web server address is correct.
- Check that NQL is installed on your Web server.

If you have access to a Web-capable mobile device, try accessing the same URL. The information displayed should match the type and characteristics of the device. On a Palm device, you should see this:

```
You are using a Palm!
This device is classified as an HTML device.
This device has a small-size display.
```

At this point you should have seen the DeviceType script, an NQL scripted Web application that can be accessed by a wide variety of mobile devices.

Step 4: Understand the DeviceType Script

We now want to make sure that we understand every part of the DeviceType script. The first line is a comment line.

```
//DeviceType - recognizes device type and displays the device and its
characteristics
```

The *getcontext* statement figures out the type of device accessing the Web server, and remembers this information for the duration of the script.

```
getcontext
```

The beginning of the response page is generated with *opencontent*. The format of the output is determined by the device type. For a browser, Palm device, Pocket PC, or RIM device, HTML (or a suitable subset) is the content language. For a WAP device, HDML or WML is the content language.

```
//output beginning page content

opencontent "test"
```

The device type is now checked with conditional code blocks. The *browser* code block executes only if the device type is a desktop PC with a Web browser; the *palm* code block executes only if the device type is a Palm; the *pocketpc* code block executes if the device is a Pocket PC; and so on.

```
//display device type

browser
{
    <% You are using a browser!<P> %>
}
palm
{
    <% You are using a Palm!<P> %>
}
pocketpc
{
    <% You are using a Pocket PC!<P> %>
}
wap
{
    <% This device is classified as a WAP device. %>
}
rim
{
    <% You are using a Rim Blackberry!<P> %>
}
```

The content language of the device is now displayed. The *html* code block executes if the device speaks HTML or a subset of HTML; the *hdml* code block executes for an older WAP phone; the *wml* code block executes for a newer WAP phone.

```
//display device language

html
{
    <% This device is classified as an HTML device.<P> %>
}
hdml
{
    <% <DISPLAY>You are using an HDML wireless device! %>
}
wml
{
    <% <card><p>You are using a WML wireless device! %>
}
```

Next, the device display characteristics are checked. The *large* code block will execute for large-display devices such as a desktop PC with a Web browser; the *small* code block will execute for small-display devices such as mobile devices.

```
//display device characteristics

large
```

```
{
    <% This device has a large-size display.<P>%>
}
small
{
    <% This device has a small-size display. %>
}
```

The end of the page is at hand. If we've been outputting to an HDML or WML-receptive device, ending tags need to be written. The *closecontent* outputs final prefix tags for the device.

```
//output closing page content

hdml
{
    <% </DISPLAY> %>
}
wml
{
    <% </p></card> %>
}

closecontent
```

At this point, you've seen an NQL Web script capable of detecting and responding to an assortment of mobile devices.

Further Exercises

You could amend this example in a number of ways. Here are some interesting modifications that can be made:

- For mobile devices with ID numbers, such as Palm devices, detect and report the device's ID.
- Include a picture in the display, but only for devices capable of supporting graphics.
- Output text to the display, choosing to output a large amount of text or an abbreviated version of the text based on the device's characteristics.

Chapter Summary

Network Query Language can drive mobile device Web applications and supports a variety of Web-enabled devices.

The device type can be sensed or can be set specifically.

- The *getcontext* statement senses the type of device accessing the Web server.
- The *setcontext* statement forces a device type in the script and is useful for debugging.
- The supported device types include a desktop PC with a Web browser, a Palm device, a Windows CE Pocket PC, a Windows CE Handheld PC, a RIM Blackberry, an HDML WAP phone, and a WML WAP phone.

The opening and closing content of a page can be automatically generated.

- The *opencontent* statement outputs the content prefix for the page.
- The *closecontent* statement outputs the content suffix for the page.

Conditional statements can execute code blocks based on device type.

- The *browser* statement executes a code block if the device type is a desktop PC with a Web browser.
- The *hpc* statement executes a code block if the device type is a Handheld PC.
- The *palm* statement executes a code block if the device type is a Palm.
- The *rim* statement executes a code block if the device type is a RIM Blackberry.
- The *wap* statement executes a code block if the device type is a WAP device such as a Web-enabled mobile phone.
- The *WinCE* statement executes a code block if the device type is Windows CE device.

Conditional statements can execute code blocks based on device characteristics.

- The *browser* statement executes a code block if the device type is a desktop PC with a Web browser.
- The *html* statement executes a code block if the device speaks HTML.
- The *hdml* statement executes a code block if the device speaks HDML.
- The *wml* statement executes a code block if the device speaks WML.
- The *large* statement executes a code block if the device type has a large display area.
- The *small* statement executes a code block if the device type has a small display area.

Some devices report an identification number, which NQL can detect.

- The *palmid* statement obtains a Palm device's ID number.
- The *rimid* statement obtains a RIM Blackberry device's ID number.
- The *wapid* statement obtains a WAP device's ID number.

Network Query Language supports mobile device Web applications that can be tailored to the type of device making the access. Mobile applications can leverage all of NQL's other capabilities, including communications with the Internet and the corporate network.

CHAPTER

30

System Actions and Information

Network Query Language can instruct the operating system to perform actions (such as rebooting the computer) and retrieve system information (such as the amount of available memory). This chapter describes NQL's operations for invoking system actions and retrieving system information. The tutorial at the end of the chapter checks that the user is an administrator and, if authorization is given, restarts the computer.

System Actions

The NQL system action statements are summarized in Table 30.1. Many of these statements are supported only for Microsoft Windows.

There is also a function edition of *settime*. The system action statements and functions are described individually in the following sections.

Logging Off the Current User

The *logoff* statement logs off the current user. There are no parameters.

```
logoff
```

Table 30.1 System Action Statements

STATEMENT	PARAMETERS	DESCRIPTION
logoff		Logs off the current user
powerdown		Powers off the computer
reboot		Reboots the computer
restart		Reboots the computer
settime	*date/time*	Sets the system date/time
shutdown		Shuts down the computer

For continued use of the computer, the user will need to log in again. *Logoff* is only supported for Microsoft Windows.

Powering Down the Computer

The *powerdown* statement powers down the computer. There are no parameters.

```
powerdown
```

The computer shuts down and powers itself off. Note that not all computers have this capability, though most recent systems do. *Powerdown* is supported only for Microsoft Windows.

Rebooting the Computer

The *reboot* or *restart* statements reboot the computer (they are identical). There are no parameters.

```
reboot
restart
```

The computer reboots as if the reset button had been pressed. *Reboot* and *restart* are supported only for Microsoft Windows.

Shutting Down the Computer

The *shutdown* statement shuts down the computer. There are no parameters.

```
shutdown
```

The computer shuts down, but does not power itself off. *Shutdown* is supported only for Microsoft Windows.

Setting the System Date and Time

The *settime* statement sets the system date and/or time. The parameter is a date/time string.

```
settime date/time
boolean settime(date/time)
```

If the date/time is accepted by the operating system, a success condition results (the function returns true). The following code uses *settime* to set the date and time.

```
settime "5/2/2002 10:00"
```

To retrieve the system time, use the *gettime* statement.

System Information

The NQL system information statements are summarized in Table 30.2.

The function versions of the NQL system information statements are listed in Table 30.3.

The system information functions are described individually in the following sections.

Table 30.2 System Information Statements

STATEMENT	PARAMETERS	DESCRIPTION
appname	[*var*]	Returns the current application name
folder	*type*	Returns the path of a logical folder
memorystatus	*nPercentInUse* [, *nTotalVirtual*, *nAvailVirtual*, *nTotalPhysical*, *nAvailPhysical*, *nTotalPageFile*, *nAvailPageFile*]	Returns information about available virtual memory
os		Returns the operating system name
osversion		Returns the operating system version
usergroups	*username* [, *server*]	Returns the permission groups a user belongs to

Table 30.3 The System Information Functions

FUNCTION	DESCRIPTION
string appname()	Returns the current application name
string folder(*type*)	Returns the path of a logical folder
string os()	Returns the operating system name
string osversion()	Returns the operating system version
boolean usergroups(*username* [, *server*])	Returns the permission groups a user belongs to

Retrieving the Application Name

The *appname* statement or function returns the name of the application that is executing the NQL script. The statement may specify a variable to receive the application name, otherwise it is placed on the stack. The function returns the application name as its result.

```
appname [var]
string appname()
```

The string returned is the name of the application calling NQL, such as C:\Program Files\NQL\NQLgui.exe. The following code uses *appname* to determine if a particular program is calling NQL.

```
if contains(appname(), "backup")
{
    show "This scripts cannot be called from the backup program")
    end
}
```

The application name is useful for determining the context in which an NQL script is running.

Determining Folder Paths

The *folder* statement or function returns a path for a logical folder type. The parameter is a string indicating the type of folder desired.

```
folder type
string folder(type)
```

The folder type may be any of the values listed in Table 30.4.

Table 30.4 Logical Folder Types

TYPE	DESCRIPTION	EXAMPLE
cookies	Browser cookies location	c:\Windows\cookies
desktop	Desktop location	c:\Windows\Desktop
favorites	Returns the working directory path	c:\Windows\Favorites
os	Operating system folder	c:\Windows
personal	Personal folder for the current user	c:\My Documents
programs	System folder for program files	c:\Windows\System
startup	Start-up folder	c:\Windows\StartMenu\Programs\StartUp

The statement form of *folder* pushes the path on the stack; the function returns the path as a result. The value returned is the path for the specified folder. The following code uses *folder* to discover the user's My Documents directory and store a file there.

```
sPath = folder("personal")
create sPath & "note.txt"
write "Test"
close
```

Monitoring Virtual Memory

The *memorystatus* statement returns information about memory, virtual memory, and the swap file. The seven parameters are variables to receive the following values, in order.

1. The percentage of memory in use.
2. The total amount of virtual memory to use.
3. The total amount of virtual memory available.
4. The total amount of physical memory in use.
5. The total amount of physical memory available.
6. The total amount of disk paging file memory in use.
7. The total amount of disk paging file memory.

```
memorystatus nPercentInUse [, nTotalVirtual, nAvailVirtual,
nTotalPhysical, nAvailPhysical, nTotalPageFile, nAvailPageFile]
```

The following code determines the percentage of memory in use.

```
int nPercentMemInUse
memorystatus nPercentMemInUse
```

Some of the values in *memorystatus* have meaning only for Microsoft Windows.

Determining the Operating System

The *os* statement or function returns the name of the operating system. No parameters are required.

```
os
string os()
```

The result is an operating system name, such as “Microsoft Windows NT Workstation”. The statement form of *os* pushes the operating system name onto the stack; the function returns the operating system name as its result. The following code uses *os* to check the operating system for Microsoft Windows.

```
if contains(os(), "Microsoft Windows")
{
     Folder = "c:\\Documents"
}
else
{
     Folder = "/Documents"
}
```

The *os* function is useful for conditionally handling functions that are operating system-specific.

Determining the Operating System Version

The *osversion* statement or function returns the version of the operating system. No parameters are required.

```
osversion
string osversion()
```

The result is a version number string, such as “5.0.2195”. The statement pushes the version onto the stack; the function returns the version as its result. The following code uses *osversion* to check the operating system version.

```
if contains(os(), "Microsoft Windows") && val(osversion())>=5
{
     bWin2000 = true
}
```

The *osversion* function is useful for conditionally handling functions that are specific to certain operating system versions.

Determining User Permission Groups

The *usergroups* statement or function creates a list of permission groups for a user. The parameters are a user name and an optional server to interrogate, which defaults to the current system if omitted.

```
usergroups username [, server]
boolean usergroups(username [,server])
```

If the specified user is recognized, a success condition results (the function returns true) and a list of permission groups is pushed onto the stack as a single string, in which each permission group is separated from the next by a newline. If the specified user is not recognized, the result is false. The following code obtains a list of permission groups for a user, then iterates through them.

```
if usergroups("jsmith")
{
    while nextline(group)
    {
        show group
    }
}
```

There is also a statement form of *usergroups*.

```
usergroups user [, server]
```

The *usergroups* function or statement is useful for determining whether or not a user has permission for an action.

Tutorial: Restart

Now that we know something about system actions and obtaining system information, let's see them at work in an NQL script. In this tutorial, you will create a script whose purpose is to restart the computer. The script will do so only if the user authorizes the restart. Moreover, the script will offer that option only if the user is an administrator. The script will demonstrate obtaining system information by determining the permission groups the user belongs to. The script will demonstrate system actions by restarting the computer.

There are four steps in this tutorial:

1. Launch the NQL development environment.
2. Enter the Restart script.

3. Run the Restart script.
4. Understand the Restart script.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Check the permissions groups a user belongs to.
- Restart a computer.

Let's begin!

Step 1: Launch the NQL Development Environment

Launch the NQL development environment. On a Windows system, this can be accomplished by clicking on the NQL Client desktop icon, or by selecting *Program Files, Network Query Language, NQL Client* from the Start Menu. On other platforms, you should have a desktop icon and/or a command-line method of launching the NQL Client.

At this point, you should have the NQL development environment active on your desktop, with an empty code window. Now you are ready to enter the tutorial script.

Step 2: Enter the Restart Script

In the NQL development environment, enter the script shown in Listing 30.1 and save it under the name Restart.nql. Enter the script, then save it by clicking the *Save* toolbar button (which has a disk icon).

If you prefer, you may copy Restart.nql from the companion CD. If you have installed the companion CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. Click the *Open* toolbar button (folder icon), and select Restart.nql from the Tutorials\ch30 folder.

Set the user name in the *usergroups* statement from *dpallmann* to your user name.

At this point, the script in Listing 30.1 should be entered and saved, either because you entered it by hand or because you opened the script from the companion CD.

Step 3: Run the Restart Script

Now run the script. You can do this by selecting *Build, Run* from the menu, clicking the *Run* toolbar button, or pressing F5. If you are logged in as an administrator, the script will ask the following:

```
You are an administrator. Would you like to restart the system?
```

If you click the *Yes* button the system will reboot (make sure you've saved everything first if you try this). If you click *No*, the script ends without rebooting.

If you are not an administrator, the script does not offer the choice of rebooting and instead displays this message:

```
//Restart - restarts the system, but only if the user is an
administrator and gives approval.

usergroups "dpallmann"
contains "Administrator"
then
{
    ask "Restart System?", "You are an administrator. Would you like
to restart the system?"
    then
    {
        restart
    }
    else
    {
        show "Okay, no action taken. Good bye."
    }

}
else
{
    show "Sorry, you are not an administrator. You cannot restart the
system."
}
```

Listing 30.1 Restart script.

```
Okay, no action taken. Good bye.
```

If this does not happen, check the following:

- Make sure you have entered the script correctly.
- Check the NQL client's *Errors* tab for error messages.

At this point you should have seen the Restart script run, checking a user's permissions and rebooting a computer if given authorization.

Step 4: Understand the Restart Script

We now want to make sure we understand every part of the Restart script. The first line is a comment line.

```
//Restart - restarts the system, but only if the user is an
administrator and gives approval.
```

The *usergroups* statement looks up permissions for the stated user name. A list of permission groups is pushed onto the stack.

```
usergroups "dpallmann"
```

If the list of permission groups contains the text *Administrator*, the system proceeds to ask for authorization to reboot with *ask*. If a positive response is received, the *restart* statement reboots the system.

```
contains "Administrator"
then
{
    ask "Restart System?", "You are an administrator. Would you like to
restart the system?"
    then
    {
        restart
    }
    else
    {
        show "Okay, no action taken. Good bye."
    }

}
```

If the user is not an administrator, the script apologizes but doesn't offer to reboot the computer.

```
else
{
    show "Sorry, you are not an administrator. You cannot restart the
system."
}
```

At this point, you've seen an NQL Web check user permissions and restart a computer.

Chapter Summary

Network Query Language can perform a number of system actions:

- The *logoff* statement logs off the current user.
- The *powerdown* statement powers off the computer.
- The *reboot* statement reboots the computer.
- The *restart* statement reboots the computer.
- The *settime* statement sets the system date/time.
- The *shutdown* statement shuts down the computer.

NQL can obtain various kinds of system information:

- The *appname* statement returns the current application name.
- The *folder* statement returns the specific path of a logical folder.
- The *memorystatus* statement returns information about available virtual memory.
- The *os* statement returns the operating system name.
- The *osversion* statement returns the operating system version.
- The *usergroups* statement returns the permission groups a user belongs to.

The NQL support for system actions and system information allows scripts to detect and interact with their environment.

CHAPTER

31

Calling NQL as a Component

Network Query Language can be called from other programming languages. By treating NQL as a component, you can continue to use your favorite programming language and call upon NQL when needed. This chapter describes the interfaces provided for calling NQL from other development environments. The tutorial at the end of the chapter uses NQL to send e-mail, called from within a Visual Basic program.

Calling NQL from Other Languages

Many of today's programming languages support the notion of components that add new functionality as add-ons. There are various kinds of components in popular use today, such as ActiveX controls, COM components, Java Beans, and others. NQL for Windows provides four programming language interfaces: ActiveX controls, COM components, a Java class, and a C++ class.

Table 31.1 lists some popular development environments and the suggested interface for accessing NQL.

For platforms other than Windows, your interface choices are limited to using the Java edition of NQL with the NQL Java class.

Table 31.1 Recommended NQL Interfaces for Windows

DEVELOPMENT ENVIRONMENT	PLATFORM	RECOMMENDED INTERFACE
Active Server Pages	Windows	NQL COM interface
C	Windows	NQL DLL interface
C++	Windows	CNQL C++ class
Delphi	Windows	NQL ActiveX interface
Microsoft Office	Windows	NQL ActiveX interface
HTML VBScript client	Windows	NQL ActiveX interface
Java	Any	NQL Java class
Visual Basic	Windows	NQL ActiveX interface
Visual Basic for Applications	Windows	NQL ActiveX interface

The ActiveX Control

The NQL ActiveX control allows Network Query Language to be called from many programming languages and development environments on the Windows platform. Through the ActiveX interface, languages such as Visual Basic, Visual C++, and Delphi can add an NQL control to their projects. In addition to formal programming languages, applications such as Microsoft Word and Microsoft Excel (which contain Visual Basic for Applications) can access NQL through the ActiveX interface.

The NQL ActiveX control is named nql.ocx and is installed with the Windows edition of NQL. Like most ActiveX controls, nql.ocx provides properties, methods, and events.

Properties

The NQL ActiveX control exposes the properties described in Table 31.2, in addition to those ActiveX properties common to all controls.

Generally, the properties are used in the following sequence:

1. Prior to executing a script, set the *Input*, *LogErrors*, *LogExecution*, *RunID*, and *Script* properties.
2. Invoke the *RunScript* or *RunScriptFile* method to execute a script.
3. Examine the *ErrorCount*, *ErrorMessages*, *Log*, and *Output* properties.

Methods

The *NQL.OCX* methods are described in Table 31.3.

There is a *RunScript* method for running direct script code, and a *RunScriptFile* method for running a script file. The general procedure for executing a script is as follows:

Table 31.2 NQL ActiveX Control Properties

PROPERTY	DESCRIPTION	EXAMPLE (VB)
ErrorCount	The number of errors that occurred in the last script execution (read-only integer).	nErrors = NQL1.ErrorCount
ErrorMessages	The text of any errors that occurred in the last script execution. This property is set only if *LogErrors* is set to true before running a script (read-only string).	MsgBox NQL1.ErrorMessages, 0, “Errors”
Input	The input XML stream to pass on to scripts (string).	NQL1.Input = “<CUSTID>001045 </CUSTID>”
Log	The execution log from the last script execution. This property is set only if *LogExecution* is set to true before running a script (read-only string).	MsgBox NQL1.Log, 0, “Log”
LogErrors	A flag indicating whether error messages should be collected during script execution and stored in the *ErrorMessages* property (Boolean, default is false).	NQL1.LogErrors = true
LogExecution	A flag indicating whether an execution log should be maintained during script execution and stored in the *Log* property (Boolean, default is false).	NQL1.LogExecution = false
Output	The output XML stream returned by the last script that was executed (read-only string).	MsgBox NQL1.Output, 0, “Results”
RunID	An optional identifier to assign to an execution thread (integer, default is 0).	NQL1.RunID = nThreadID
Script	The script code to execute or a file specification. If the value contains a newline, it is assumed to be script code rather than a file specification (string).	NQL1.Script = txtScript.Text

1. Prior to executing a script, set the *Input*, *LogErrors*, *LogExecution*, and *RunID* properties as desired.
2. Invoke the *RunScript* or *RunScriptFile* method to execute a script, specifying the script code or the script file name as a parameter to the method.
3. Examine the *ErrorCount*, *ErrorMessages*, *Log*, and *Output* properties as desired.

An alternate method for running scripts, *Run*, uses the pre-loaded property *Script* instead of accepting a string argument. Some development environments, such as

Table 31. 3 NQL ActiveX Control Methods

METHOD	DESCRIPTION	EXAMPLE (VB)
Run	Executes the script contained in the *Script* property	NQL1.Script = “show ‘hello’\n”
		NQL1.Run
RunScript	Executes the script contained in the method parameter (a string)	NQL1.RunScript (sScriptCode)
RunScriptFile	Executes the script specified by the file name in the method parameter (a string)	NQL1.RunScriptFile (sFilespec)

VBScript, are unable to pass string parameters directly to an ActiveX control. In these cases, the *Script* property and the *Run* method should be used. The procedure for using *Script* and *Run* is as follows:

1. Prior to executing a script, set the *Input*, *LogErrors*, *LogExecution*, and *RunID* properties as desired.
2. Set the script code to execute in the *Script* property.
3. Invoke the *Run* method to execute the script.
4. Examine the *ErrorCount*, *ErrorMessages*, *Log*, and *Output* properties as desired.

Events

The *NQL.OCX* control provides the events described in Table 31.4.

The events may be used in the following ways:

- To take action when any script completes execution, place code in the *End* event.
- To respond to real-time notification, place code in the *Notify* event.
- To provide an implementation for the *show* statement, place code in the *Show* event.

The following code shows a Visual Basic program calling the NQL ActiveX control.

```
Private Sub Form_Load()
    NQL1.Script = "openmail" & vbCrLf & _
                  "sendmessage '" & sAddress & "','reminder'," & _
                      "Your account is still past due" & vbCrLf & _
                  "closemail" & vbCrLf
    If NQL1.Run() Then
        MsgBox "The message has been sent"
    Else
        MsgBox "The message could not be sent"
```

Table 31.4 NQL ActiveX Control Events

EVENT	DESCRIPTION	EXAMPLE (VB)
End	Fires when a script completes execution.	Private Sub NQL1_End()
		CmdRun.Enabled = True
		End Sub
Notify	Fires when an NQL notify statement is executed. The script's Run ID and the notification message are passed as parameters.	Private Sub NQL1_Notify (ByVal RunID As Integer, Message As String)
		MsgBox Message, vbInformation, "Notification"
		End Sub
Show	Fires when an NQL show statement is executed. The script's Run ID and the display message are passed as parameters.	Private Sub NQL1_Show (ByVal RunID As Integer, Message As String)
		MsgBox Message, vbInformation, "Message"
		End Sub

```
    End If
End Sub
```

The COM Interface

The ATL COM interface (NQLATL.DLL) is a lightweight alternative to the ActiveX interface. It provides a component access to NQL that is easily accessed by many environments, and is especially recommended for multi-threaded server environments, such as Active Server Pages.

Properties

The NQLATL.DLL component exposes the properties described in Table 31.5.

Generally, the properties are used in the folllowing sequence:

1. Create an "nqlatl.nqlcom.1" object.
2. Prior to executing a script, set the *Input*, *LogErrors*, *LogExecution*, *RunID*, and *Script* properties as desired.
3. Invoke the *NQLRunScript*, *NQLRunScriptFile*, or *NQLRun* method to execute a script.
4. Examine the *ErrorCount*, *ErrorMessages*, *Log*, and *Output* properties as desired.

Table 31.5 NQL COM Interface Properties

PROPERTY	DESCRIPTION	EXAMPLE (VB)
ErrorCount	The number of errors that occurred in the last script execution (read-only integer).	nErrors = oNQL.ErrorCount
ErrorMessages	The text of any errors that occurred in the last script execution. This property is set only if *LogErrors* is set to true before running a script (read-only string).	MsgBox oNQL.ErrorMessages, 0, "Errors"
Input	The input XML stream to pass on to scripts (string).	oNQL.Input = "<CUSTID>001045 </CUSTID>"
Log	The execution log from the last script execution. This property is set only if *LogExecution* is set to true before running a script (read-only string).	MsgBox oNQL.Log, 0, "Log"
LogErrors	A flag indicating whether error messages should be collected during script execution and stored in the *ErrorMessages* property (Boolean, default is false).	oNQL.LogErrors = true
LogExecution	A flag indicating whether an execution log should be maintained during script execution and stored in the *Log* property (Boolean, default is false).	oNQL.LogExecution = false
Output	The output XML stream returned by the last script that was executed (read-only string).	MsgBox oNQL.Output, 0, "Results"
Script	The script code to execute or a file specification. If the value contains a newline, it is assumed to be script code rather than a file specification (string).	oNQL.Script = txtScript.Text

Methods

The NQLATL.DLL component uses the methods described in Table 31.6.

The *NQLRunScript* method runs direct script code, and the *NQLRunScriptFile* method runs a script file. The general procedure for executing a script is as follows:

1. Prior to executing a script, set the *Input*, *LogErrors*, and *LogExecution* properties.
2. Invoke the *NQLRunScript* or *NQLRunScriptFile* method to execute a script, specifying the script code or the script file name as a parameter to the method.

Table 31.6 NQL COM Interface Methods

METHOD	DESCRIPTION	EXAMPLE (VB)
NQLRun	Executes the script contained in the *Script* property	oNQL.Script = "show 'hello'\n" oNQL.NQLRun
NQLRunScript	Executes the script contained in the method parameter (a string)	oNQL.NQLRunScript(sScriptCode)
NQLRunScriptFile	Executes the script specified by the file name in the method parameter (a string)	oNQL.NQLRunScriptFile(sFilespec)

3. Examine the *ErrorCount*, *ErrorMessages*, *Log*, and *Output* properties.

An alternate method, *NQLRun*, uses the pre-loaded property *Script* instead of accepting a string argument. Some development environments, such as VBScript, are unable to properly pass string parameters directly to a method—in these cases, the *Script* property and the *NQLRun* method should be used.

The Java Class

For Java programs, NQL provides nql.java and nql.class, a class that calls the NQL DLL using the Java Native Interface (JNI). The class is called nql; its full designation, including the package name, is ac.nql. Your calling program can supply a script or a script file name to the NQL engine. The calling program may also pass variable values to the script and retrieve results from the script.

Required Files

The NQL Java interface files shown in Table 31.7 are initially installed under the \nql\java directory; you will need to move these files to the appropriate locations on your system.

Table 31.7 NQL Java Class Files

FILE	PURPOSE
nql.java	nql class (source code)
nql.class	nql class (binary)
nqljni.dll	Connects Java Native Interface to NQL

The nql class files are nql.java and nql.class. If you place nql.class in a search path location for classes, you can use a Java *import* statement to reference the ac.nql class. Alternately, you can add nql.java directly to your project. There is also a third file (nqljni.dll) which must be installed in your Java library path.

Fields

The ac.nql class contains the fields described in Table 31.8.

Generally, the fields are used in the following sequence:

Table 31.8 NQL Java Class Fields

PROPERTY	DESCRIPTION	EXAMPLE (JAVA)
nErrorCount	The number of errors that occurred in the last script execution (read-only integer).	int nErrors = nql1.nErrorCount;
nErrorMessages	The text of any errors that occurred in the last script execution. This property is set only if *LogErrors* is set to true before running a script (read-only string).	String sErrorMsgs = nql1.sErrorMessages;
sInput	The input XML stream to pass on to scripts (string).	nql1.sInput = "<COUNT> 5</COUNT>";
sLog	The execution log from the last script execution. This property is set only if *LogExecution* is set to true before running a script (read-only string).	String sLog = nql1.sLog;
bLogErrors	A flag indicating whether error messages should be collected during script execution and stored in the *ErrorMessages* property (Boolean, default is false).	nql1.bLogErrors = true;
bLogExecution	A flag indicating whether an execution log should be maintained during script execution and stored in the *Log* property (Boolean, default is false).	nql1.bLogExecution = false;
sOutput	The output XML stream returned by the last script that was executed (read-only string).	String sOutput = nql1.sOutput;
nID	An optional identifier to assign to an execution thread (integer, default is 0).	Nql1.nID = nThreadID;

1. Prior to executing a script, set the *sInput*, *bLogErrors*, *bLogExecution*, and *nID* properties.
2. Invoke the *RunScript* method to execute a script.
3. Examine the *nErrorCount*, *sErrorMessages*, *sLog*, and *sOutput* properties.

Methods

The ac.nql class contains a single method, *runScript*, shown in Table 31.9. The *RunScript* method accepts NQL script code as a string parameter, returning the error count after the script completes execution.

Using the ac.nql Class

To execute an NQL script:

1. Import the ac.nql class with an import statement (or, add nql.java directly to your project).

   ```
   import ac.nql;
   ```
2. Create a new instance of the .nql class.

   ```
   nql n = new nql();
   ```
3. Set any desired pre-execution fields, such as *sInput*, *bLogExecution*, *bLogErrors*, and *nID*.

   ```
   n.sInput = "<ID>5</ID><NAME>ACME Trucking</NAME>
   ```
4. Call the *RunScript* method to execute the script and return an error count.

   ```
   int nErrors = n.runScript("...NQL script...");
   ```
5. Examine any desired post-execution fields, such as *nErrorCount*, *sLog*, *sOutput*, and *sErrorMessages*.

   ```
   String results = n.sOutput;
   ```

The C++ Interface

For C++ programs, NQL provides CNQL, a wrapper class that calls the NQL DLL (nql.dll). Your calling program can supply a script or a script file name to the NQL

Table 31.9 NQL Java Class Methods

METHOD	DESCRIPTION	EXAMPLE (JAVA)
runScript	Executes the script contained in the method parameter (a string)	int nErrors = nql1.runScript(sScriptCode);

engine. The calling program can also pass variable values to the script and retrieve results from the script.

Calling NQL.DLL to Run a Script

The function declarations for nql.dll are contained in the file nql.h. Use an *include* directive to reference this file in your source code:

```
#include <nql.h>
```

Rather than making direct calls to the DLL, nql.h defines a handy class (CNQL) that makes it easy to run NQL scripts under program control.

At its simplest level, running an NQL script consists of nothing more than creating an instance of CNQL and calling its *RunScript* function, passing an NQL script to execute:

```
#include <nql.h>
    ...
CNQL nql;
nql.RunScript(sScript);
```

Checking for Errors

The CNQL class's *RunScript* function returns an integer that is the number of errors encountered. A value of zero indicates that no errors occurred.

```
CNQL nql;
if (nql.RunScript(sScript)>0)
... errors occurred ...
```

If errors are reported, the CNQL class's *sErrMsg* member function contains error message text.

Passing Input to an NQL Script (XML)

NQL has facilities for calling programs to pass data to NQL scripts using XML.

To pass an XML string, set the *sInput* member variable to an XML string prior to calling *RunScript*:

```
CNQL nql;
nql.sInput = "<AUTHOR>Twain</AUTHOR><TITLE>Huckleberry Finn</TITLE>";
if (nql.RunScript(sScript)>0)
    ... errors occurred ...
```

Passing Input to an NQL Script (Variable Name-Value Pairs)

Formatting input to NQL in XML form may be cumbersome. The CNQL class includes a *SetVar* member function that builds the XML input for you. The program simply calls *SetVar* one or more times, passing names and values.

```
CNQL nql;
nql.SetVar("AUTHOR", "Twain");
nql.SetVar("TITLE", "Huckleberry Finn");
if (nql.RunScript(sScript)>0)
    ... errors occurred ...
```

This example is functionally identical to the example in the preceding section.

Retrieving Output from an NQL Script (XML)

NQL has facilities to return data to calling programs using XML. To retrieve XML output as a string, access the *sOutput* member variable after calling *RunScript*:

```
CNQL nql;
if (nql.RunScript(sScript)>0)
    ... errors occurred ...
MessageBox(nql.sOutput, "XML Output From Script", 0);
```

Retrieving Output from an NQL Script (Variable Name-Value Pairs)

Deciphering output from NQL in XML form may be cumbersome. The CNQL class includes a *GetVar* member function that parses the XML output for you. The program simply calls *GetVar* one or more times, passing names and variables to receive values.

```
CNQL nql;
if (nql.RunScript(sScript)>0)
... errors occurred ...
sAuthor = nql.GetVar("AUTHOR");
sTitle = nql.GetVar("TITLE");
```

Retrieving Multiple-Record Output from an NQL Script (Variable Name-Value Pairs)

If the XML output consists of one or more record pairs, it is necessary to advance to the next record after retrieving all wanted variables. The *GetRecord* function moves to the next record that has beginning and ending tags for the name specified. If there are no more records, *GetRecord* returns false.

Imagine that you are running a script that retrieves art pieces from an online art print site, and that the output looks like this:

```
<Piece><title>Self-portrait</title><price>$255.00</price></Piece>
<Piece><title>Lady in waiting</title><price>$65.00</price></Piece>
<Piece><title>Blue Boy</title><price>$400.00</price></Piece>
```

The following code fragment submits an artist name to the script as input and retrieves title and price information from the output. Any number of output records is accommodated through the use of the *GetRecord* statement.

```
CNQL nql;
nql.SetVar("artist", "Monet");
nql.RunScript(sArtSiteScript);
CString sTitle;
CString sPrice;
while (nql.GetRecord("Piece"))
{
    sTitle = nql.GetVar("title");
    sPrice = nql.GetVar("price");
    ... do something with sTitle and sPrice ...
} // end while
```

Notification Callback Functions

NQL scripts can pass back output in real-time before they have completed executing. This is useful for applications that need to pass back some results quickly. If the CNQL class is given a notification callback routine, the NQL *notify* statement passes output to the routine in real time, then continues with script execution. A notification callback routine has the following form. The *CString** parameter is the script's output to that point.

```
void Notify (CString* sValue)
{
    ... function body ...
}
```

To inform the CNQL class of a notification callback routine, its *pNotifyFunc* member is set prior to calling *RunScript*.

```
CNQL nql;
nql.sInput = sInputXML;
nql.pNotifyFunc = Notify;
nql.RunScript(sNQL);
sOutputXML = nql.sOutput;
```

Show Callback Functions

NQL scripts can ask for the display of messages as they execute, which the calling program may elect to implement as a message display or some other appropriate implementation (including ignoring the request in the case of non-interactive software). If the CNQL class is given a *show* callback routine, it is called after a *show* statement is encountered in the script that is executing. A *show* callback routine has the following form. The *CString** parameter is a pointer to the message to be displayed.

```
void ShowRoutine (CString* sMessage)
{
    ... function body ...
}
```

To inform CNQL of a *show* callback routine, its *pShowFunc* member is set prior to calling *RunScript*.

```
CNQL nql;
nql.sInput = sInputXML;
nql.pShowFunc = ShowRoutine;
nql.RunScript(sNQL);
sOutputXML = nql.sOutput;
```

Monitor Callback Functions

NQL scripts can be monitored as they execute, so that the calling program can monitor elapsed time and have the option of aborting execution if desired. This is useful for aborting scripts that are taking too long or handling the situation where a user wishes to cancel. If the CNQL class is given a *monitor* callback routine, it is called after each NQL statement executes to report elapsed time and query whether execution should continue. A *monitor* callback routine has the following form. The integer parameter is the elapsed time in seconds since the script began execution. If the function returns true, script execution continues. If the function returns false, script execution halts.

```
BOOL Monitor (int nElapsedTime)
{
    ... function body ...
    ... return true or false ...
}
```

To inform CNQL of a *monitor* callback routine, its *pMonitorFunc* member is set prior to calling *RunScript*.

```
CNQL nql;
nql.sInput = sInputXML;
nql.pMonitorFunc = Monitor;
nql.RunScript(sNQL);
sOutputXML = nql.sOutput;
```

Calling NQL from Other Environments: NQL.DLL

If an environment cannot access NQL via the ActiveX control interface, the C++ CNQL class, or the Java NQL class, another option is to call the NQL DLL directly. Your calling program can supply a script or a script file name to the NQL engine. The calling program may also pass variable values to the script and retrieve results from the script.

Calling NQL from C++

The function declarations for the NQL DLL are contained in the file nql.h, shown in Listing 31.1. The nql.h file, which defines the CNQL C++ class, also contains the entry point declarations for the NQL DLL and demonstrates how the NQL DLL can be accessed.

```
//---- nql.h - general include file for calling nql.dll from a VC++
program -----
typedef void (*ShowCallback)( CString* );
typedef void (*NotifyCallback)( CString* );
typedef BOOL (*MonitorCallback)( int );
typedef INT (*TraceCallback)( CString* /* location */, CString* /*
statement */,
CString* /* reserved */, CString* /* reserved */,
CString* /* reserved */, CString* /* reserved */,
CString* /* reserved */, CString* /* reserved */);
_declspec(dllexport) int WINAPI nqlGetVersion(CString& sVersion);
_declspec(dllexport) int WINAPI nqlRunScript(const CString& sScript);
_declspec(dllexport) int WINAPI nqlRunScriptEx(const CString& sScript,
const CString& sInput, CString& sOutput, CString* psErrors = NULL,
CString* psLog = NULL, NotifyCallback pNotifyFuncAddress = NULL,
MonitorCallback pMonitorFuncAddress = NULL,
TraceCallback pTraceFuncAddress = NULL,
ShowCallback pShowFunc = NULL);
_declspec(dllexport) BOOL WINAPI nqlSetVar(const CString& sName,
const CString &sValue, CString &sInput);
_declspec(dllexport) BOOL WINAPI nqlGetVar(const CString& sName,
CString &sValue,
CString &sOutput);
_declspec(dllexport) BOOL WINAPI nqlNextRecord(const CString&
sRecordTag,
CString &sOutput);
```

Listing 31.1 Contents of nql.h header file.

```
_declspec(dllexport) int WINAPI nqlRunImmigrantScript(const CString&
sScript);
_declspec(dllexport) int WINAPI nqlRunScriptFromFile(const CString&
sFilespec);
_declspec(dllexport) int WINAPI nqlRunScriptFromFileEx(const CString&
sFilespec,
const CString& sInput, CString& sOutput, CString* psErrors = NULL,
CString* psLog = NULL, NotifyCallback pNotifyFuncAddress = NULL,
MonitorCallback pMonitorFuncAddress = NULL,
TraceCallback pTraceFuncAddress = NULL, ShowCallback pShowFunc =
NULL);

//CNQL class - front ends calls to nql.dll

class CNQL
{
public:
    CNQL();
    void SetVar(CString sName, CString sValue);
    int RunScript(CString sScript);

    int RunScriptFromFile(CString sScriptFile);
    CString GetVar(CString sName);
    BOOL GetRecord(CString sRecordTag);
public:
    BOOL bLogExecution, bLogErrors;
    CString sInput, sOutput;
    CString sErrors, sLog;
    NotifyCallback pNotifyFunc;
    MonitorCallback pMonitorFunc;
    TraceCallback pTraceFunc;
    ShowCallback pShowFunc;
};

CNQL::CNQL()
{
    pNotifyFunc = NULL;
    pMonitorFunc = NULL;
    pTraceFunc = NULL;
    pShowFunc = NULL;
    bLogErrors = false;
    bLogExecution = false;
}

void CNQL::SetVar(CString sName, CString sValue)
{
```

continues

Listing 31.1 Contents of nql.h header file (continued).

```
    nqlSetVar(sName, sValue, sInput);
}

int CNQL::RunScript(CString sScript)
{
    CString *psErrors = NULL;
    if (bLogErrors) psErrors = &sErrors;
    CString *psLog = NULL;
    if (bLogExecution) psLog = &sLog;
    return nqlRunScriptEx(sScript, sInput, sOutput, psErrors, psLog,
        pNotifyFunc, pMonitorFunc, pTraceFunc, pShowFunc);
}

int CNQL::RunScriptFromFile(CString sScriptFile)
{
    CString *psErrors = NULL;
    if (bLogErrors) psErrors = &sErrors;
    CString *psLog = NULL;
    if (bLogExecution) psLog = &sLog;
    return nqlRunScriptFromFileEx(sScriptFile, sInput, sOutput,
        psErrors, psLog, pNotifyFunc, pMonitorFunc,
        pTraceFunc, pShowFunc);
}

CString CNQL::GetVar(CString sName)
{
    CString sValue;
    nqlGetVar(sName, sValue, sOutput);
    return sValue;
}

BOOL CNQL::GetRecord(CString sRecordTag)
{
    return nqlNextRecord(sRecordTag, sOutput);
}
```

Listing 31.1 Contents of nql.h header file (continued).

Calling NQL from Non-Object-Oriented Languages

If an environment cannot read and write *CString* objects, the generic interface provides an API that utilizes C-style null-terminated string parameters and the _stdcall calling convention. Most older languages should be able to communicate with the NQL DLL generic interface using these older calling standards.

With the NQL generic interface, your calling program can supply a script or a script file name to the NQL engine. It may also pass variable values to the script and retrieve results from the script. Also, the NQL DLL can temporarily return control to your calling program through various callback interfaces.

The function declarations for NQL DLL are contained in the file nqlG.h, shown in Listing 31.2. The nqlG.h file also contains the entry point declarations for the NQL DLL and demonstrates how the NQL DLL can be accessed.

```
//---- nqlG.h - general include file for calling nql.dll from non-
// object-oriented languages -----
typedef void (__stdcall *ShowCallbackG)( int, char * );
typedef void (__stdcall *NotifyCallbackG)( int, char * );
typedef bool (__stdcall *MonitorCallbackG)( int, int );
typedef int (__stdcall *TraceCallbackG)( int,
char * /* location */, char * /* statement */,
char * /* reserved */, char * /* reserved */,
char * /* reserved */, char * /* reserved */,
char * /* reserved */, char * /* reserved */);
extern int __stdcall nqlGetVersionG(char * pszVersion);
extern int __stdcall nqlRunScriptG(const int nID, const char *
sScript);
extern int __stdcall nqlRunScriptExG(const int nID,
const char * pszScript, const char * pszInput,
char * pszOutput, char * pszErrors, char * pszLog,
NotifyCallbackG pNotifyFunc,
MonitorCallbackG pMonitorFunc,
TraceCallbackG pTraceFunc,
ShowCallbackG pShowFunc,
int nOutputSize,
int nErrorsSize,
int nLogSize);
extern int __stdcall nqlRunScriptFromFileG(const int nID, const char *
pszFilespec);
extern int __stdcall nqlRunScriptFromFileExG(const int nID,
const char * pszFilespec, const char * pszInput,
char * pszOutput, char * pszErrors, char * pszLog,
NotifyCallbackG pNotifyFunc,
MonitorCallbackG pMonitorFunc,
TraceCallbackG pTraceFunc,
ShowCallbackG pShowFunc,
int nOutputSize,
int nErrorsSize,
int nLogSize);
```

Listing 31.2 Contents of nqlG.h header file.

Tutorial: vbNQL

Now that you have had a tour of interfaces to NQL, let's see one in action. In this tutorial, you will call NQL into action from a Visual Basic (VB) program. The VB program will call upon NQL to send a mail message.

There are five steps in this tutorial:

1. Launch the Visual Basic development environment.
2. Create the vbNQL Visual Basic project.
3. Enter the vbNQL program code.
4. Run the vbNQL program.
5. Understand the vbNQL program.

When you are finished with this tutorial, you will have seen how to do the following in NQL:

- Add the NQL ActiveX control to a Visual Basic project.
- Read and write the NQL ActiveX control properties.
- Execute an NQL script.
- Check for success or failure of the script execution.

Let's begin!

Step 1: Launch Visual Basic

Launch Visual Basic, which you will need in order to perform this tutorial.

At this point, you should have the Visual Basic development environment active on your desktop.

Step 2: Create the vbNQL Visual Basic Project

Before we enter any code, we will need to create a project and set up the controls on the main form. Follow the steps to create the vbNQL project.

1. Create a new standard project with a single form. Give the form the caption *VB-NQL Example*.
2. Add a label with the caption *Recipient*. To the right of the label, add a text box named *TxtRecipient*.
3. Add a label with the caption *Subject*. To the right of the label, add a text box named *TxtSubject*.

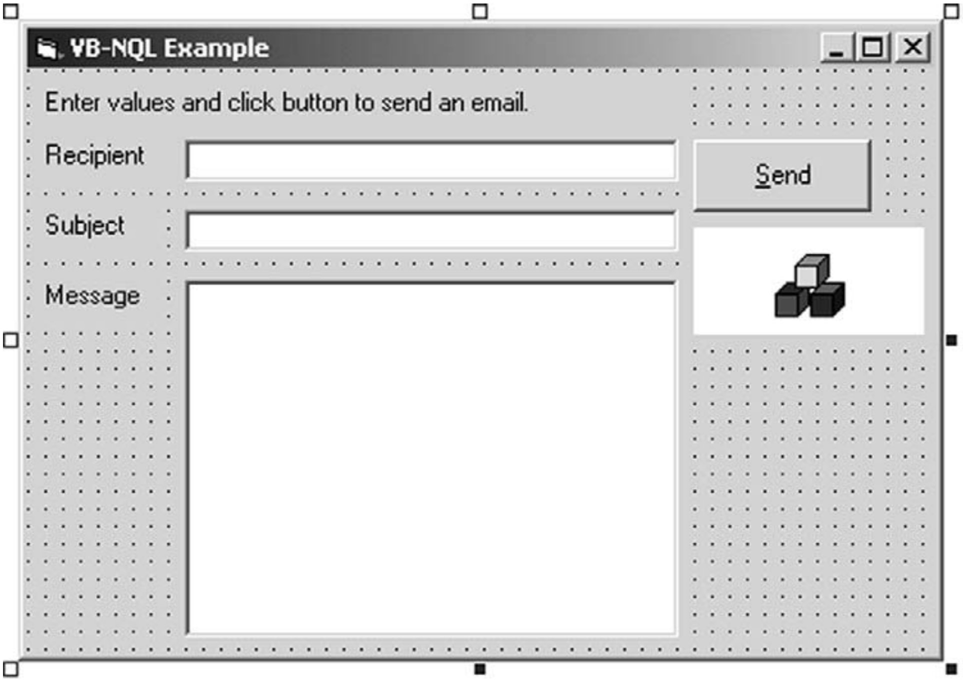
4. Add a label with the caption *Message*. To the right of the label, add a text box named *TxtMessage*. Set the *MultiLine* property to true, and make sure the message box is deep enough to permit multiple-line messages.
5. Add a button. Give it the caption *Send* and the name *CmdSend*.
6. Select *Project, References* from the VB development environment's menu. Select the NQL ActiveX control. When you close the *References* dialog, a new icon appears in Visual Basic's control toolbox.
7. Drag an instance of the NQL ActiveX control onto the form. It will have the name *NQL1*.
8. Save the project and form under the name vbNQL.

At this point, your Visual Basic form should resemble Figure 31.1.

The only remaining task is to enter the Visual Basic program code, which is handled in the next step.

Step 3: Enter the vbNQL Program Code

Click on the *CmdSend* button so that you are in the code editor for the *CmdSend_Click* event. Enter the Visual Basic code shown in Listing 31.3. If you prefer, you may copy the Visual Basic program files from the companion CD. If you have installed the companion



Enter values and click button to send an email.

Recipient

Subject

Message

Figure 31.1 Visual Basic form.

```
Private Sub CmdSend_Click()
    MousePointer = vbHourglass

    NQL1.Script = "openmail" & vbCrLf & _
                  "sendmessage '" & TxtRecipient.Text & "','" & _
                      TxtSubject.Text & "','" & _
                      TxtMessage.Text & "'" & vbCrLf & _
                  "closemail" & vbCrLf

    NQL1.LogErrors = True

    NQL1.Run
    If NQL1.ErrorCount = 0 Then
        MsgBox "The message has been sent"
    Else
        MsgBox "The message could not be sent: " & vbCrLf &
NQL1.ErrorMessages
    End If

    MousePointer = vbNormal
End Sub
```

Listing 31.3 Visual Basic program code.

CD on your system, you will have all of the book's tutorial materials in a \Tutorials directory on your hard drive. The Visual Basic program files are in the Tutorials\ch31 folder.

At this point, the script in Listing 31.3 should be in your code window, either because you entered it by hand or because you opened the script from the companion CD. You are now ready to run the script.

Step 4: Run the vbNQL Program

Now, we are ready to run the program. Before doing so, make sure you have a MAPI mail client active on your desktop, such as Microsoft Outlook. You can do this by selecting *Run, Start* from Visual Basic's menu, clicking the *Run* toolbar button, or pressing F5. The form will display. Enter an e-mail address, a subject, and a message, and click the *Send* button. The success or failure of the message transfer will be displayed. If this doesn't happen, or if it doesn't work, check the following:

- Check the program code for typographical errors.
- Check that you have Outlook or some other mail client up and running on the desktop. The NQL script being called in this program code assumes you have a MAPI mail session already available to connect to.
- If any Visual Basic or NQL error messages are displayed, research their meaning and attempt to resolve them.

At this point you should have seen the vbNQL script run, allowing you to send a mail message by interfacing to NQL to run a mail script. In the next step, we'll dissect the program and explain exactly how it works.

Step 5: Understand the VbNQL Program

We now want to make sure we understand every part of the vbNQL program code. All of the code is in the event handler that Visual Basic runs when the *Send* button is clicked.

```
Private Sub CmdSend_Click()
```

The mousepointer is set to an hourglass so that the user will wait for the processing to come.

```
MousePointer = vbHourglass
```

Next, the NQL script is created and assigned to the *Script* property of the NQL1 object (the NQL ActiveX control). The NQL script performs an *openmail*, a *sendmessage*, and a *closemail*.

```
NQL1.Script = "openmail" & vbCrLf & _
              "sendmessage '" & TxtRecipient.Text & "','" & _
                  TxtSubject.Text & "','" & _
                  TxtMessage.Text & "'" & vbCrLf & _
              "closemail" & vbCrLf
```

The NQL ActiveX control's *LogErrors* property is set to true. If an error occurs running the script, this will allow the program to retrieve description(s) of the error.

```
NQL1.LogErrors = True
```

The script is now executed by invoking NQL1's *Run* method.

```
NQL1.Run
```

To see if the script executed properly, its *ErrorCount* property is checked. A value of 0 means no errors; a negative value indicates compilation errors; a positive value indicates runtime errors. An appropriate message is displayed to the user. If an error occurred, the ActiveX control's *ErrorMessages* property is retrieved and displayed to the user.

```
If NQL1.ErrorCount = 0 Then
    MsgBox "The message has been sent"
Else
```

```
    MsgBox "The message could not be sent: " & vbCrLf &
NQL1.ErrorMessages
End If
```

Processing is complete at this point. The mousepointer is set back to the normal arrow pointer.

```
    MousePointer = vbNormal
End Sub
```

At this point, you've seen a Visual Basic program call upon NQL to run a script.

Chapter Summary

Other development environments can call Network Query Language as a component. This permits you to leverage the power of NQL without giving up your favorite programming language.

The ActiveX control is the most convenient path to NQL for many languages on the Windows platform, including Visual Basic. Once the control is added to a project or form, its properties, methods, and events can be accessed to run NQL scripts. The control is named nql.ocx and requires NQL for Windows.

The ATL COM interface for the Windows platform is especially recommended for development environments that can't work with ActiveX controls, such as Active Server Pages. The COM component is named nqlatl.dll and requires NQL for Windows.

The Java class interface allows Java programs to run NQL scripts; adding the ac.nql class to a project is all that is necessary. The Java interface is platform-independent and can be used with NQL for Windows as well as NQL for Java.

Lower-level programmers on the Windows platform can call NQL directly as a DLL (nql.dll): C++ programs can access CNQL, a wrapper class for nql.dll; C and non-MFC C++ programs have an alternate means of accessing nql.dll through a set of function definitions in nqlG.h.

NQL offers many choices for interoperability with other languages, maximizing your ability to combine the power of NQL with preferred development tools and existing code.

CHAPTER 32

NQL Cookbook

This chapter is a cookbook for NQL: It contains recipes for putting NQL to work. Each task is explained with an itemized procedure and accompanied by a sample script. These scripts are also included on the CD-ROM that accompanies this book.

AddDatabase

Purpose: Adds records to a database.

PROCEDURE

1. Specify the database access method with *dbprotocol*.
2. Open the database with *opendb*.
3. Issue an action query with *execute*, specifying a SQL INSERT query.
4. Close the database with *closedb*.

SCRIPT

```
//AddDatabase - adds records to a database

dbprotocol "odbc"
```

```
DataSourceName = "Customers"
UserID = ""
Password = ""

if opendb(DataSourceName, UserID, Password)
{
    execute "INSERT Customers
(CustomerNumber,LastName,FirstName,Phone,Address,City,State,Zip)
      VALUES (00025,'Jeffries,Jason','631-555-4223',
            '1 Sixth St','Islip','NY',11542)"
    closedb
}
else
{
    show "Error: Unable to open database"
}
```

AppendTextFile

Purpose: Appends to a text file.

PROCEDURE

1. Create and open a file with *append*.
2. Write to the file with *write*.
3. Close the file with *close*.

SCRIPT

```
//AppendTextFile - appends data to a text file

string OutputFile

OutputFile = "myfile.txt"

if append(OutputFile)
{
    write "The three little kittens\r\n"
    write "Lost their mittens\r\n"
    write "And they began to cry,\r\n"
    write "'O mother dear,\r\n"
    write "We greatly fear\r\n"
    write "That we have lost our mittens!'\r\n"
}
close
```

AppendUnicodeTextFile

Purpose: Appends data to a Unicode text file.

PROCEDURE

1. Open a file with *append*, specifying Unicode mode.
2. Write Unicode strings to the file with *write*.
3. Close the file with *close*.

SCRIPT

```
//AppendUnicodeTextFile - appends data to a Unicode text file

string OutputFile

OutputFile = "myfile.txt"

if append(OutputFile, "unicode")
{
    write U"The three little kittens\r\n"
    write U"Lost their mittens\r\n"
    write U"And they began to cry,\r\n"
    write U"'O mother dear,\r\n"
    write U"We greatly fear\r\n"
    write U"That we have lost our mittens!'\r\n"
}
close
```

DeleteDatabase

Purpose: Deletes records from a database.

PROCEDURE

1. Specify the database access method with *dbprotocol*.
2. Open the database with *opendb*.
3. Issue an action query with *execute*, specifying a SQL DELETE query.
4. Close the database with *closedb*.

SCRIPT

```
//DeleteDatabase - deletes records from a database

dbprotocol "odbc"
```

```
DataSourceName = "Customers"
UserID = ""
Password = ""

if opendb(DataSourceName, UserID, Password)
{
    execute "DELETE Customers WHERE Balance = 0.00"
    closedb
}
else
{
    show "Error: Unable to open database"
}
```

FtpReceive

Purpose: Transfers files from an FTP server to a local directory.

PROCEDURE

1. Specify a user name and password with *ftpidentity*.
2. Connect to an FTP server with *ftpopen*.
3. If necessary, change the local working directory with *ftplcd*.
4. If necessary, change the remote working directory with *ftpcd*.
5. Transfer files with *ftpget*.
6. Close the FTP connection with *ftpclose*.

SCRIPT

```
//FtpReceive - receives files from an FTP server

UserName = "anonymous"
Password = "myname@mydomain.com"
Server = "ftp.my-ftp-server.com"
LocalDir = "c:\\ftp"
RemoteDir = ""

ftpidentity UserName, Password

if ftpopen(Server)
{
    ftplcd LocalDir
    ftpcd RemoteDir
    ftpget "*.*"
    ftpclose
}
else
{
    show "Error: unable to connect to FTP server"
}
end
```

FtpSend

Purpose: Transfers files from a local directory over to an FTP server.

PROCEDURE

1. Specify a user name and password with *ftpidentity*.
2. Connect to an FTP server with *ftpopen*.
3. If necessary, change the local working directory with *ftplcd*.
4. If necessary, change the remote working directory with *ftpcd*.
5. Transfer files with *ftpput*.
6. Close the FTP connection with *ftpclose*.

SCRIPT

```
//FtpSend - transfers files to an FTP server

UserName = "anonymous"
Password = "myname@mydomain.com"
Server = "ftp.my-ftp-server.com"
LocalDir = "c:\\ftp"
RemoteDir = ""

ftpidentity UserName, Password

if ftpopen(Server)
{
    ftplcd LocalDir
    ftpcd RemoteDir
    ftpput "*.*"
    ftpclose
}
else
{
    show "Error: unable to connect to FTP server"
}
end
```

GetWeb

Purpose: Retrieves a Web page.

PROCEDURE

1. Retrieve a Web page with the *get* statement, specifying a Web address.
2. Work with the HTML on the stack. For example, use *pop* to put the page into a variable or *match* to extract data from the page.

SCRIPT

```
//GetWeb - get a web page

get "http://www.my-web-site.com"
...work with HTML on the stack...
pop sPage
```

GetWebData

Purpose: Retrieves a Web page and extracts data from it.

PROCEDURE

1. Retrieve a Web page with *get*, specifying a Web address.
2. Extract data from the page using one of the pattern-matching statements. For example, use *match* with a pattern of HTML tags and text. Check the match for success or failure.
3. Work with the extracted data, now in variables.
4. If additional matches are required, repeat steps 2 and 3.

SCRIPT

```
//GetWebData - extract data from a web site

get "http://www.my-retail-site.com/search.dll?title=sportswear"
match -<tr><td>{sku}</td><td>{desc}</td><td>price</td></tr>"
while
{
    show sku, desc, price
    nextmatch
}
```

GetWebFile

Purpose: Retrieves a Web page and stores it as a file.

PROCEDURE

1. Issue a *getfile* statement to retrieve the Web page and save it as a file.
2. Specify a Web address and a local file name.

SCRIPT

```
//GetWebFile - get a web page and save as a file

getfile "http://www.my-web-site.com/products.htm", "c:\\products.htm"
```

QueryDatabase

Purpose: Selects records from a database.

PROCEDURE

1. Specify the database access method with *dbprotocol*.
2. Open the database with *opendb*.
3. Issue a query with *select*, specifying a SQL SELECT query.
4. If a failure condition results, go to step 8.
5. Process the record, whose fields have been mapped to variables.
6. Issue a *nextrecord* statement to advance to the next matching record.
7. Go to step 4.
8. Close the database with *closedb*.

SCRIPT

```
//QueryDatabase - select records from a database

dbprotocol "odbc"

DataSourceName = "Customers"
UserID = ""
Password = ""

if opendb(DataSourceName, UserID, Password)
{
    select "SELECT * FROM NewCustomers WHERE STATE='CA'"
    while
    {
        dumpvars    /* ...do something with database record... */
        nextrecord
    }
    closedb
}
else
{
    show "Error: Unable to open database"
}
```

ReadMail

Purpose: Reads all mail messages.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the POP3 protocol, specify the name of a mail server and any required account information.
3. Read the first mail message using a *firstmessage* statement.
4. Use a *while* loop to process each message. A *nextmessage* statement should appear at the bottom of the *while* loop to advance to the next message.
5. Within the body of the *while* loop, process the message. The message information is stored in these variables: *MessageFrom*, *MessageTo*, *MessageDate*, *MessageSubject*, and *MessageText*.
6. Close the mail session with a *closemail* statement.

SCRIPT

```
//ReadMail - reads all mail messagesw

openmail
firstmessage
while
{
    output MessageFrom, MessageSubject, MessageText
    nextmessage
}
closemail
```

ReadMailFile

Purpose: Reads mail messages with file attachments.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the POP3 protocol, specify the name of a mail server and any required account information.
3. Read the first mail message using a *firstmessage* statement.
4. Use a *while* loop to process each message. A *nextmessage* statement should appear at the bottom of the *while* loop to advance to the next message.

5. Within the body of the *while* loop, save the first attachment with a *firstattachment* statement. Specify the disk directory to save the attachment to.
6. Use an inner *while* loop to process each attachment. A *nextattachment* statement should appear at the bottom of the *while* loop to advance to the next attachment. Again, specify the disk directory to save the attachment to.
7. Within the body of the outer *while* loop, process the message. The message information is stored in these variables: *MessageFrom*, *MessageTo*, *MessageDate*, *MessageSubject*, and *MessageText*.
8. Close the mail session with a *closemail* statement.

SCRIPT

```
//ReadMailFile - reads mail messages with file attachments
openmail
firstmessage
while
{
    output MessageFrom, MessageTo, MessageDate,
    MessageSubject, MessageText
    firstattachment 'c:\\temp\\'
    while
    {
        nextattachment 'c:\\temp\\'
    }
    nextmessage
}
closemail
```

ReadNewMail

Purpose: Reads new (unread) mail messages.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the POP3 protocol, specify the name of a mail server and any required account information.
3. Read the first unread mail message using a *firstunreadmessage* statement.
4. Use a *while* loop to process each message. A *nextmessage* statement should appear at the bottom of the *while* loop to advance to the next unread message.
5. Within the body of the *while* loop, process the message. The message information is stored in these variables: *MessageFrom*, *MessageTo*, *MessageDate*, *MessageSubject*, and *MessageText*.
6. Close the mail session with a *closemail* statement.

SCRIPT

```
//ReadNewMail - reads new (unread) mail messages

openmail
firstunreadmessage
while
{
    output MessageFrom, MessageSubject, MessageText
    nextmessage
}
closemail
```

ReadNewsgroup

Purpose: Reads articles from a newsgroup.

PROCEDURE

1. Open a newsgroup session with an *opennews* statement. Specify the name of a newsgroup server and a newsgroup.
2. Read the first newsgroup message using a *firstarticle* statement.
3. Use a *while* loop to process each message. A *nextarticle* statement should appear at the bottom of the *while* loop to advance to the next newsgroup message.
4. Within the body of the *while* loop, process the message. The message information is stored in these variables: *ArticleDate*, *ArticleFrom*, *ArticleID*, *ArticleSubject*, *ArticleText*.
5. Close the newsgroup session with a *closenews* statement.

SCRIPT

```
//ReadNewsgroup - reads articles from a newsgroup

opennews "news.microsoft.com", "public.microsoft.comp.perl"
firstarticle
while
{
    output ArticleFrom, ArticleSubject, ArticleText
    nextarticle
}
closenews
```

ReadNewsgroupHeaders

Purpose: Reads headers (not full messages) from a newsgroup.

PROCEDURE

1. Open a newsgroup session with an *opennews* statement. Specify the name of a newsgroup server and a newsgroup.
2. Read the first newsgroup message header using a *firstarticleheader* statement.
3. Use a *while* loop to process each header. A *nextarticleheader* statement should appear at the bottom of the *while* loop to advance to the next newsgroup header.
4. Within the body of the *while* loop, process the message header information. The header information is stored in these variables: *ArticleDate*, *ArticleFrom*, *ArticleID*, *ArticleSubject*.
5. Close the newsgroup session with a *closenews* statement.

SCRIPT

```
//ReadNewsgroupHeaders - reads newsgroup headers

opennews "news.microsoft.com", "public.microsoft.comp.perl"
firstarticleheader
while
{
    output ArticleFrom, ArticleSubject
    nextarticleheader
}
closenews
```

ReadTextFile

Purpose: Reads the contents of a text file, a line at a time.

PROCEDURE

1. Open a file for input with *open*.
2. Read lines of text with *read* in a *while* loop until a false value is returned.
3. Close the file with *close*.

SCRIPT

```
//ReadTextFile - reads the contents of a text file, a line at a time

string InputFile, Line

InputFile = "myfile.txt"

if open(InputFile)
{
    while read(Line)
    {
        show Line
    }
}
close
```

ReadUnicodeTextFile

Purpose: Reads the contents of a Unicode text file, a character at a time.

PROCEDURE

1. Open a file for input with *open*, specifying Unicode mode.
2. Read wide characters with *read* in a *while* loop until a false value is returned.
3. Close the file with *close*.

SCRIPT

```
//ReadUnicodeTextFile - reads the contents of a Unicode text file, a
line at a time

string InputFile, Char

InputFile = "myfile.txt"

if open(InputFile, "unicode")
{
    while read(Char)
    {
        show Char
    }
}
close
```

WriteTextFile

Purpose: Creates a text file.

PROCEDURE

1. Create and open a file with *create*.
2. Write to the file with *write*.
3. Close the file with *close*.

SCRIPT

```
//WriteTextFile - creates a text file

string OutputFile
```

```
OutputFile = "myfile.txt"

if create(OutputFile)
{
    write "The three little kittens\r\n"
    write "Lost their mittens\r\n"
    write "And they began to cry,\r\n"
    write "'O mother dear,\r\n"
    write "We greatly fear\r\n"
    write "That we have lost our mittens!'\r\n"
}
close
```

ReadXML

Purpose: Reads data from an XML document.

PROCEDURE

1. Open the XML document (on the Internet, on disk, or on the stack) with *openxml*.
2. Navigate through the XML to desired sections with *firstnode*, *nextnode*, and *closenode*.
3. Extract document data with *getnode* statements.
4. When finished, close the document with *closexml*.

SCRIPT

```
//ReadXML - read data from an XML document

if !openxml("properties.xml")
{
    show "Could not open XML document"
    end
}

boolean bHaveData = firstnode("property")
while bHaveData
{
    getnode listing, desc, area, year, bedrooms, view, pool,
airconditioning
    getnode fireplace, city, state, zip, price
    closenode
...process data in variables...
    bHaveData = nextnode("property")
}

closexml
end
```

SendHtmlMail

Purpose: Sends an HTML mail message.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the SMTP protocol, specify the name of a mail server and any required account information.
3. Compose an HTML mail message and place it on the stack. A *push* statement can be used to put HTML on the stack directly. A *load* statement can be used to load a text message from a disk file.
4. Send the mail message with a *sendmessage* statement, specifying a subject line and recipient name.
5. Close the mail session with a *closemail* statement.

SCRIPT

```
//SendHtmlMail - sends an HTML mail message

openmail
push "<HTML><HEAD><TITLE>Test Message</TITLE></HEAD>
<BODY>This is a test message.<P>The end.<P>"
sendmessage "Test message", "jsmith@mycompany.com"
closemail
```

SendMailWithFile

Purpose: Sends an email message with a file attachment.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the SMTP protocol, specify the name of a mail server and any required account information.
3. Compose a text or HTML mail message and place it on the stack. A *push* statement can be used to put a message on the stack directly. A *load* statement can be used to load a message from a disk file.
4. Specify the file to be attached with a *messageattachment* statement.

5. Send the mail message with a *sendmessage* statement, specifying a subject line and recipient name.
6. Close the mail session with a *closemail* statement.

SCRIPT

```
//SendMailWithFile - sends a mail message with a file attachment

openmail
push "<HTML><HEAD><TITLE>Test Message</TITLE></HEAD>
<BODY>This is a test message.<P>The end.<P>"
messageattachment "report.doc"
sendmessage "Test message", "jsmith@mycompany.com"
closemail
```

SendMultiMail

Purpose: Sends a mail message in multiple formats.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the SMTP protocol, specify the name of a mail server and any required account information.
3. Compose mail messages in as many formats as desired (the possibilities are *text*, *rtf*, and *html*). Place the messages in variables.
4. Send the mail message with a *sendmessage* statement, specifying a subject line, recipient name, and format/message pairs.
5. Close the mail session with a *closemail* statement.

SCRIPT

```
//SendMultiMail - sends a mail message in multiple formats

MsgText = "This is a test message.\r\nThe end.\r\n"

MsgHTML = "<HTML><HEAD><TITLE>Test Message</TITLE></HEAD>
<BODY>This is a test message.<P>The end.<P>"

sendmessage "Test message", "jsmith@mycompany.com", "text", MsgText,
"html" MsgHTML

closemail
```

SendTextMail

Purpose: Sends a text email message.

PROCEDURE

1. Declare the desired mail protocol with a *mailprotocol* statement. This step can be omitted if you are using the default mail protocol (MAPI in the Windows version, POP3/SMTP in the Java version).
2. Open a mail session with an *openmail* statement. For the SMTP protocol, specify the name of a mail server and any required account information.
3. Compose a text mail message and place it on the stack. A *push* statement can be used to put text on the stack directly. A *load* statement can be used to load a text message from a disk file.
4. Send the mail message with a *sendmessage* statement, specifying a subject line and recipient name.
5. Close the mail session with a *closemail* statement.

SCRIPT

```
//SendTextMail - sends a text mail message

openmail
push "This is a test message.\r\nThe end.\r\n"
sendmessage "Test message", "jsmith@mycompany.com"
closemail
```

UpdateDatabase

Purpose: Updates records in a database.

PROCEDURE

1. Specify the database access method with *dbprotocol*.
2. Open the database with *opendb*.
3. Issue an action query with *execute*, specifying a SQL UPDATE query.
4. Close the database with *closedb*.

SCRIPT

```
//UpdateDatabase - updates records in a database

dbprotocol "odbc"

DataSourceName = "Customers"
UserID = ""
```

```
Password = ""

if opendb(DataSourceName, UserID, Password)
{
    execute "UPDATE Customers SET Active = 'N' WHERE OrderCount = 0"
    closedb
}
else
{
    show "Error: Unable to open database"
}
```

WriteTextFile

Purpose: Creates a text file.

PROCEDURE

1. Create and open a file with *create*.
2. Write to the file with *write*.
3. Close the file with *close*.

SCRIPT

```
//WriteTextFile - creates a text file

string OutputFile

OutputFile = "myfile.txt"

if create(OutputFile)
{
    write "The three little kittens\r\n"
    write "Lost their mittens\r\n"
    write "And they began to cry,\r\n"
    write "'O mother dear,\r\n"
    write "We greatly fear\r\n"
    write "That we have lost our mittens!'\r\n"
}
close
```

WriteUnicodeTextFile

Purpose: Creates a Unicode text file.

PROCEDURE

1. Create and open a file with *create*, specifying Unicode mode.
2. Write Unicode string to the file with *write*.
3. Close the file with *close*.

SCRIPT

```
//WriteUnicodeTextFile - creates a Unicode text file

string OutputFile

OutputFile = "myfile.txt"

if create(OutputFile, "unicode")
{
    write U"The three little kittens\r\n"
    write U"Lost their mittens\r\n"
    write U"And they began to cry,\r\n"
    write U"'O mother dear,\r\n"
    write U"We greatly fear\r\n"
    write U"That we have lost our mittens!'\r\n"
}
close
```

WriteXML

Purpose: Generates an XML document.

PROCEDURE

1. Create a file for output with *create*.
2. Write tags and text using the *write* statement.
3. Close the file with *close*.

SCRIPT

```
//WriteXML - create an XML document

create "document.xml"
write '<?xml version="1.0">\r\n'
write "<employee>\r\n"
write "  <name>Lingenfelter, Joel</name>\r\n"
write "  <title>Business Manager</title>\r\n"
write "  <type>Fulltime</type>\r\n"
write "</employee>\r\n"
close
```

APPENDIX

NQL Versions and Editions

Network Query Language is available in a variety of versions and in several editions for different computing platforms. This appendix is a guide to making sense of the various versions and editions of NQL.

This book was written with NQL version 2.0 in mind. If you are using a different version of NQL, this appendix should prove helpful in understanding which parts of the book will apply to the version you are using.

You can find out more about the versions and editions of NQL from the Web sites, www.nqli.com or www.networkquerylanguage.com.

A Brief History of NQL

I conceived of the need for an SQL-like solution for Internet and network programming in 1998. A year later, I decided to stop thinking about the idea and start turning it into reality. Yet another year later, the first version of NQL was commercially released in the Spring of 2000. The company I worked for, originally Alpha Microsystems, changed its name to NQL Inc. that same year.

The original NQL 1.0 was implemented in C++ and optimized for Windows. Much of the rest of 2000 was spent in porting NQL to Java for the sake of other platforms.

NQL 1.0, Java edition, was released in Fall, 2000. Now, the NQL language could be used on all major computing platforms.

Both the Windows and Java editions of NQL received ongoing maintenance and improvements, and in early 2001 NQL 1.1 was released, containing support for new protocols and new features. Performance was also improved.

While this was happening, the design parameters for the next generation of NQL were being decided. Developers liked the unique features, brevity, and power of NQL, but because the language was so simple, some people felt they were making tradeoffs: For every great feature of NQL, they were perhaps giving up something that older, more mature languages had to offer. For example, NQL 1.x had just one data type (string), no user-defined functions, and other limitations that were sometimes a hardship.

In 2001, version 2.0 of NQL came out. It offered all of the power of NQL 1.1 and included some great new features. It also provided more of the features found in other, longer-established languages, including typed variables, true user-defined functions, and support for complex expressions. The compiler and runtime system were completely overhauled to ensure a small memory footprint and full scalability. More internationalization features were added. As of version 2.0, NQL is a grown-up language.

NQL 1.1 vs. NQL 2.0

If you are using NQL 1.x, you will need to know which parts of this book apply to you and which parts don't.

First, the good news: Every major subject area in this book is supported to some degree in NQL 1.x. That means you will be able do such things as crawl Web sites, match patterns, send email, and access databases very effectively in NQL 1.1. NQL 1.1 isn't a bad environment to work in, it just isn't as elegant as 2.0 is.

The bad news is, many of the nice touches in NQL 2.0 that this book describes are not present in NQL 1.1. For example, most statements in NQL 2.0 can also be used as functions, but NQL 1.x doesn't support the concept of functions at all. That means you will have to ignore all of the material about functions in this book. You will also have to change the tutorials to use only statements.

Table A.1 lists the primary differences between NQL 1.x and 2.0.

NQL 1.x has only one loose data type, implemented internally as character strings. NQL 2.0 provides twelve different data types. This might seem like a drastic difference, but in fact NQL 1.x's monotype variables are adequate for most purposes.

NQL 1.x is full of built-in functions. The majority of statements also come in function form so that they can be used in expressions. For user-defined functions, NQL 1.x has a very limited subroutine ability that cannot be used in expressions. NQL 2.0, in contrast, offers full user-defined functions that can return any of the twelve data types and may be used in expressions.

NQL 1.x required you to work with a math mode, so that you were performing only integer or floating-point calculations at any one time based on the current mode. NQL 2.0 works like modern languages, automatically computing result types in expressions

Table A.1 Differences Between NQL 1.x and 2.0

AREA	NQL 1.X	NQL 2.0
Data types	String	Binary, Boolean, Currency, Datetime, Float, Fuzzy, HTML, Integer, Object, String, Unicode, XML
Built-in functions	Not supported	Most statements also available as functions
User-defined functions	Primitive subroutines	Complete user-defined functions
Math mode	Integer or float	No math mode needs to be specified
Web queries	Not present	Present
Equals operator (=)	Same as == (in if and while statements)	Different from =
Editor and debugger	Simple	Enhanced
Code generation	None	.bot file
Desktop exe appearance	Simple window	Deluxe MP3-like appearance with customizable skins

based on the terms in the expression. Thus, 1/3 is 0 while 1.0/3.0 is 0.333333, as is the case with many languages.

NQL 1.x has four kinds of pattern matching for extracting data from Web sites: HTML pattern matching, data pattern matching, HTML table processing, and regular expressions. NQL 2.0 adds a fifth method, Web queries.

The editor and debugger are more elaborate in NQL 2.0. The look and feel of the editor are similar to NQL 1.x, but the general experience is smoother and more polished. NQL 2.0 adds dynamic help and an enhanced user interface. The debugger is quite improved, now positioning you at each source line in the code editor as a script executes. Clicking on an error in the *Errors* tab in the NQL 2.0 development environment now takes you to the offending line in the code editor.

NQL 1.x is an interpreted scripting language, where no code generation normally takes place. NQL 2.0 is also an interpreted scripting language, but it has the ability to generate a binary file called a .bot file. The value of this is that in NQL 2.0 you don't have to put the source code to your scripts on an end-user system.

NQL 1.x can generate a Windows .exe file that encapsulates a script, which is useful for desktop agents that you want to be visible when running. In NQL 1.x, a simple dialog window shows agent status as it runs. In NQL 2.0, you can choose from a selection of different appearances for desktop agents that resemble MP3 players. You can define new skins for the desktop agents to have your own cool look for desktop agents.

Windows Edition versus Java Edition

Network Query Language is available in two editions: NQL for Windows and NQL for Java.

The Windows edition of NQL is itself implemented in C++. This version is slightly more capable and better-performing than the Java edition. There are Windows editions of NQL 1.0, 1.1, and 2.0.

The Java edition of NQL is, as its name implies, a Java implementation. It can run on any platform that supports Java 1.2 or later, which includes all major computing platforms, such as Unix, Linux, Solaris, Windows, and Mac OS X. NQL 1.0 and 1.1 have been implemented in Java as of this writing.

NQL is intended to be uniform regardless of platform, but there are a few capabilities in NQL that are operating-system specific. Some of those areas are not supported in the Java edition. The following capabilities are absent from the Java edition or are different from the Windows edition:

- The browser recorder and the controlled browser are available in the Windows edition only, as they depend on Internet Explorer.
- The e-commerce credit card support is available in the Windows edition only, as it calls upon Windows software from other vendors.
- The email protocols supported by the Windows edition are MAPI, POP3, and SMTP. In the Java edition, only POP3 and SMTP are supported.
- The database protocols supported by the Windows edition are ODBC, OLEDB, and ADO. The database protocol supported by the Java edition is JDBC.
- The application integration capabilities use OLE Automation and are supported only in the Windows edition. The export statements, which use OLE Automation, are likewise available only in the Windows edition.
- The agent characters facility requires Microsoft Agent and is only available in the Windows edition.

With the exception of those areas listed in this appendix, using NQL is a consistent experience, regardless of platform.

Index