Documenting C, C++ and Java Software from GUI to API

by John Darrow

See `http://www.phoons.com/john/classes/aboutbook.html`

DOCUMENTING JAVA AND C++

Preface

How this book is organized

The chapters of this book are organized into these major parts:

Part 1, “Introduction”, page 15

> An overview of who this book is for, what kinds of documents software technical writers create, and how to communicate with "generalized syntax."

Part 2, “Programming concepts”, page 49

> A detailed presentation of the programming concepts fundamental to most software topics that you will document, beginning at the most basic level.

Part 3, “Creating documentation”, page 153

> Instructions on how to create various kinds of software documentation.

Part 4, “Backmatter”, page 227

> Sample code, advanced topics, a glossary, and answers to exercises.

Conventions

Throughout this book, I use the style of writing detailed in "Generalized syntax expressions" on page 31. For a summary, see page 33.

Other books, courses and services

See `http://www.phoons.com/john/`

Table of contents

PART 3—CREATING DOCUMENTATION

PART 4—BACKMATTER

PART 1

Introduction

1

With the technical writer in mind

In this chapter:

The need for software technical writers

Let's get this straight: I'm not talking about writers who can document how to *use* software—there are far too many writers who can do that.

The need is far greater for writers who are able to extract such information on their own directly from the source code and know how to present it effectively in documentation. Can you find the gems in source code? Do you know what such documentation needs to look like? Do you know what questions to ask a programmer? *What are you doing to head in that direction?*

Reasons to read this book

Are any of the following true about you?

- You are curious about the field of software technical writing.
- You are surrounded by those who use programming terminology and you would rather follow along than look and feel lost. You want to be able to communicate better in the language of programmers.
- You want to avoid asking what might be perceived by a programmer as "stupid questions" and have confidence about what you can ask.
- You want a better grasp of the basic concepts and principles of object-oriented programming without having to be a programmer.
- You want the ability to recognize programming expressions in source code and extract as much information as possible directly from the code without annoying the programmer.
- Your company has assigned you a job of documenting software or you want to pursue such a job.
- You want to be able to demonstrate that you know what kind of information is needed by a programming audience and that you can write effectively for such an audience.
- You want to take advantage of the great need (and higher pay) for API writers.

Skip the programming books

Many years ago, a coworker stepped into my office and wanted to learn how to do what I was doing—documenting software. He mentioned several books he'd seen at the book store and asked where he should start. I laughed. All of the books he mentioned were about programming and were written *by* programmers *for* programmers. (Does that sounds like your journey, too?)

That discussion got me thinking. What knowledge would he need to have? Well combination of programming and technical writing knowledge did *I* have that enabled me to skim through source code, pick out the gems and document them effectively? *I knew that he didn't need to be a programmer to do this stuff*. I developed a course on the topic and this book flowed out of the course.

What I believe you need to learn most is how to create software documentation. Sure, you need to learn some programming, but only that which supports the primary goal: learning how to create software documentation. To that end, my focus is on this: *pattern recognition* and a basic understanding of the *concepts behind those patterns*. In the end, you will see that creating API reference documention is actually quite simple, once you know how to find the handful of items in source code that need to be documented.

Job listings and interviews

It is not uncommon for a manager to be delegated a software project that includes an aspect with which the manager is not familiar: the need for API documentation. The manager might not know what kinds of skills to seek in prospective technical writers. Should the manager seek writers who know how to read and write code? Perhaps a fellow programming manager imposed that requirement, believing that such skills are a "must" for creating API reference documentation.

I am glad to regularly see managers in my API documentation courses. They want to get an accurate picture of what it means to create API documentation and see what skills are really needed. I assert that what managers need most are writers who understand certain fundamental programming concepts and source code patterns and can extract necessary information directly from the source code and create a significant portion of the API reference documentation without ever bothering the programmers.

Imagine how impressed the programmer would be to see you show up with a first draft document that shows a detailed, organized presentation of the public variables, methods, functions and constructors of a Java or C++ class. I think the programmer would be more than willing to help provide the descriptions you need for the members you have so effectively identified and presented. In constrast, a writer who has little idea where to begin will have little to no future in this business.

Let’s say you go to an interview for a job whose posting says "read and write code." I believe the best position for you to be in is to know what needs to appear in API reference documentation and to be able to articulate that. You might be asked, "Can you tell me what this code sample does?" With the knowledge you can gain from this book, you could confidently counter, "I know that API reference documentation needs to help the reader learn about the public members of a class. I can look through source code and identify a class’s members—for example, its variables, methods and constructors. And I know how to present that information effectively in documentation so that the reader can quickly find a particular method or function description and know what kind of information it needs to do its job and what it gives back as a result. If your company does not yet have a style for API reference documentation, I would like to get the company off to an excellent start." Isn’t that what managers want most, someone who knows what needs to be done and knows how to do it better than so many other candidates out there who do not have that knowledge?

I have watched students of mine get snatched up for API documentation jobs soon after class, as managers were anxious to find someone who could demonstrate that they knew how to run with the company’s software documentation project.

Equip yourself with the knowledge of this book.

Background you should have

Are you concerned that this book might be too technical for you? Do you think you might need more background before reading this book?

If you can breathe and are eager to learn some technical details, I consider you ready for this book! Sure, I have just used unusual phrases like "API reference docuementation" and "object oriented programming," but don’t worry: I will carefully walk you through what each of these is. I believe you will enjoy the ride and be satisfied at the number of technical concepts you will have absorbed in reading the book—and be excited about your increased job prospects.

I start with the basics and work from there. It has been rewarding to hear students express their delight at learning fundamental programming concepts that they wished had been presented in past programming classes. And others have been excited about a whole new technical world opened up to them that they had never considered was within their grasp.

Objectives

Upon completion of this book, you will:

- know what "generalized syntax" is
- know how to use generalized syntax to document a command line expression
- know what it means to "document a graphical user interface (GUI)" and "document an applet"
- be able to recognize programming concepts like variables, methods, functions, constructors, destructors, arguments and inheritance when looking through source code
- know how to create an effective first draft of API reference documentation for a programming audience
- know what information needs to be gathered from developers
- know how to reformat code samples

The importance of various topics

Which chapters are most important for you read? Which topics are most important for you to understand in your pursuit of creating API documentation or other aspects of software documentation?

You need to be prepared to document various aspects of a software product:

- how to use the program (“Documenting a graphical user interface,” page 155 and “Documenting a program’s command line syntax,” page 161)
- how to use a Java applet in web page (“Documenting an Applet,” page 171)
- how to make use of classes or functions provided by the company (“Documenting the API of software,” page 175)

You need to:

- understand why API documentation is needed by programmers and what they will be looking for in your documentation

To do so, you need knowledge of OO concepts and syntax.

- extract key information about classes, variables, methods, functions, constructors and destructors from source code

  To do so, you need knowledge of OO concepts and syntax.

- interview developers to gather descriptions of members

  To do so, you need knowledge of OO concepts and syntax.

Book recommendations

Looking for a good reference book on programming? Having such a book nearby is not required as you go through this book. However, it often helps to hear the same concept said in different words. For that reason, you might want to have a Java reference book nearby.

Rather than highlight any particular book, let me suggest this approach. Go to the bookstore and grab several Java or C++ tutorial or reference books that appear to be what you may need. (And check my web site, in case I have completed a book on introductory Java programming.)

Look in the index of each for the `for` expression. If you cannot find it in a book, reject that book. If the author did not provide that simple of a reference, you can't count on the author to be of help elsewhere.

Read the descriptions of the `for` expression in each book. Select the book whose style of explanation best fits your learning style.

Oh, and beware: Javascript is not Java nor is it related to Java, even though it has j-a-v-a in its name. (It is a programming language for and unique to web browsers.) So do not purchase a Javascript book when looking for Java. Also, beware of books that claim to teach Java but are actually about applets—how to find applets out there on the internet and make use of them in your web page. That is about HTML, not about Java.

2

Common software documentation tasks

Your job as a software technical writer is to provide the appropriate documentation for the software that your company sells. There are different kinds of documentation for different kinds of software and for different audiences.

This section introduces you to various kinds of software products that are sold and notes the documentation that may be fitting for such products.

In this chapter

Terminology

The user of your company’s product might be a skilled programmer or a low-skilled button-clicker. Let’s get some terms right before moving forward.

- You

  I will assume you are a technical writer (or on your way to becoming one).

- Your company

  I will assume your company sells a software product and your job is to create documentation related to it in some way.

- Your company's customer

  Rather than try to guess all of the kinds products a company might create and who might be the target audience, I will state a couple of examples to illustrate the variety.

  Your company's product might be ready for the low-tech end user to use, clicking buttons and selecting menu items. Or it might be software that the end user never sees because customers purchase your company's product to merge in some way with their own software product, a product which the end user sees.

  It is therefore important for you to completely understand who the target customer is for the product for which you are writing. It may be that the intended audience for your documentation is end users. Or the audience might be sophisticated programmers. Or it might be both, depending on what aspect of the product the customer is using.

  - End user

    This is a person who interacts with the program, such as through a graphical user interface or command line interface.

    Depending on the product, you might have the expectation that some of your users could have very little technical knowledge (for example, users of web browser software) or you might expect that the users are more skilled (for example, users of graphical software development environments or high-end 3D modeling software).

    It is good for you to consider what minimum knowledge or skill you expect the end user to have.

  - Developer

    This is a person who writes software, also called a programmer, engineer or software engineer.

    Depending on the product, you might anticipate that new programmers will be using it. On the other hand, you might expect only the brightest programmers or system administrators to even consider touching your product.

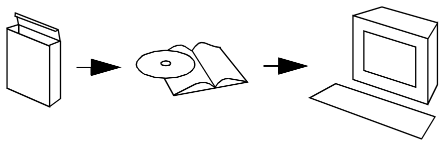
It is good for you to consider what minimum knowledge or skill you expect the developer to have.

Software products and their documentation

Software companies sell one or more the following:

- Basic computer programs
- Extendable software
- Applets

Basic computer programs



I consider this the simplest of the tech writing tasks—helping the reader understand how to *use* the software.

Consider a web browser. Does the end user need to know what programming language was used to create that web browser? No. He or she simply needs to know how to use the buttons and menu options to accomplish a task.

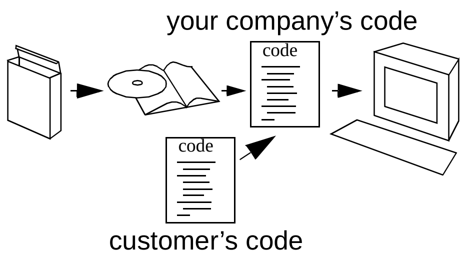
If the software is to be used straight out of the box, you explain these kinds of things to the end user:

- how to *install* the software and otherwise prepare the computer
- how to *start* the software from a command line or icon (this is called "documenting a command-line expression")
- how to *use* the program, such as "click this button" and "select this menu option" (this is called "documenting the graphical user interface"); you might even provide a tutorial that guides the user through some simple scenarios in order to become familiar with key aspects of the product
- bits of information that you didn't get in time to include them in your formal documentation at print time (which may be called "release notes" or "readme" files)
- what such-and-such means (perhaps some reference material)

Extendable software

Software can be written so that a customer’s software can communicate or interact with it. A Solitaire game on a computer is not designed to interact with other programs. It is a standalone product. But a web browser is designed to be flexible and to enable plugins. The latter is an example of extendable software.

Software that a customer can extend



Your company might create software to which its customer can add its own software.

Imagine that your company sells browser software. The end user sees and uses the browser on the computer after it has been properly installed.

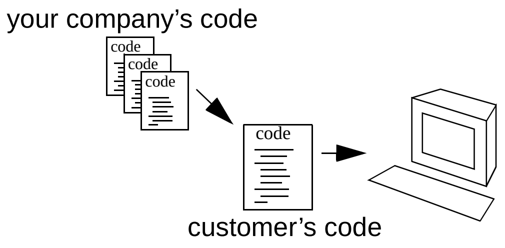
Further, imagine that other companies can create plugins for that browser. Their plugins appear to run as part of the browser. When the end user clicks on a link, he or she might be informed that a plugin needs to be downloaded. Once the plugin is downloaded and installed, something new appears in the browser or in a separate window. Visits to links like this one in the future activate that same plugin. The plugin appears to be part of the browser, working smoothly with it (or at least, that’s what the plugin company intends).

Note how your company has two kinds of customers in this example: those who simply use the browser (end users), and those who create software that interacts with your company’s product (developers), perhaps to enhance the experience for the other customers, the end users.

How does the developer-customer make its plugin work smoothly with your company’s browser software? What is required is that your company create and identify certain programming expressions that others can use to interact or communicate with your company’s software. These rules of interaction are called the "API" of the software, the application programming interface.

It is your job to understand what those rules are, what those software expressions are, and present them in an effective form to a programming audience who is interested in knowing just what they can do with your product and what are the guidelines for doing it. Such documentation is called API documentation. "Application" is another word for software or program. "Programming interface" refers to the way one program interacts with another. Such documentation might be comprised only of a reference or might also include explanation and examples of how to use portions of the API to accomplish a particular task.

Software that a customer can incorporate



Imagine your company sells graphics subroutines. They provide basic functionality such as drawing lines and curves. A video game company might prefer to buy your company's graphics subroutines for use in its own software rather than try to write such subroutines from scratch.

The programmers at the video game company (your company's customers), need to understand the rules of interaction of your company's software, how to make use of such software within their own software.

Where there are rules of interaction that need to be understood by programmers, there needs to be API documentation.

Applets

This is a Java concept. There are Java applets and Java applications.

You might think that the "-let" ending of applet means "small" as in "small application." Well, that's not true. (Indeed, some applets are huge and some applications are tiny.)

An applet is a Java program that runs in a web browser. An application is a Java program that does not require a web browser.

In short, an applet is something that can appear in a web page just like an image. It occupies a certain width and height within that page.

Just as the `<IMG>` tag is for images, so the `<APPLET>` tag is for Java applets.

Your company might sell applets that customers can include in their web pages. If your job is to document the applet, most likely you are to explain to the reader how to use the applet as-is in his or her web page. You would explain the proper use of the `<APPLET>` tag and its parameters.

Note that the tech writer probably does not need to know anything about how the applet was written—the focus is primarily on the proper use of HTML tags. In this sense, to "document an applet" typically means to "tell the reader how to use HTML tags to include the applet in a web page."

Knowing your audience

Make a decision

Who will be using your company’s product? For whom should you write your documentation? Should you assume that your audience will include those with less skill and try to help them along at every point that could be considered challenging? Should you assume that only the brave and brilliant will be using your company’s product and write succinctly and at a high level?

Part of your job is to establish just who the audience is for your documentation and write accordingly. It may be appropriate to get your software and tech writing departments to make a decision on this early on so that you are less tempted to waver and try to take care of a larger audience than your company intends.

Consider Software A. It was designed to be used by both unskilled users and skilled programmers. It is appropriate for you to consider just how much help the unskilled users need to be successful with the software. At the same time, you don’t want to bore the advanced user who does not want to read through all the hand-holding details you might provide for the unskilled user.

Consider Software B. It is to be used by advanced users. What should you do about the less-skilled user who wants to use Software B? Should you try to dumb-down the documentation just a little bit, just to "take care of everyone," or should your company decide that, no, the product really is intended for an advanced audience, and that less-skilled user is on his or her own?

Depending on the simplicity of the product and on your decisions about your company’s customers, you might choose to provide a single document whose chapters cover all related topics, or you might choose to break out certain sections as their own books.

Avoid dumbing down

If your audience is skilled programmers, be realistic about your knowledge compared to theirs. Realize that they probably know far more than you do about programming. If your inclination is to write all of your documentation as if your audience has the same questions and skill as you, that could result in terrible documentation for your advanced audience.

So, please keep in mind that, by your working through this book, you are "catching up" with programmers, learning concepts and terms appropriate for software documentation, terms which they probably know inside and out.

Is it possible to create quality documentation for an advanced audience when you don’t understand the concepts yourself? I think so! If my audience finds my documentation effective, I have succeeded, even if I do not fully understand what what I wrote about.

My practice is to take a tape recorder with me when I am gathering information from a programmer for my documentation because the programmer is likely to express many concepts I just don’t have a clue about yet. I transcribe the recording, rework the phrases into sentences that I think make sense and pass them by the programmer for comments. If other technical reviewers are available (for example, quality assurance folks or field engineers), I might get their responses to the descriptions as well. If they say my descriptions are fine, I may accept that and move on.

Now, hopefully, you will also come to understand the programming concepts or details of the software as you become more familiar with the product. There have been many times when it was weeks, even months, before I understood some concept I had presented in my documentation. With delight, I’d think, "Oh! That’s what that means!"

So, establish who your audience is and write clearly, concisely and effectively for them.

3

Generalized syntax expressions

In this chapter:

A writing convention, not programming

"Generalized syntax" is a symbolic style of writing, a kind of "short hand." Learn it and benefit from using it in many forms of communication with programmers. The remaining instruction in this book also depends on your understanding generalized syntax expressions.

Certain keyboard characters, in the context of software documentation, have special meaning to an audience that is familiar with such symbols. Those symbols help convey in just a few characters what might otherwise take many words and examples.

What I present in this chapter is a composite of styles I have seen. You might discover that your company enforces a variation on what you will learn here. So, learn the basics from this chapter and then check with your company for its required style.

Note—While the concepts of generalized syntax are not programming, they probably will feel like programming as you get used to them and learn the finer points of how to use generalized syntax expressions effectively.

An example of generalized syntax

You have seen the "Hello! My name is" tag that people affix to their clothing at gatherings. Certainly, people do not need instructions on how to fill in the tag, but let’s pretend that you were asked to document the format of the hello tag. If you know that the word after "is" should be a first name, and more specifically, the person’s choice of first name, the generalized syntax form would be this:

```
Hello! My name is <firstName>
```

The angled brackets around firstName are generalized syntax symbols. They help the reader understand that the term within the angled brackets represents the concept of what the reader is to provide in that location in the phrase.

There are many instances in software documentation when we need to tell the reader what to type to accomplish a particular task. Often, variations are allowed in what is typed to achieve different results. We have rules in mind for what must be typed by the reader and what is optional, what can be repeated and what choices are available. The symbols of generalized syntax help compactly convey such details.

Beware! Many of the symbols used in the generalized syntax style of writing are also used in Java and C++ programming and have entirely different meaning in such contexts. As you read through this section, keep in mind that you are learning about symbols that can be used in documentation—you are *not* learning about programming. (And to underscore this point, I will teach you the symbols by giving documentation examples that have nothing to do with programming.)

Here are examples of where you might see generalize syntax in use:

- in documentation that explains how to start a program from a command prompt—see "Documenting a program’s command line syntax" on page 161
- in meetings with or emails to writers or programmers who are familiar with generalized syntax
- in documentation about what text is allowed in a GUI text box
- in documentation that explains the format for the contents of a data file
- throughout this book

The symbols of generalized syntax

The following table summarizes the symbols and meanings of the symbols of generalized syntax.

Symbol	**Meaning**
`< >`	“Replace this placeholder with your own value,” page 33
`[ ]`	“It is up to you whether you supply the item,” page 37
`|`	“Choose one of these,” page 40
`...`	“The item can be repeated,” page 43
`[ ...]`	“The optional item can be repeated,” page 44

Replace this placeholder with your own value

Syntax

`<nameOfConcept>` `or` *nameOfConcept*

There are two ways you can represent a placeholder in your text: either put the name of the placeholder within angled brackets, or italicize the name of the placeholder and omit the angled brackets. (Should the italicized version be courier-italics? That is up to you. I like how the non-courier version stands out.)

The angled bracket form is necessary when you cannot be assured that your reader can view italics, when it is possible that the reader can only see plain text. Whichever style you use, use it in all cases throughout your documentation. Do not mix the two.

If the concept name is made of two or more words, start the first word with a lowercase letter and cram the remaining words together after each is started with a capital letter. (This style of naming is common in programming, as well.)

Meaning

You use a placeholder in a larger generalized syntax expression to communicate to the reader that that portion of the expression is to be replaced with a value of the reader's choosing. The value must be valid for the concept that is identified by the placeholder name. An example of a placeholder is <firstName> on page 32.

For example, if the placeholder is <firstName>, a valid value is John. If the placeholder is <number>, a valid value is 12. If the placeholder is <color>, a valid value is red.

The placeholder name should be crafted to categorize what the reader is to provide in its place. An adjective in the name can help clarify or narrow the category. For example, <petName> and <businessName> are clearer than just <name>.

A placeholder name should not include symbols or punctuation in an attempt to tell the reader such details as restrictions on values or formatting rules. For example, if you want the reader to provide a number in place of the placeholder and that number must between 1 and 10, a proper placeholder is <number>, not <1-10>. When you recall that the name of a placeholder should be the name of a category or concept, it is clear that "1-10" is not a name. The expression <number> within a larger generalized syntax expression conveys that the reader is to provide a number in that location. The fact that the number must be between 1 and 10 is something you would express in normal sentences after the generalized syntax expression.

Likewise, these would be *invalid* placeholders.

```
<Mary>
<telephone#>
<Mr.>
<firstInitial.>
<0thru5>
```

Example 1

```
Hello! My name is <firstName>
```

Observations and interpretation

This generalized syntax expression contains one placeholder. The remainder of the expression is literal text (that is, the reader should type it as-is with no changes.)

The concept of the placeholder is "first name." Therefore, after typing `Hello! My name is`, the reader is to type a valid first name, such as `John`. A complete, valid interpretation is therefore this:

```
Hello! My name is John
```

Should the reader type a period at the end? No, because the original generalized syntax expression does not include a period after <firstName>.

Example 2

```
Mary had a little <noun>.
```

and

`Mary had a little` *noun*`.`

Observations and interpretation

This illustrates two ways of saying the same thing. As noted in the Syntax description, the reason to use one format over the other depends what the reader will be able to see. (Note that I use both styles in this book as part of increasing your skill in reading both styles. You should stick with one style in your own documentation.)

A valid interpretation

```
Mary had a little tree.
```

Reverse engineering

So far, the examples have been of completed generalized syntax expressions, ready for interpretation. Let’s start with the end results and work backwards.

Here are several interpretations, and you need to come up with the generalized syntax that is valid for all of the interpretations:

```
For my vacation, I went to France.
For my vacation, I went to Italy.
For my vacation, I went to the Bahamas.
```

A simple approach is to look for what is unchanging from line to line. That text should appear in your generalized syntax with no angled brackets:

```
For my vacation, I went to
```

Next, come up with a concept name for the part that changes from one example to the next. They are all what—Countries? Destinations? Tourist spots? As the technical writer, you choose the name that you feel best categorizes or reflects the variations. For example, I will choose "country," thus ruling out places like California and Disneyland.

The completed generalized syntax expression is therefore this:

```
For my vacation, I went to <country>.
  OR
For my vacation, I went to country.
```

Here are three sample statements which you wish to turn into a single generalized syntax expression:

```
For my vacation, I went to France.
For my vacation, I went to Italy.
For my vacation, I went to the Bahamas.
```

EXERCISES—PLACEHOLDERS

Answers being on page 297.

Note—Most of the examples and exercises in this chapter are odd. My purpose is to get you to ignore your sense of "what things should look like" (something that I have observed can get in the way for some folks) and focus on understanding the pure meaning of each of the syntax symbols in generalized syntax.

For each of the following, provide two valid interpretations.

Exercise 1:

```
5 <letterOfAlphabet> chicken
```

Exercise 2:

```
(<bodyPart> <carPart> breakfastItem)
```

Exercise 3:

> *lastName firstName*

It is up to you whether you supply the item

Syntax

```
[ validExpr ]
```

validExpr can be one or more items, where an item is plain text, a placeholder, or any other generalized syntax expression.

Meaning

The entire expression within the square brackets is optional. It is the reader's choice whether to type the text within the square brackets.

Example 1

```
I am [ very ] hungry.
```

This syntax includes one optional item. The reader can type either of the following:

```
I am hungry.
I am very hungry.
```

Example 2

```
I am [ often ] [ very ] hungry.
```

This syntax includes two independent optional items. The reader can type any of the following:

```
I am hungry.
I am often hungry.
I am very hungry.
I am often very hungry.
```

Example 3

```
I am [ often very ] hungry.
```

This syntax includes one optional item that contains two words. The reader either types both of the words or neither of the words. The reader therefore can type either of the following:

```
I am hungry.
I am often very hungry.
```

Example 4

```
Mary had a [ <adjective> ] lamb.
  OR
Mary had a [ adjective ] lamb.
```

This syntax includes one optional item. That item is a placeholder, shown in both legal styles.

Some valid interpretations:

```
Mary had a rectangular lamb.
Mary had a oblique lamb.
Mary had a lamb.
```

The reader must type `Mary had a` , then the reader has the option of whether to type an adjective, and finally, the reader must type `lamb`.

The adjective "oblique" introduces a problem, a complication with using the limited set of symbols of generalized syntax. You would want the reader to type "an" before "oblique." Should this be the corrected syntax?

```
Mary had a[n] [ adjective ] lamb.
```

Unfortunately, since the reader can choose which optional expressions to observe and which to ignore, a valid interpretation could be this:

```
Mary had an lamb.
```

One of the main goals of generalized syntax is to provide as accurate of a meaning as possible *in as short of an expression as possible*. When that shorter expression leaves some ambiguity, simply supplement the single generalized syntax line with normal sentences, examples, definitions, and so on.

Considering this objective, what would be an effective way of helping the reader understand when to use "a" or "an" in the prior example? Here is sample documentation:

Sample Documentation

```
Mary had a [ adjective ] lamb.
```

Note that "a" should be replaced with "an" if an adjective beginning with a vowel is selected.

Examples:

```
Mary had a rectangular lamb.
Mary had an oblique lamb.
Mary had a lamb.
```

EXERCISES—OPTIONAL PART

1. "Reverse engineer" to provide a generalized syntax expression for these address examples. Provide additional description or examples as needed to help the reader.

   ```
   124 Main St.
   23 Schrader Ave. #14
   14882 Candlewick Blossom #102
   750 Hampton Plaza
   ```

   These are all addresses. Some have an apartment number, some do not.

   Hint: Since there are no common characters from line to line, there will be nothing literal in the solution—all items will be placeholders.

2. Interpret the following syntax:

   ```
   5 <letterOfAlphabet> [ chicken ]
   ```

3. Interpret the following syntax:

   ```
   5 [ <letterOfAlphabet> [ chicken ]]
   ```

PAGES OMITTED FROM THIS SAMPLE DOCUMENT

PART 2
Programming concepts

These chapters present the fundamental concepts of programming that you should understand if you are to document C++ or Java.

Chapters in this section:

- “Background,” page 51

  This chapter presents the concepts of source code and compiling and shows what source code comments look like.

- “Datatypes and data structures,” page 59

  Programs create and manipulate information. Datatypes and data structures are the means of storing such information. You need to be familiar with the basic datatypes and data structures provided with Java and C++.

- “Object-oriented programming,” page 79

  Object-oriented (OO) programming is, in some ways, a philosophy, an approach when writing software. You must know certain concepts of OO programming to know what to document and why to document it.

- “Syntax,” page 93

  This chapter presents the Java and C++ ways of expressing the OO concepts of the prior chapter. This chapter teaches what the code expressions are and how to find them in code by looking for certain patterns. You should know how to find each of these items when looking through source code.

- “Modifiers,” page 129

  Modifiers are code expressions which modify the definition or behavior of other code expressions. Modifiers are mentioned through the chapter on Syntax. This chapter explains the modifiers.

- “Interfaces,” page 139

  This chapter elaborates on the concept of a Java interface.

- “Package topics,” page 145

  This chapter explains packages and package statements, import statements, and the CLASSPATH system variable.

4

Background

In this chapter:

From source code to executable

Source code

A programmer creates a text file (human-readable) that has programming expressions. These expressions tell the computer what information is to be stored and retrieved and modified, and in what sequence these events should happen. The text file that contains the programming expressions is called the source file.

For example, these lines from a Java program store the number 5 in a variable named `numFingers` and print the value of `numFingers`:

```
int numFingers = 5;
System.out.println(numFingers);
```

In Java, the name of a file containing source code ends with `.java`, as in `Card.java`. In C++, a source code file name ends with `.cpp` or `.h`.

The programmer uses expressions that conform to a specific programming language, such as Java or C++. Each programming language has its own way of expressing how to store information and how to do common tasks, such as repeat an event and decide whether or not a certain event should occur. (If you have not yet learned a programming language, you will likely find that it is hardest to learn the first language you select. It is far easier to learn the same concepts in another programming language since you only need to how to express those concepts in that particular language.)

Compiling

The programmer then uses a compiler to compile the source code. The compiler is software that interprets human-readable source code and generates expressions intended for the computer (and therefore not readable by most humans).

Bytecode, executables and library files

In Java, the compiled result is called bytecode and is in a new file whose name ends with `.class`. In C++, the compiled result is typically either an executable file (a program) or a library file (a set of routines available for other programs to use) and is in a file whose name end in one of many ways, depending on the platform and the purpose of the source code (`.exe`, `.dll`, `.com`, `.so`, etc.).

Machine-dependent or not?

A C++ program is written with a particular platform in mind. (A platform is a particular combination of operating system and machine, such as Windows 2000 on an IBM PC.) When the C++ program is compiled, the executable only runs on that platform. If the programmer wants to see the same results on a different platform, he or she must use different programming expressions and compile for that platform.

In contrast, most Java source code can be written without considering what platform it might eventually be run on. Compiling a Java program creates bytecode. The information in the bytecode is specific enough to say what the program will do, but not specific enough to work on any platform. In this sense, it is platform-independent—it can be run on any platform. But something extra is needed for it to run: a Virtual Machine.

Virtual Machine

Virtual Machine (VM) is the Java name for software whose job it is to read the generic bytecode and translate it so it runs on a selected platform. This is needed because different platforms have different ways of storing data, different ways of displaying information on the monitor, etc. (Indeed, that is why the C++ programs are different for each platform.)

A VM has been created for all the common platforms. It has even been made part of the popular web browsers. Have you visited a web site and seen a message about "loading Java"? It appears in response to your viewing a web page that needs to run bytecode. The bytecode is downloaded from another location, and the VM on your computer is "loaded" or started up so that it can interpret the bytecode and make the Java program run.

Definitions

These terms have been defined in the Glossary for your future reference: Source code, Compiling, Executable, Bytecode, Operating system, Platform, Virtual Machine.

Common elements of programs

Comments

A programmer can add comments to the source code. Comments help explain the purpose of anything from a single line of code to the overall program. Such comments can be of help to the programmer or to whoever may look at the code in the future, including the tech writer.

The compiler knows what comments look like and ignores them when compiling so that they are not part of the compiled results.

Various styles of comments are allowed in C++ and Java.

Style 1: double slash

Examples:

```
// only the stuff to the end of the line is a comment
int x = 5; // store 5 in the variable x
```

When the compiler encounters the double slash, it knows that the double slash and everything to the right of it is a comment and can be ignored. Note that the comment ends at the right end of the line. In the example above, the compiler only compiles this much:

```
int x = 5;
```

If you decided the first comment expression was too long for one line, perhaps you'd split it onto a second line, like this:

```
// only the stuff to the end of the line
is a comment
```

What needs to be done to ensure this is still a comment? It needs its own pair of slashes before "is a comment". Otherwise, the compiler would gag, since it wouldn't know what "is a comment" means.

Style 2: slash-asterisk, asterisk-slash

Example:

```
/* whatever is between the pair is treated by
   the compiler as a comment */
```

When the compiler encounters the opening slash-asterisk, it looks for the first asterisk-slash after that. Everything between and including the pair is ignored by the compiler. The style 2 comment can span several lines (even the whole program, if the programmer so chooses).

What parts of this example makes it a style 2 comment?

```
/***** wow *****/
```

Only the first two and last two characters. All of the other asterisks are just part of the comment. They help make the comment look "pretty".

Many programmers use style 2 in the following way to create a marquee-like introduction to a key part of their program:

```
/**********
 *
 * www.phoons.com
 *
 **********/
```

Again, only the first two and last two characters make it a style 2 comment. All the rest is fluff.

Another common use of style 2 is to "hide" lines of programming code temporarily. Imagine a program has these lines of code:

```
  int x = 5;
  int y = 5;
```

Now imagine that the programmer wishes to try some different programming expressions in place of these two, yet does not wish to lose the original lines. One solution is to use the style 2 comment symbols around the original lines, and provide the new lines after the comment. The result is that the original lines are still visible for the programmer to compare with the new, yet they are hidden from the compiler.

```
/*
  int x = 5;
  int y = 5;
*/
  int x = 25;
  int y = 25;
```

The example below will not compile. Can you figure out why? The programmer originally had two lines (the comment about 50 and the line that stores 50 in z). He then decided to hide both lines by adding additional style 2 comment tags around both lines.

```
/*
  /* 50 is the minimum temperature */
  int z = 50;
*/
```

Here is a question to help you determine the cause of the prior bug: Which closing tag does the compiler encounter first, when trying to find a match for the first opening tag?

Style 3: slash-asterisk-asterisk, asterisk-slash

Example:

```
/** Assigns the value of the input variable to x. */
void setX(int input) {
  x = input;
}
```

You can see that this is simply a style 2 comment with an extra asterisk. As you learned about style 2 comments, that extra asterisk is treated as part of the comment and is ignored by the compiler. So why have a style 3 comment?

There are programs such as `javadoc` and `doxygen` which scan the source code, looking for:

- class members
- style 3 comments above such members

These programs then construct documentation (in formats such as HTML and FrameMaker) that lists the members and the comments associated with those members. Such documentation is called API documentation. As you might guess, that means that the quality and readability of such documentation depends on the quality and readability of the style 3 comments in the source code

As will be discussed in more detail later in "Documenting the API of software" on page 175, a tech writer can use a program like javadoc to gather information from source code to use as the basis for API documentation or as a cross-check of the tech writer’s work.

For more information on javadoc, see

```
http://java.sun.com/products/jdk/javadoc/
```

Expressions

Semicolons and spaces

As you look through source code, you will see many semicolons (`;`). The compiler uses them to determine where one programming expression ends and the next begins.

If a programmer wanted, he or she could type an entire program on one really, really long line, because the compiler is not interested in appearance but rather in isolating expressions. Thus, this first sample:

```
/***** wow *****/ int x = 5; int y = 5;
```

is no different to the compiler than this sample:

```
/*****
 wow
 *****/
```

```
int x
  = 5;
int y = 5;
```

A programmer uses blank lines, spaces, tabs and carriage returns primarily for human readability.

Braces

You will also see many braces. They are always paired. And they can be nested. Typically, the expressions within a pair of braces are indented the same amount.

Note the two pairs of braces in the following example:

```
while (x > 5) {
  x = x - 1;
  y = y + 2;
  for (int i = 0; i < 4; i++) {
    System.out.println(i);
  }
}
```

5

Datatypes and data structures

The basis of programming is storing, modifying and retrieving information. Each programming language, such as Java, C++ FORTRAN and BASIC, provides a way for the programmer to store information. The form in which each bit of information is stored is called its *datatype* or, in more complex cases, its *data structure*. This chapter presents these concepts.

◎ Where does this chapter fit in the big scheme of things? Why read it at all? See "The importance of various topics" on page 21.

In this chapter:

Datatypes

The basis of programming is storing, modifying and retrieving information. Each programming language, such as Java, C++ FORTRAN and BASIC, provides a way for the programmer to store information.

The form in which each bit of information is stored is called its *datatype* or, in more complex cases, its data structure.

We will discuss these means of storing information:

- primitive datatypes
- data structures:
    - arrays
    - structs
    - classes

Primitive datatypes

The simplest or most primitive types of information a programmer needs to store are whole numbers, numbers with decimal points and text. The programming languages provide "primitive datatypes" for these basic types of information.

The main objective here is for you to become familiar with the primitive datatypes listed in the tables below. Such primitives are used extensively in the remaining chapters of this book to illustrate more advanced programming topics. Thus, the more recognizable these items are as primitive datatypes, the less of a hindrance they will present to you as you move to other programming concepts.

In the end, precise understanding of the datatypes is not essential, for you will simply copy/paste datatypes information from source code into your documentation in a form that is more easily digestible by your reader, as presented in the chapter "Documenting the API of software" on page 175.

Primitive datatypes in C++

`byte`	whole number (`char` can be a single character)
`short`	
`char`	
`int`	
`long`	
`float`	decimal number (not as accurate as `double`)
`double`	decimal number (most accurate)
`bool or boolean`	`true` or `false`
`char *`	often used for text

The number of bytes occupied by each datatype depends on the operating system and hardware.

Primitive datatypes in Java

The primitive datatypes store the basics: numbers and characters.

datatype	what it stores	memory it occupies
`byte`	whole number (`+/- 128`)	1 byte
`short`	whole number (`+/- 32768`)	2 bytes
`char`	whole number (`+/- 32768`) or single character (`'t'`)	2 bytes
`int`	whole number (`+/- 2,000,000,000`)	4 bytes
`long`	whole number (`+/- 9` with 18 zeros)	8 bytes
`float`	decimal number (not as accurate as `double`)	4 bytes
`double`	decimal number (most accurate)	8 bytes
`boolean`	`true` or `false`	1 byte

Note—The datatype for storing text is `String`. (It is a class and therefore not a primitive datatype, but it is worth mentioning now because of the common need to store text.)

About the primitive datatypes

Several datatypes are provided for storing a whole number. The ones listed here are listed in order of the size of whole number each can store.

A `byte`, for example, occupies a single byte of memory in the computer. The biggest number it can store is 128. (There are more technical details and variations, but those don't matter for our discussion. Just note for now that it's a smaller number than can be stored in the other datatypes for whole numbers. And feel free to get the fine details from a programming book.)

A `short` is 2 bytes in size. It occupies twice the memory of the `byte` datatype, yet it is capable of storing a number as large as 32,000. (Here's where you either recall your training in the binary or you allow it to be one of those magical mysteries about a computer.)

In C++, the amount of memory for some datatypes depends on the operating system. With one, an `int` might occupy 8 bytes. On another, just 4. On another, 16. (A 4-byte `int` can store numbers as large as 2 billion. You can imagine that a 16-byte `int` can store a whopping huge number.)

In Java, an `int` is always 4 bytes, regardless of operating system.

The `char` datatype stores numbers of a certain size. As it turns out, single letters are represented as numbers in a computer. As the datatype name suggests, a `char` can store a character (not a word—just a single character). In the programming world, char is pronounced "char," not "care."

The `float` and `double` datatypes are for numbers that have a decimal point. A `float` is less accurate to the right of the decimal point than a `double`.

The `boolean` datatype can store only one of two things: the word `true` or the word `false`. (Neither word has quotes.)

C++ specifics

C++ has many possible variations of the primitive datatypes through use of additional keywords like `unsigned`. Thus, unlike Java, datatypes can be more than one term. Further, C++ datatypes can include one or more asterisk, meaning pointer.

A skill you should develop with C++ is recognizing the datatype portion of an expression.

Text is stored with `char *`. Text typically appears in quotes within a program.

Why is "char *" the dataype for text? To explain requires some background. The combination of a datatype and an asterisk is called a pointer and represents the computer memory address of the first byte of memory occupied by a chunk of information (see "Variables and memory" on page 66).

It is pronounced with the rhyming sounds "char star" in the programming world (and not "care star" or "car star"). It is also referred to as a "pointer to `char`" or "`char` pointer."

Letters of a word or phrase are typically stored as individual characters, `char`s, in adjacent bytes of memory. The C++ compiler treats "char *" as the string of characters beginning at the address stored in the "char *" variable.

Pointers can be used on any datatype. Here are a pointer to int and a pointer to Card (a playing card from a deck):

```
int *
Card *
```

A pointer to Card would the beginning of the region of memory that is occupied with information about a particular card.

Thc C++ language comes in various flavors. Several companies and organizations have written their own variations. On top of that, there can be differences based on the operating system (for example, Mac versus Windows versus Unix).

Most versions include additional keywords. One is `unsigned`. When this keyword is combined with various datatypes, the resulting two-word datatype means "this datatype is restricted to store only positive numbers."

For example:

```
unsigned short
unsigned int
```

Trivia: `unsigned`, when used by itself in source code, is shorthand for `unsigned int`.

And there are some additional combinations:

```
short int // same as short
long int // same as long
long double // can be larger than double, dep on lang
```

It is not important that you understand all these variations. When documenting variables, you simply copy/paste the datatypes you find in the source code into your documentation. Basically, you are just passing on the information to your reader who knows their meaning. The important angle for you is how to present the information effectively, as is described in the chapter "Documenting the API of software" on page 175.

Java specifics

Text is stored in `String` objects. Text typically appears in quotes within a program.

Why choose one datatype over another?

If a program will be storing thousands, even millions of bits of information, memory might become very, very precious to the programmer. The programmer would probably then choose the smallest-sized datatype for storing the information. For example, the number of cans in a coke machine is likely to be less than 200, so `byte` would be a better choice than `int` since it occupies 1/4 or less of the memory of an `int`.

For a smaller program, however, the programmer might not care how much memory is taken up by little bits of information here and there. The favorite datatype among programmers for whole numbers in such cases appears to be `int`.

The computer representations of decimal numbers (or "floating point" numbers) in a float or double are nothing like the datatypes for whole numbers. Size in memory does not determine how large of a number can be stored in either datatype. Rather, it determines how accurate the number is. You can count on bank software and satellite software using `doubles` to be as accurate as possible.

In Java, `int` is the default representation for a whole number and `double` is the default representation for a decimal point (or "floating point") number. That is, if you tell the computer to add `4.56` and `6.78`, it treats both as `doubles` and gives back the answer in the form of a `double`. If you insist on the values being `floats`, you add an `f` after each number: `4.56f` and `6.78f`.

Declaring a variable

A datatype and a name

Once a programmer has identified the appropriate datatype for a bit of information, he or she must choose a name by which the information is to be known within the program. For example, he or she might choose "phoneNum" for the text of a phone number of the user, and "amtChange" for the decimal amount of change in the user's pocket.

For the computer to know that `phoneNum` is text and `amtChange` is a decimal number, the programmer must formally declareeach as a variable.

To "declare a variable" is to put a choice *datatype* and *name* together, once. From then on, the compiler will know that *name* is of type *datatype*.

> *datatype name*`;`

Note—I am using generalized syntax here to represent programming concepts. Please refer to "Generalized syntax expressions" on page 31 as needed.

Here are several variable declarations:

```
int population;
float cost;
double satelliteSpeed;
boolean isUSCitizen;
```

That is, `population` is the name chosen for an `int` that will store a whole number of people, `cost` is the name of a `float` that will store the price of something, `satelliteSpeed` is the name for a decimal number, and `isUSCitizen` is the name for a variable that can contain either the value `true` or `false`.

From now on, the computer will know that the name `satelliteSpeed` refers to a place in memory that can store a `double`. Depending on the programming language, that place in memory is empty or is assigned a default value like 0, 0.0 or `false`, depending on the datatype.

Naming conventions

The C++ and Java programming languages are case-sensitive. You can name one variable `population` and another `Population` and the program will know they are different.

When it comes to inventing variable names, you can choose whatever letters and case you want. You could use `pOpULaTion`, as long as you stick with that form every time you refer to the variable.

There are various preferences out there regarding how to name variables. The most popular is to start variables with a lowercase letter and use uppercase for the first letter of adjoining words.

Declaration styles

You have seen the style of *datatype name*, as in:

```
int population;
int numTrees;
```

An alternate style is to combine same-datatype variables into a single declaration line. The datatype appears once. Names are separated by commas.

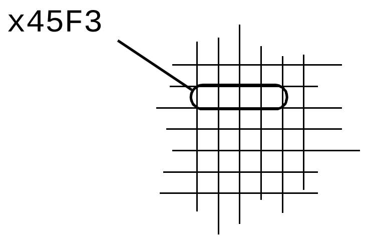
```
int population, numTrees;
```

Variables and memory

When the programmer declares `population` to be an `int`, the computer sets aside enough memory for that `int` to be stored, notes that that memory represents an `int` (and not a `double` or `boolean`) and records the starting address of that memory.

Pretend that the computer chooses memory location `x45F3` as the start of the `int` in memory, and assume that an `int` occupies 4 bytes on this computer. I have represented a portion of the bytes in memory by a grid.

name used by programmer	location in memory	datatype stored there
`population`	`x45F3`	`int`



My understanding is the computer maintains a lookup table. Whenever the programmer uses the name `population`, the computer looks in its table and finds that `population` is the nickname for the `int` stored at location `x45F3`. In effect, the programmer can consider the name of the memory location to be `population`. The computer does the job of translating that into the address `x45F3`.

Now, what does the word "variable" mean? Well, since `population` refers to a location in memory, and the information stored at a location in memory can be changed or *varied*, "variable" is a suitable concept name; `population` is a variable whose value can be changed.

EXERCISE—DATATYPE AND NAME

What *datatype* and *name* would you use for each of the following concepts?

whether exact change is required
body temperature of 98.6
street address (answer for Java, then for C++)

Assigning values to variables

A variable is declared so that information can be stored in memory. To store information, information must be assigned via variable name:

name `=` *value*`;`

DOCUMENTING JAVA AND C++

- “About code samples,” page 211
- “Documenting functions that are not members of classes,” page 212

The essential elements for a programming audience

As a programmer, I often need to look up information in API documentation. I know what kind of information I am looking for when browsing such documentation. And as a technical writer, I have a sense of what would make it easier for a programmer to find such information.

When a programmer is reading API documentation, he or she is looking for certain details about a class or class member. The focus of this chapter is on the essential elements that must be presented for each class or class member.

I also suggest styles for documenting classes and their members. They are, for the most part, suggestions. While I might show indentation or bulleted lists or certain fonts, I don’t expect that you should or will match such presentation details. The focus is on the details which should appear somehow in some fashion, and I give examples on how such details might be presented.

Should you adhere to my style? Actually, there *are* some style details that I consider essential, and I point those out along the way, helping you understand why they are essential. Other than that, however, use whatever presentation style reflects your understanding of the essential elements of API documentation.

At the end of the chapter, I review three documents written by three authors on the same topic, helping you see the positives and the negatives of each approach style in light of your knowledge of the essential elements that should appear in API documentation.

Gathering information

Reviewing the code

If you do not have access to the source code on your work computer, by all means get it. If your role is to document source code, you should be allowed access, even if it is read-only access or you have a copy of the source code.

You might find the following tasks helpful:

- On a printout of the class source code, highlight the line that identifies the class, and highlight all public members of the class.

  Typically, companies let customers know only about the public members of a class. Therefore, assuming that you will only be documenting the public members and be prepared to check this assumption with the developer.

  In some cases, the raw source code is also provided to the customer, so it doesn't matter if the members are public or not since the customer can look at the code.

  If it is Java code and you find members with no access modifier such as `public`, make note to ask the developer if any of those members should actually be public and therefore be documented. (It could be that the programmer forgot to include the access modifier.)

  If it is C++ code, perhaps there are header files that can give you an initial sense of what needs to be documented. See "About header files (C and C++ only)" on page 179.

- If you find it helpful, make note of which members are variables, methods, functions, destructors or constructors.
- If a member is marked `static`, circle `static` to distinguish this member from non-static members. If you use the styles of this chapter, you will document static members separately from non-static members.

Questions for the developer

With printouts in hand, visit your developer and ask a few questions:

- Are the members that you have highlighted what are to be documented?

  It could be that some of the public members should *not* be documented. (There can be public members which the developer believes the customer should neither use nor know about.) It could be that some of the non-public members should be documented.

- Will the customer receive the source code for this or any other class?

  If yes, the customer sees every member, whether public or private. Considering that, should you document just the public members or all members?

- Should any nested classes be documented (if it is Java code)?

My guess is that you will not document any nested classes.

- Should main() be documented (if found) for its command line syntax?

  You should not document `main`() as you would document other functions or methods. `main`() has a special purpose. It enables the compiled code to be started from a command line, and other programmers know this. So, if you find a function or method named `main`(), you should ask the programmer if command line documentation is needed for that code. (It could be that the programmer simply added a `main`() to that code for personal testing of the code and expects no documentation of it.)

Dealing with descriptions

Eventually, you will need to provide a description for the class and each of its members that you document.

If you are lucky, you can find helpful comments near the members in the source code that you can use when developing your first-draft description of the member. Such comments traditionally precede the line of code to which they refer.

As I develop my documentation, I like to use {{ }} to surround notes to myself, notes to the developer or incomplete descriptions. For example:

```
int numCtx -- {{need description}}
```

This approach helps in a couple ways:

- I can convert my FrameMaker documentation to text, use a tool like Unix’s `grep` or the Windows tool TextPad (visit www.textpad.com) to print out a list of all lines containing {{ }}, and have a nice list of what is left for me to correct or research.
- I can print the API documentation and highlight the {{ }} items for which I need answers from the developer.

About javadoc and doxygen

Programs are available which search through source code and create API reference documentation. In addition to identifying the members and their modifiers, they search for comments of a given format and include the text of such comments in the API reference documentation.

`javadoc` is available for use with Java source code. When you download the free Java 2 SDK from Sun Microsystems, you get `javadoc`, too. A program available for C++ source code is `doxygen`.

Considering that programs like `javadoc` create API reference documentation, you might have questions like the following:

- "If `javadoc` creates API reference documentation, what point is there in my continuing to learn about how to create API reference documentation? Why shouldn't I just use `javadoc` and spare myself the effort?"

  `javadoc` does not provide all of what I consider the required elements of effective API reference documentation (at least at the time I am writing this).

  Further, the output of `javadoc` is only as good as the text found in the comments. Were the comments written by programmers? For programmers with technical writers in mind? The output will certainly be best if the comments were written by those skilled in the written language and who understand the required elements of API reference documentation.

- "My company is thinking about just using `javadoc` or `doxygen` to create its API documentation. Is that a good idea?"

  It could be that your company got this idea from programmers who do not appreciate the value of technical writers in the creation of API documentation. Management might not be aware of the consequences of compromised documentation and needs your input.

  Or it could be that the project is under time or financial pressure and someone noted that the use of `javadoc` could be faster or cheaper than going down the traditional technical writer path. That might be appropriate.

  Help your management consider the alternatives and consequences. Do what you can to help your company make the right decision.

- "If I learn how to create API reference documentation as presented in this chapter, will `javadoc` be of any benefit to me?"

  Absolutely. I find `javadoc` output to be an excellent starting point in my gathering of information about a class. I might copy/paste its content into my own documentation.

- "Can `javadoc` generate output in formats other than HTML?"

Sun Microsystems designed `javadoc` so that others could extend it to generate other formats. if you search the web, you will find companies whose code generates formats such as .doc, .rtf, and .xml. Sun Microsystems provided an unsupported solution that generates MIF files in a format similar to standard `javadoc` output.

I came up with a solution based on `javadoc`, one which generates HTML or MIF files in the style that I present in this chapter and which uses the customer’s FrameMaker paragraph tags, thus bypassing deficiencies I have found with the standard `javadoc` output. Using this solution makes it possible for a company to generate nearly perfect FrameMaker documentation directly from the Java source code, significantly shortening the documentation process for medium to large documentation tasks.

Document design

Design

If you are documenting classes or interfaces, their names become the first level headings within the chapter.

If you are documenting functions that are not in classes (C or C++), the function names become the first level headings within the chapter. See "Documenting functions that are not members of classes" on page 212.

Details before summaries

We work on summary sections after we have filled in enough details to be able to create summaries.

For example, we identify the classes that are to be in the chapter before introducing them at the start of the chapter. And we complete the details about their members before we create summary tables that list those members.

Notation

I use expressions like [H1] and [H2] in this chapter to signify heading levels, not the generalized syntax concept of "optional."

```
[ChapTitle] - chapter title (only one per chapter)
  [H1] - first level heading within chapter (one
         for every class or interface)
    [H2] - next level heading within H1
      [H3] - next level heading within H2
  [H1] - next class or interface within chapter
```

Note—I am not suggesting you create documentation that matches the indentation you see in the samples in this chapter. Such indentation is just a teaching tool I use to help you see the heading levels and details.

API documentation template

The details of this template are presented in the remaining sections of this chapter.

```
[ChapTitle] title
  introToLinks
  bulletedLinksToH1s
  [H1] className|interfaceName|OuterClassName.NestedClassName
    Package: packageName|none   (Java only)
    classSyntaxLine
    description
    [H2] Constructor[s] Summary
      summaryTable
    [H2] Class Variable[s] Summary
      summaryTable
    [H2] Instance Variable Summary
      summaryTable
    [H2] Class Method/Function[s] Summary
      summaryTable
    [H2] Instance Method/Function[s] Summary
      summaryTable
    [H2] Destructor Summary  (C++ only)
      summaryTable
    [H2] Nested Class Summary  (Java only)
      summaryTable
    [H2] Constructor(s)
      [H3] shortenedHeading
        description
        syntax
        argDocumentation
    [H2] Class Variable[s]
      [H3] variableName
        syntax--description
    [H2] Instance Variable[s]
      [H3] variableName
        syntax--description
    [H2] Class Method/Function[s]
      [H3] shortenedHeading
        description
        syntax
        argDocumentation
    [H2] Instance Method/Function[s]
      [H3] shortenedHeading
        description
        syntax
        argDocumentation
    [H2] Destructor   (C++ only)
      [H3] shortenedHeading
        description
        (syntax not needed)
    [H2] Nested Class[es]  (Java only)
      [H3] OuterClassName.nestedClassName
  [H1] nextClassName
    (etc.)
```

As noted in "Issues of variables versus constants" on page 188, you might choose to present your variables and constants under a different heading than shown above.

Introducing a chapter

At some point, you need to decide which classes and interfaces should be in the same chapter. Perhaps they are related by purpose or task. Perhaps they are related by package (Java only). Ask your programmer for his or her opinion.

Required elements

The chapter needs:

- a chapter title that is representative of its contents
- a list of the classes or interfaces presented in the chapter

Style

[ChapTitle] *chapTitle*

introduction to list of [H1] links

- *link to text of first [H1]*
- *link to text of next [H1]*

Comments about the style:

- The introduction is a standard tech writing way of saying, "Here are the main headings within this chapter." You might also explain why the selected classes or interfaces are in the same chapter.
- A bulleted list is a standard tech writing way of listing the major headings of the chapter (which are, in this case, the names of the classes or interfaces).

Example

[CH] {{ask developer}}

This chapter provides details about the following classes:

- Hashtable
- Vector

Introducing a class or interface

Required elements

The reader needs to know:

- the name of the class or interface
- whether the class or interface belongs to a package and what the name of the package is (Java only)
- the name of the parent class or interface, if this class is a subclass

- a description of the class

Style

[H1] *className* | *interfaceName* | *OuterClassName.NestedClassName*

`Package:` *packageName*`|none (Java only)`

syntax (everything up to but not including "{")

description

Comments about the style:

- The heading identifies the class (or interface or nested classname) by name.
- The heading does not include the words "class" or "interface" or "nested class" (that is, the heading is "Dog", not "Class Dog" or "Dog class").
- In the unlikely case that the Java code has no `package` statement, make that clear.
- The syntax is copied directly from the source code to the syntax portion of the section. As a result, it includes key details about the class's inheritance and access modifiers, thus taking care of a few required elements in one line. The syntax is everything up to but not including the opening curly brace of the class.
- Programmers are paid to read code. If the syntax includes an expression of inheritance, the programmer will immediately understand it, even if it is currently difficult for you as you are "catching up" in programming knowledge. You do not need to provide a separate explanation of inheritance.
- One class might have one line of description while another has a full page of description. Therefore, description should be last, so that the package and syntax details are easy to find at the top of the section.

Examples

In these examples, the name of the class is clear, the package information is easy to find, inheritance is communicated in the syntax, and a description introduces the class.

Java

[H1] Dog

Package: `prog.animals`

```
    public class Dog extends Mammal
```

The Dog class represents all dogs, regardless of breed.

C++

[H1] Dog

```
    class Dog : public Mammal
```

The Dog class represents all dogs, regardless of breed.

Documenting a variable declaration

Note the structure for documenting variables:

```
[H1] className
  [H2] Class|Instance Variable[s]
    [H3] variableName
```

The class name is an [H1] heading. [H2] is for member categories within the class, such as Class Variables, Instance Methods, Constructors and so on. Thus, [H3] is for specific members within a category.

Let’s focus on documenting the variable at the [H3] level first, and then we will look at the [H2] heading.

Required elements

The reader needs to know:

- the name of the variable
- the modifiers and datatype of the variable
- the description of the variable, including its purpose, any restrictions on its use, etc.