17

Exam Documentation

CERTIFICATION OBJECTIVE

- Understand the Sun Certified Java Developer Exam Documentation Requirements

CERTIFICATION OBJECTIVE

Understand the Sun Certified Java Developer Exam Documentation Requirements

We know that you all know the benefits of thorough, accurate, and understandable documentation. There may be some of you out there who wish that documentation wasn't an integral part of a programmer's job. There may be others of you who are thrilled to write documentation, to exercise a different part of your brain, to help your fellow programmers, to capture (hey you, in the back, stop laughing!) your company's technical assets. Well, whatever your inclination, you're going to have to write good, solid documentation to support your project if you want to have any chance of passing this exam. It turns out that proper documentation plays as big a role in determining your exam score as many of the software aspects themselves.

The assessors will be expecting several pieces of documentation when you submit your exam. They are discussed briefly in the exam packet you receive from Sun; we will go into them more thoroughly in this chapter. The five areas of project documentation that we will cover are

- Developer's Documentation
- End User Documentation
- *javadoc*
- The Developer Choices File
- Comments and the Version File

Developer's Documentation

This area of the project's documentation is the most open ended. Your assessor is most interested in the final results of your project; these optional documents represent the design work that you did as you were working on the project. Documentation that you might consider providing in this section includes UML diagrams, schema documentation, algorithm documentation, flow diagrams, prototype plans, and test results. Given that the rest of the standalone documentation is to be submitted via ASCII text files or HTML, we recommend the same here.

End User Documentation

Your assessor is going to wear at least two hats when reviewing your project. (This makes her appear taller than she really is.) Initially, she will review your project from the standpoint of an end user. Once the end user review is complete, she will put on her 'techie' hat and dive into your code and technical documentation. But (and this is a big but), if she can't get through the end user portion easily and with no problems, she probably has no choice but to fail the project. It won't matter how unbelievably fabulous your code is, she'll never see it if the end user experience is challenging.

The actual end user documentation should be pretty easy; all it has to do is describe how to install, launch, and run your project. You will probably be told exactly how the application must be installed and launched, and from the end user's perspective, those tasks will have to be incredibly easy and relatively option free, so there won't be much to document. The key will be to document how to use the programs once they have been launched. When documenting the GUIs, the most important concepts to remember are

- Keep it simple.
- Keep it concise.

The GUIs themselves, if designed properly, should be very easy to use, so there is no need to go on and on.

The end user documentation can take several forms. The install and launch documentation must be provided in either an ASCII text file or as HTML. Make sure to follow the naming conventions described in your instructions! The GUI documentation can be added to either of these files, or it can be provided as online help.

javadoc and You

One of Java's more wonderful features is *javadoc*. While we're sure that all of you are well versed in the use of *javadoc*, and use it religiously, we are bound to review it here on the off chance that this bit of Java technology has somehow eluded you.

An Overview of *javadoc*

When you crank up your browser to look at the Java API documentation (let's say you've forgotten what arguments the `setInitialContextFactory`

`Builder()` method takes), you are really looking at the output of the *javadoc* utility. Most likely, that online documentation was created by the guy who actually wrote that method for that class (in this case the NamingManager class). *javadoc* is a utility for programmers to use to help other programmers use their programs. (We'll get off our soapbox in a minute.)

Every programmer should use *javadoc*. Even if you're a one-man shop, someday you'll want to refresh your memory on how a certain method works, and the very *javadoc* that you wrote months earlier will be right there to help you out. If you work with other programmers, then *javadoc* is truly a miracle. When you add *javadoc* comments to your code as you are creating it, you have an instant answer for anyone who wants to bug you about how your code works. (If the cute programmer in the cubicle next to you wants help, you can always provide additional assistance.) Likewise, if you're trying to update a class that was written by somebody else, you'll be grateful for their *javadoc* documentation, especially if for some reason that programmer is no longer around.

At a high level, *javadoc* comments are nothing more than specially formatted comments that you add in certain, very specific places in your code. When you run the *javadoc* utility on your Java files, it takes those comments, and the appropriate adjacent code, and creates HTML-based API documentation, just like you see on your browser.

If you've never used *javadoc* (gasp!), we recommend trying some experiments once you've read through more of this chapter. It's very useful to write a little code, produce some *javadoc*, and compare the two. With a little practice your *javadoc* comments will look just like those created by those 'think-tank' boys at Sun. Earlier, we promised to get off our soapbox; consider us officially off.

A Summary of the Project's *javadoc* Requirements

To pass the developer's exam, your code must include *javadoc* comments. Once your code is properly commented, you must then run the *javadoc* utility against it and include the resulting *javadoc* files in the *docs* directory for your project. Specifically, your *javadoc* comments might document some of the classes and interfaces you are submitting, including class, interface, constructor, method, constant, and exception comments. Your instructions will specify which elements you must document.

A Brief Tutorial on the Use of *javadoc*

It has often been said that if you know 20 percent of a certain technology you can accomplish 80 percent of everything that you ever have to do with it. That said,

we're going to describe for you what is, in our humble opinion, the most crucial 20 percent of the commands provided by the *javadoc* utility. If you want to know more about *javadoc* we recommend starting with these two links:

http://java.sun.com/j2se/1.4/docs/tooldocs/solaris/javadoc.html, and
http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

The Structure of a Comment As you will soon see, a single comment can grow to quite a large size. Comments can contain a wide variety of elements, but there are some restrictions to the order in which you can place these elements. To begin, the first line must start with /** (the / must be in column 1), all of the rows that contain descriptive content start with an * in column 2, and the closing delimiter is */ with the * in column 2. Finally, remember that the member declaration follows immediately after the *javadoc* comment. This format will hold true for any multiline *javadoc* comment used in documenting classes, interfaces, constructors, methods, instance variables, or exceptions; for example,

```
/**
 * the descriptive section
 * of a multiline javadoc comment
 */
public class Test {
```

A comment can contain two main sections: the description section followed by the tag section. Both sections are optional. When used, the descriptive section can contain any free form text following the column 2 *, and can span multiple lines.

The tag section of the comment begins with the first occurrence of a '@' that starts a new line of the comment (ignoring the leading *). There are two types of tags: *standalone* and *inline*. A standalone tag has the general form @tag. An inline tag has the general form { @tag }. Inline tags can be included in the descriptive section of the comment, but once a standalone tag has been encountered in a comment, no more descriptive text can be used in that comment; for example,

```
/**
 * the descriptive section
 * we're still in the descriptive section
 * {@link doStuff doStuff} and
 * after this line the tag section will begin:
 * @author  Joe Beets  (the leading @ marked the beginning
 * of the tag section
```

```
* @version  2.1
*/
```

Launching *javadoc*, and Exciting *javadoc* Capabilities

We're not forgetting our orientation toward 80/20, and at the same time we want to let you know about some of *javadoc*'s other capabilities. Think of this as a high-level briefing.

- **Doclets** *javadoc*'s output format is determined by a 'doclet'. The default, standard doclet is built-in to *javadoc*, and produces the HTML API documentation normally associated with *javadoc*. If you want to create custom output you can subclass the standard doclet, or you can write your own doclet. For the adventurous, you can create XML or RTF; we know one guy who used *javadoc* to capture all his favorite beef jerky recipes. A good placed to start your doclet odyssey is at:

  http://java.sun.com/j2se/1.4/toolodocs/javadoc/overview.html

- **Command-line Cornucopia** Let's look at a few simple examples of calling *javadoc*:

  To run *javadoc* against all the java files in the current directory,

  ```
  % javadoc *.java (we tried to start with an easy one.)
  ```

  To run *javadoc* on a package called com.testpkg, first move to the parent directory of the fully qualified package (in other words, the directory containing the package), then

  ```
  % javadoc –d /home/html-dest com.testpkg
  ```

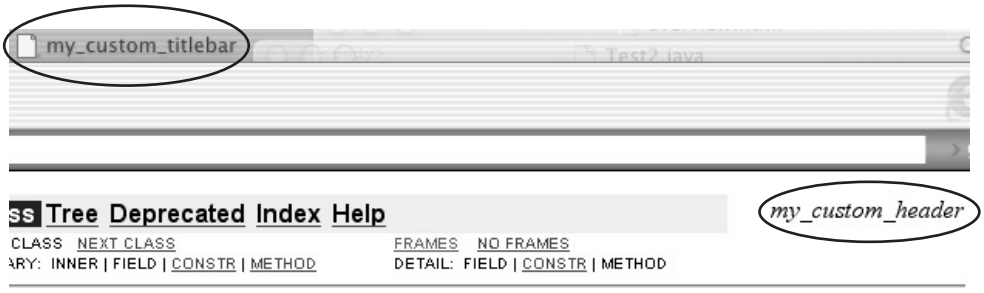
  In this case we used the –d flag to indicate the destination directory for the HTML output. So the command line reads, "Run *javadoc*, put the output in a directory called home/html-dest, and run the utility against all of the java files in the com.testpkg package."

- **Other Capabilities** *javadoc* has a wide range of command line options, in fact, a huge range of command-line options…so many that there is a facility that allows you to store your command-line options in a file. Let's cover some of options you might find useful for your project:

  - **`-windowtitle`** Allows you to specify the description that appears in the title bar of your browser window. See Figure 17-1.

FIGURE 17-1

Example of a custom title bar and header



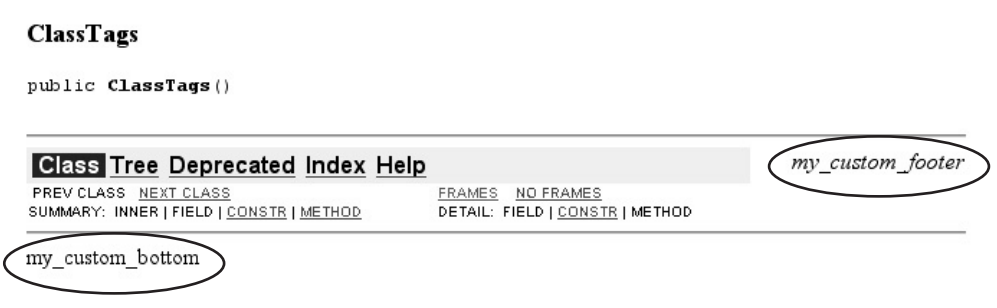
- **`-header`** Allows you to specify a description that appears in the top right of your class documentation. See Figure 17-1.
- **`-footer`** Allows you to specify a description that appears in the lower right 'footer' area of your class documentation. See Figure 17-2.
- **`-bottom`** Allows you to specify a description that appears in the bottom of your class documentation. See Figure 17-2.

The following collection of command-line arguments allow you to specify which classes and members are documented, based on their access modifiers:

- **`-public`** Documents only public classes and members.
- **`-protected`** This is the option if you don't specify a command-line argument. It documents only protected and public classes and members.

FIGURE 17-2

Example of a custom footer and bottom



- **`-package`** Documents package level (default), protected, and public classes and members.
- **`-private`** Documents *all* classes and members. (`private means "everything," including things marked private.`)

Here are some more potentially useful command line arguments:

- **`-help`** Displays online help—a good way to access all of these options.
- **`-source 1.4`** Enables *javadoc* to handle assertions if you have used them in your code. Use it for documenting code that you've *compiled* using the -source 1.4 flag.

The World's Shortest Review of HTML Tags

Inside your *javadoc* comments you can format your text using standard HTML tags. The following (exhaustive) list of tags should be enough for you to properly document your project.

- `<a href=> </a>` The anchor tag will allow you to link your *javadoc* to a URL, for example, Go to Wickedly Smart
- `<code> </code>` This tag will tell the *javadoc* utility to use code style font (probably courier) for the enclosed content, perfect for indicating code snippets in your comments.
- `<pre> </pre>` This tag will tell the *javadoc* utility to maintain the formatting of the enclosed content. This is very useful if you want to include a multiline code snippet in your *javadoc* and maintain the formatting (indenting, spacing, etc.).

The following code snippet was run through the *javadoc* utility, and Figure 17-3 shows a portion of the API style documentation that was generated. Notice that the *javadoc* utility ignored the formatting of the paragraph documentation, but preserved the formatting of the code snippet inside of the <pre> tag. Also notice how the <a href> tag was formatted to produce a live link to a website.

```
/**
 *  An example of HTML tags in a javadoc comment.
```

```
 *
 *   The <code>Byte</code> class wraps a primitive type
 * <code>byte</code> in an object.  An object of type
 * <code>Byte</code> contains a single field whose type
 * is <code>byte</code>.
 *
 * <pre>
 * int doStuff() {
 *     if (x < y) {
 *         x = 42;
 *     }
 * }</pre>
 *
 * @see <a href="http://wickedlysmart.com">Go to Wickedly Smart</a>
 */
public class Tags { }
```

FIGURE 17-3 Common HTML tags enhancing *javadoc* API output

Class Tags

```
java.lang.Object
  |
  +--Tags
```

---

public class **Tags**
extends java.lang.Object

An example of HTML tags in a Javadoc comment. The `Byte` class wraps a primitive type `byte` in an object. An object of type `Byte` contains a single field whose type is `byte`.

```
int doStuff() {
    if (x < y) {
        x = 42;
    }
}
```

See Also:

Go to Wickedly Smart

Useful *javadoc* Tags for Classes and Interfaces

Here are some useful *javadoc* tags for classes and interfaces:

- `@author` You can provide from zero to many author tags in your class comments. Although, given the nature of the exam, we'd advise zero or one. There are no formatting rules for the content after these tags. By default, author information is not included in the final API documentation; it will only be seen by people reading your source code. If you want to include the author information in your final *javadoc* output, you must run *javadoc* with the `-author flag.`
- `@version` This tag allows you to tie into Source Code Control Systems, which will automatically provide accurate versioning and date updates. Given that this is a one-person project, we recommend that if you use this tag, you insert your own manual version and date information. By default, version information is not included in the final API documentation; it will only be seen by people reading your source code. If you want to include the version information in your final *javadoc* output, you must run *javadoc* with the `-version` flag.

Useful Tags for All *javadoc* Comments

Here are some useful tags for all *javadoc* comments:

- `@see` This tag allows you to add a "See Also" entry to your *javadoc*. These entries are extremely flexible; you saw one in action in Figure 17-3, providing an intro to a URL link. `@see` can also be used to preface character strings (for instance referring to a reference book), or it can be used to preface other members in the same or other classes. Figure 17-4 shows the `@see` tag used in several different ways. There are many more wonderful possibilities that the `@see` tag offers, but we're sticking to our 80/20 guns.
- `@link` This tag is similar to `@see;` however, it creates an inline link with a label. These inline links allow online users of your API documentation to navigate quickly through your content using the hypertext links you have created with `@link.`

The following code snippet shows an example of how to use the links we just discussed. Figure 17-4 illustrates how the code sample was converted into *javadoc*.

In this case *javadoc* was run with two flags, `-version` and `-author;` without these flags the final output would not have included that information.

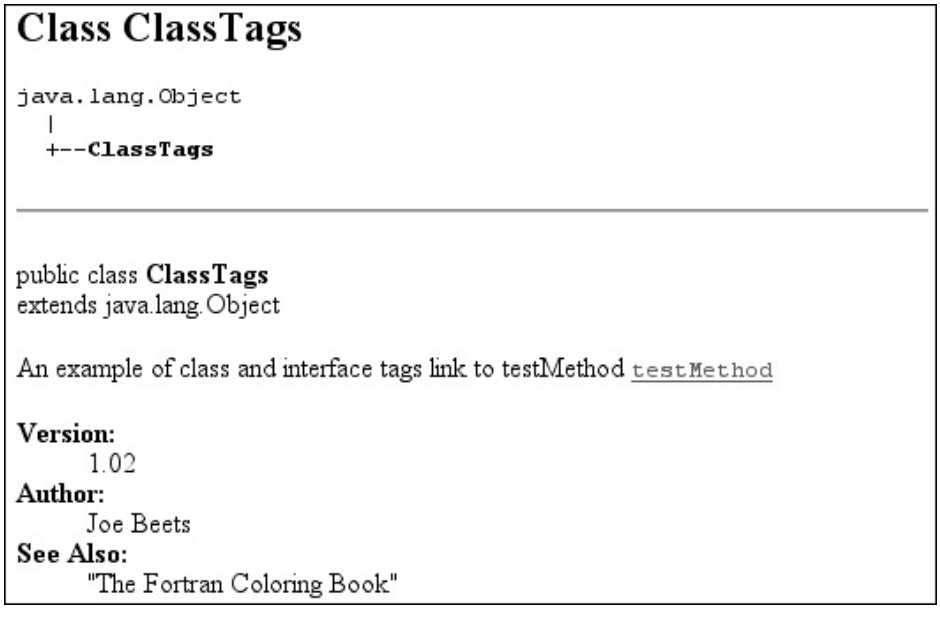
```
/**
 *  An example of class and interface tags
 *  link to testMethod {@link TestJD2#testMethod testMethod}
 *
 *  @author  Joe Beets
 *  @version  1.02
 *
 *  @see "The Fortran Coloring Book"
 */
public class ClassTags { }
```

Useful Tags for Constructors and Methods

Here are some useful tags for constructors and methods:

- `@param` This tag allows you to add method or constructor argument names and a description for the argument to the 'Parameters' section of the *javadoc*.

FIGURE 17-4 Class and interface tags in action



- `@return` This tag allows you to add a method return description to the 'Returns' section of the *javadoc*.
- `@exception` This tag has the same functionality as the `@throws` tag. They allow you to add a 'Throws' subheading to the *javadoc* for the constructor or method being documented. These tags take the exception class name and a description of the exception.

The following code example and Figure 17-5 demonstrate method and constructor tags in action:

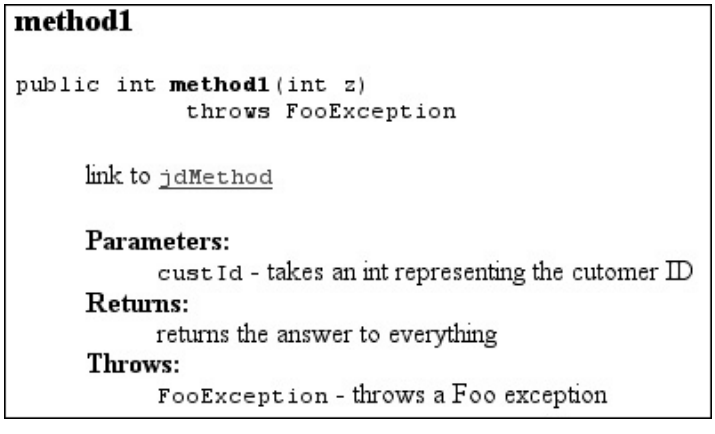
```
/**
 * link to {@link TestJD#jdMethod jdMethod}
 * @param custId takes an int representing the customer ID
 * @return returns the answer to everything
 * @throws FooException throws a Foo exception
 */
public int method1(int z) throws FooException {
```

javadoc Comments for Classes

The *javadoc* comment for the class must directly precede the class declaration. This comment is used to describe the purpose of the class and its capabilities. It may also describe, at a high level, how the class is implemented. The Java API often includes a class comment that can run several pages long. That level of detail is probably not necessary, but it's a good idea to provide a paragraph or two of explanation. Later on in the class you will be documenting your constructors and methods, so this is

FIGURE 17-5

Method and constructor tags in action



not the appropriate place for that documentation. The following is an example of a class level *javadoc* comment:

```
/**
 * The <code>Byte</code> class wraps a primitive type <code>byte</code>
 * in an object.  An object of type <code>Byte</code> contains a single
 * field whose type is <code>byte</code>.
 *
 * In addition, this class provides several methods for converting a
 * <code>byte</code> to a <code>String</code> and a <code>String</code>
 * to a <code>byte</code>, as well as other constants and methods
 * useful when dealing with a <code>byte</code>.
 *
 * @author  Joe Beets
 * @version .997
 *
 */
public class ByteSample {
```

There are several things to notice in the above example. First, notice the use of the tags `<code>` and `</code>`. These tags tell *javadoc* to use a different font (probably a courier font) for the content between the tags, to indicate a code snippet. The next things to notice are the `@author` and `@version` tags whose purposes were described in the previous "Useful Tags for Constructors and Methods" section.

javadoc Comments for Interfaces

The *javadoc* comment for an interface must directly precede the interface declaration. This comment is used to describe the purpose of the interface. The Java API often includes an interface comment that can run several pages long. That level of detail is probably not necessary, but it's a good idea to provide a paragraph or two of explanation. The following is an example of an interface level *javadoc* comment:

```
/**
 * The <code>Runnable</code> interface should be implemented by any class
 * whose instances are intended to be executed by a thread.  The class
 * must define a method of no arguments called <code>run</code>.
 *
 * This interface is designed to provide a common protocol for objects
 * that wish to execute code while they are active.  For example,
 * <code>Runnable</code> is implemented by class <code>Thread</code>.
 * Being active simply means that a thread has been started and has not
 * yet been stopped.
 *
 *
 * @author  Joe Beets
```

```
 * @version .997
 *
 */
public interface RunnableSample {
```

javadoc for Constructors

The *javadoc* comment for a constructor must directly precede the constructor declaration. This comment is used to describe the purpose of the constructor. When creating a comment for a constructor, it's a good idea to provide a paragraph or two of explanation. The following is an example of a constructor comment from the Java API:

```
/**
 * Constructs a newly allocated <code>Byte</code> object that represents
 * the <code>byte</code> value indicated by the <code>String</code>
 * parameter.  The string is converted to a <code>byte</code> value in
 * exactly the same manner used by the <code>parseByte</code> method
 * for radix 10.
 *
 * @param s the <code>String</code> to be converted to <code>Byte</code>
 * @throws NumberFormatException If the <code>String</code> does not
 * contain a parseable <code>byte</code>.
 */
 public Byte(String s) { }
```

javadoc for Methods

The *javadoc* comment for a method must directly precede the method's declaration. This comment is used to describe the purpose of the method. When creating a comment for a method it's a good idea to provide a paragraph or two of explanation. The following is an example of a method comment from the Java API:

```
/**
 * Returns a new <code>String</code> object representing the specified
 * <code>byte</code>.  The radix is assumed to be 10.
 *
 * @param b the <code>byte</code> to be converted
 * @return  the string representation of the specified <code>byte</code>
 */
public static String toString(byte b) {
```

javadoc for Exceptions

The *javadoc* comment for an exception must directly precede the declaration of the method that throws the exception. This comment is a part of the overall comment for the method in question. This comment is used to describe the class of the

exception thrown along with a description of why the exception might be thrown. The Java API often includes an exception comment that can run a page long. That level of detail is probably not necessary, but it's a good idea to provide a paragraph or two of explanation. After a brief discussion of using *javadoc* for variables, we will give an example of a method that throws an exception and the *javadoc* to support that. In this case, we used `@exception` and in an earlier example we used @throws; they work the same way. Finally, see Figure 17-6 to see how this *javadoc* looks in a browser.

javadoc for Variables

The *javadoc* comment for a variable must directly precede the variable declaration. This comment is used to describe the purpose of the variable. The most common reason to use *javadoc* for a variable is for constants (`static final` variables). Constants are often used to represent minimum or maximum values. When documenting a constant it's a good idea to provide a sentence or two of explanation. The following code listing and Figure 17-6 show an exception throwing method and a related constant.

```
/** Minimum allowable Radix is 2 */
public static final int MIN_RADIX = 2;

/**
 * Parses the string argument as a signed byte in the radix specified
 * by the second argument.  The characters in the string must all be
 * digits, of the specified radix.  The resulting <code>byte</code>
 * value is returned.
 * <pre>
 * An exception of type <code>NumberFormatException</code>
 * is thrown if any of the following situations occur:
 *  - The first argument is <code>null</code> or is
 * a string of zero length.
 *    - The radix is either smaller than {@link Tags#MIN_RADIX
 *  Character.MIN_RADIX}
 *    - Any character of the string is not a digit</pre>
 * @param s the <code>String</code> containing the <code>byte</code>
 *  representation to be parsed
 * @param radix the radix to use while parsing s
 * @return the <code>byte</code> value represented by the string
 * argument in the specified radix
 * @exception NumberFormatException If the string does not contain a
 * parseable <code>byte</code>.
 *
 */
public static byte parseByte(String s, int radix) throws
      NumberFormatException
```

FIGURE 17-6 Documenting exceptions and constants

Field Detail

MIN_RADIX

```
public static final int MIN_RADIX
```

Minimum allowable Radix is 2

---

parseByte

```
public static byte parseByte(java.lang.String s,
                             int radix)
                      throws java.lang.NumberFormatException
```

Parses the string argument as a signed byte in the radix specified by the second argument. The characters in the string must all be digits, of the specified radix. The resulting `byte` value is returned.

An exception of type NumberFormatException is thrown if any of the following situations occurs:

- The first argument is null or is a string of zero length.
- The radix is either smaller than Character.MIN_RADIX
- Any character of the string is not a digit

Parameters:
`s` - the `String` containing the `byte` representation to be parsed
`radix` - the radix to use while parsing s

Returns:
the `byte` value represented by the string argument in the specified radix

Throws:
`java.lang.NumberFormatException` - If the string does not contain a parsable `byte`.

The Developer's Choices Document

One of the key pieces of documentation you must provide when you submit your project is the document that reviews and justifies the choices you made during the design phase of your project. This document is affectionately referred to as the 'Choices' document. Your instruction packet will tell you exactly what this document must be named and where it must be located. The intention of this document is to *briefly* explain and justify the thinking you did while designing and implementing your application. In Chapters 15 and 16 we gave you lists of things to think about while designing your application. Those lists can give you good clues as to what to talk about in this document. You will have to make sure that a lot of situations are

handled correctly when you design this application. In some cases, there is no perfect solution, and you will have to consider the tradeoffs and weigh the pros and cons of several possible solutions to a problem. This is the place to review those tradeoffs and pros and cons!

As is hinted at in the instruction packet, the assessors are looking for solutions that are understandable and maintainable. If you come up with a new search algorithm that is 3 percent faster than a well-known solution, you'd better be careful. Your solution had better be really clear and really easy to understand, or you might be penalized for a solution that is a bit slower, but is well known and clear. That said, you will probably have to think about database implementation issues, networking issues (RMI vs. sockets), record-locking issues, and GUI design issues in the course of this project. You may well have other design issues also. Without creating a masters thesis, describe them all in the 'Choices' document.

The Return of Comments and the Versions File

We're almost finished with *javadoc*, but there are still a couple of issues to look at. Besides correctness of your *javadoc*, your documentation should be clear and helpful. Remember, the easier it is for the assessor to figure out what your code is doing, the better your chances for a good score on the exam.

Just a Little More About Comments

We spent a lot of time in this chapter discussing the nuts and bolts of *javadoc*. Now let's spend just a little time discussing the style of the comments that you should create. There is a definite art to proper code commenting—we wish we could say it was a science, but it's not. The key points to remember for this exam are

- Make sure your code comments and clear and concise.
- Make sure the comment you are about to type is necessary.

Keep in mind that the best Java code is to a large degree self-documenting. If you find yourself documenting a lot of your code, think about these things:

- Are your variable names self-descriptive?
- Are your method names self-descriptive?
- Do you find yourself explaining why you wrote your code a certain way?

- Do you just really love to type?
- Remember some of the best Java code you've ever read, how little commenting it needed, and how clear it was.
- Is excessive commenting making up for a muddy design?

Lest We Forget, the Versions File

Not much to say here really. In the interest of being complete, we somewhat redundantly offer this advice. The instruction packet will probably ask you to provide a very small document in which you will list the version of the JDK that you used, and on which host platform(s) you developed and tested your project. Do as you're told. :)

Key Points Summary

Here, in a handy portable format, are the highlights from this chapter:

- You'll probably want to include these six forms of documentation (plus anything else the instructions ask for):
  - The Developer's Documentation—design docs.
  - End User Documentation—how to install and run the application.
  - *javadoc*—the programmer's technical inline comments.
  - The Choices Document—the architect's design choices and tradeoffs.
  - Inline code comments—in addition to the *javadoc*.
  - The Version file—SDK version used and hardware platform(s) used.
- Developer's documents are probably optional documents; include them if they briefly and clearly aid in understanding your project.
- End User documents; keep them simple and concise.

***javadoc* highlights:**

- It's how the Java API was created.
- It generates HTML.
- It's mandatory for your project.

- Comments can have a descriptive section and a tag section.
- Tags can be inline `{@tag}`, or standalone `@tag`.
- *javadoc* has a huge arguments library.
- You can store your command line arguments in a file.
- You can use HTML tags in your *javadoc*.
- Not all *javadoc* tags can be used for all class members.
- You can document the following members in *javadoc*:
  - Classes
  - Interfaces
  - Constructors
  - Variables
  - Methods
  - Exceptions
- The Choices document describes architectural decisions that you make:
  - Database design
  - Networking design
  - GUI design
  - Record-locking design
- Keep your code comments clear and concise.
- Try to make your variable and method names self-documenting.
- Are your comments propping up a muddy design?
- Don't forget your Versions file.