The Design of Distributed Hyperlinked Programming Documentation

Lisa FRIENDLY

Sun Microsystems Inc.
100 Hamilton Avenue, Palo Alto, California 94301 USA
E-mail: friendly@eng.sun.com

ABSTRACT

HotJava is a World-Wide Web browser that adds dynamic behavior to hypertext access by supporting the downloading and execution of architecture-neutral, interactive applets from inside a Web page. HotJava is written in Java, a new object-oriented language and environment developed at Sun Microsystems.

This paper describes the design of the documentation for Java's application programming interface (API), for display and distribution on the World-Wide Web. Following in the footsteps of the literate programming paradigm, the documentation was automatically generated from source code. We designed a syntax for documentation comments which are embedded in the source code and parsed by the Java compiler to produce HTML markup.

The display environment of the World-Wide-Web presented many challenging design requirements for readability, usability, and navigation. This paper discusses the design process, from augmenting the source code commenting syntax to designing the layout for the Web pages. The resulting product is a set of integrated Web pages which are hyperlinked, highly readable, and easily navigated. The API documentation was first published on the World-Wide Web in December of 1994.

1. INTRODUCTION

Java [Sun 95-1] is Sun Microsystems' new object-oriented programming language for Internet-aware, embedded applications. The language is dynamic, multithreaded, and architecture-neutral. HotJava [Sun 95-1] is a Java-based World-Wide Web browser that adds dynamic behavior to hypertext access on the Internet by supporting the execution of interactive programs from within a Web page.

The application programming interface (API) is essential reference material for programmers writing applications in the Java language. This paper describes the design and creation of the API documentation for HotJava's alpha1 software release in December of 1994. The API document is provided as part of the downloadable HotJava binaries, but is also available directly, via the World-Wide Web [SUN 95-2].

1.1 Goals

We began this project by establishing three core goals for the API documentation: the distribution mechanism would be the World-Wide Web, the documentation would be hyperlinked and therefore written in HTML [NCSA 95], and it would be generated directly from source code.

On top of this we postulated some usability goals: readability, browsability, and ease of navigation. Our design had to support users well enough that their focus never had to shift from understanding the API to navigating its documentation.

1.2 Precedents

Java's support for automatic document generation was influenced by earlier work on literate programming. In 1984, Knuth [91, 93] designed the Web system as both a document formatting language and a programming language. Reflecting on Knuth's work, John Bentley wrote [Bentley 86], "Knuth's system allows the programmer to think at a high level, and has the computer do the dirty work of translating the literate description into an executable program." Our goal was to take the executable program and translate it into a literate description. In addition, as Cordes and Brown describe in their overview of the literate programming paradigm [Cordes 91], we hoped to increase awareness among the creators of the Java API that documentation is an integral part of the program itself.

Some literate programming tools include HTML as an output format. Of particular relevance is Ramsey's "noweb" system which attempts to simplify and at the same time generalize the use of literate programming. In contrast to "noweb," the "javadoc" tool we implemented is tied to the Java programming language. However, it is simple to use[1], automatically generates cross-reference and index information, and outputs DTD-compliant HTML (displayable on any WWW browser).

Another difference between javadoc and most literate programming tools is that its purpose is to extract documentation for the programming interface only. For this reason we did not have to extend the program to support method implementations, as most literate programming tools do. Our focus was on figuring out which elements to extract from the source code, how to develop a commenting syntax to support those elements, and from that input alone, to automatically create a well designed piece of Web-based language documentation.

To generate ideas for the document layout, we looked for examples of distributed hypertext language documentation. While there are many excellent APIs published in book format, we could find only a few published on the Internet. Of these, only two contained hypertext links. (See [Icon 95] and [Perl 95].)

---

[1] The user supplies a single command-line argument to javadoc containing either the package name or the file name of the source to be formatted.

2.0 THE JAVA LANGUAGE

2.1 The Application Programming Interface

The API is the programming interface for the packages in the Java language. Packages are collections of classes and interfaces with a common function such as a window toolkit. Classes are collections of methods (behavior) and variables (data) that enable users to write their own applications. Interfaces are groups of methods that can be implemented by one or more classes in a package.

Object-oriented languages such as Java use encapsulation to protect the implementation details of a method. Only the definition of the method is required for a programmer to use it. The implementation is private so that it can be modified later without affecting code which uses the method. In addition, individual variables and methods, as well as entire classes, can be defined as private. Any items which are private are not part of the API.

Java classes are structured hierarchically. They have superclasses from which they inherit methods and variables and subclasses which inherit from them. Every class and interface is a type, just as integers and booleans are types. Every variable has a name and a type. Methods have lists of parameters. Every parameter has a name and a type.

A method may return a value and it may generate an exception for error handling. (Exceptions are also types.) A method can be overridden (superseded) by a method in a subclass. Within a class, a method can be overloaded so that different versions of the method take different parameters. Specialized methods called constructors create instances of the class. There are two categories for variables and methods: static and instance[2]. (For additional information on object-oriented programming see [Lippmann 91].)

It is very inefficient for programmers to refer to the source code every time they need information about the programming interface. Java's API documentation organizes the information they do need and presents it as a set of hyperlinked pages.

2.2 Documentation Comments

In most programming languages, comments in the source code are stripped by the compiler before code generation [Lippman 93]. One of the unique features of Java is that it supports embedded documentation comments (henceforth referred to as doc comments) which are used to generate the API documentation. While parsing the source code to create class files (object files), the compiler converts the declarations and doc comments into HTML documentation.

---

2 Static variables and methods affect the class as a whole. Instance variables and methods affect an individual instance of the class.

Doc comments have a simple formatting syntax of their own. When a doc comment is placed in front of a declaration, it signals to the compiler that the comment is a description of the declared item. The comments take the form, /***text**/ and can be of any length. Comments are associated with classes, interfaces, methods, and variables. Appendix A.1 contains an example of Java source code with standard doc comments. The class illustrated is called containers.Stack.Java (its prefix indicates it is part of the Containers package).

To support the API documentation project, the compiler was extended to parse two new elements within doc comments: embedded HTML and specialized "doc tags." Embedded HTML is useful because it allows programmers to include code samples in their documentation. The doc tags, which all begin with an @ sign, allow programmers to embed comments for a specific attribute (such as a method parameter or a return value). These enhancements to the doc commenting syntax are described further in section 5.

3. DESIGN REQUIREMENTS

To design the document, we took a task-oriented perspective. The API provides information on functionality provided with the Java language. Programmers then create the specialized functionality they need by subclassing the existing classes.

How would users search for such information? How would they browse and navigate? What kind of additional data representations, such as a graphical class hierarchy, should we generate from the basic declarations? What are the specific data attributes that will need special doc tags? Based on the analysis of these and other tasks, the following list of requirements and related design issues was derived.

3.1 Distributed On the World-Wide Web

HotJava is a tool for the Internet. The Java source code and binaries are distributed via the Internet, and all HotJava and Java documentation is available on-line, fully indexed and searchable, from HotJava's World-Wide Web home page. Although the API is a detailed reference document, it must support the project's documentation model.

3.2 Formatted in HTML

To make full use of the Web's capabilities, the API must be a hypertext document, not just a large static file. This requirement brought with it the challenge to maximize the limited document formatting capabilities of HTML while at the same time generating fully DTD-compliant HTML. This project was initiated in the fall of 1994 when very few HTML extensions were supported by the standard Web browsers.

Being compliant with standard HTML was considered essential. Even if potential HotJava users did not have the required hardware for the alpha release[3], they would be able to read the documentation on any platform that supported a WWW browser.

3.3 Automatically Generated from Source Code

The document must be automatically generated from source code. A static document is too likely to get out of synch with the source code on which it is based. We also wanted to generate synthesized information from the source code such as the graphical class hierarchy and callouts for overridden methods.

3.4 Highly Readable on a Display Monitor

The Java API is published on-line. The display quality on a monitor is much lower than on the printed page. This factor had to be considered when designing the various layout formats and deciding how to use links.

3.5 Supports Navigation, Searching, and Browsing

One of the common pitfalls of hyperlinked material is that users get lost easily. Although this document would contain many hyperlinks, it must hold together as a single reference document. User must always be able to determine their exact location in the API and move easily to their next destination. The navigational cues must support many browsing styles: random, hierarchical, and sequential.

3.6 Reflects the Structure of the API

The structure of the document must reflect the package/class structure of the API. The document must be indexed by packages, and within packages by classes and interfaces. Within each class, there must be an index of variables and an index of methods. The user must be able to distinguish between static and instance-level entries and between constructor methods and other methods.

3.7 Content Rich

The commenting syntax must support a content-rich document. When a technical reference document is written by hand, special attention is given to those items and concepts which need the most explanation; where usage is obscure, varied, or relatively complex. The Java document formatting syntax must enable coders to include this kind of information in the source code. The syntax should also support embedded code samples because they are extremely useful to programmers trying to understand how to implement a language.

---

[3] The alpha release of HotJava ran only on Solaris Sparc machines. The documentation was made accessible outside the release via HotJava's WWW pages. Users waiting for ports to other platforms could, in the meantime, read all of the documentation for Java and HotJava.

3.8 Presents an Inviting Visual Interface

The API should present an inviting, engaging visual interface. How should graphical images such as buttons, icons, and illustrations be used to enhance the look and feel of the document? At what level would we achieve an acceptable trade-off between image loading time and the benefits obtained from the use of images?

4. PROTOTYPING AND EVALUATION

4.1 Initial Prototype

The first prototype was hand written in HTML, using the API for a set of hypothetical classes. Together these classes included every entry a class could contain. We tried several formatting variations for each entry type, displayed this prototype on a Web page and began our evaluation. Prototyping in this phase remained at the class level only, as we experimented with various layout, ordering, and formatting possibilities.

4.2 The Design Iteration Cycle

For the next phase of the design we generated the API from the real Java source code. We began with just one package in order to expand the design to the package level, and then generated the document for the entire API. In this phase we began to experiment with extensions to the syntax of the doc comments.

We adopted a design iteration cycle of generating the API document, evaluating it and doing some user testing, modifying the format, and regenerating the document. We preferred to iterate in small increments, repeating the cycle one or more times a day as we fine-tuned the design.

4.3 User Testing

The user testing was informal, but still very valuable. The subject pool was the HotJava project engineers. They represented a subset of the API's projected end-user population. For the testing, we selected team members contributing directly to the API (i.e. writing classes) as well as those focused on other tasks, such as porting.

In brief ten-minute sessions, volunteer subjects were asked to navigate through the API, find certain items, and try getting back to a specific location. At the end of the session they were prompted on how to improve the document's ease-of-use. Their feedback helped us to improve the navigational cues and provide better task-oriented support. It was an important element of the design process.

For example, in the final design, class-level indexes are located on the same Web page as the corresponding API data to which they are linked. This decision was made as a result of user testing which indicated that if the two parts were on different pages, users were too likely to get lost.

4.4 Design Challenges

Many design issues had been anticipated. They ranged from basic structural issues such as how to organize the document, to implementation details such as whether to repeat the method name heading for overloaded methods. Many more issues, especially at the most detailed levels of the format, came to light during the design process itself. This section describes some interesting examples.

4.4.1 How to represent class names in links

It was fairly easy to decide that every class name, when used as a type (i.e., of a variable or parameter), should be a hyperlink to the class itself. However, it required a fair amount of prototyping and user testing to decide whether to "qualify"[4] the name when it appeared in this context. Prototyping revealed that fully qualifying the name made it more difficult to read and select.

Instead, we provided other ways to determine the ancestry of a class. Users can traverse the link back to the parent class; when the cursor is placed over the link, the HotJava status line displays the package-level ancestry, and most importantly, the ancestry is displayed in hyperlinked form in the hierarchy diagram at the beginning of every class.

4.4.2 How to format method declarations

Another case that required a lot of experimentation was the format for method declarations. A commonly used format is to print the method name and its parameter list on one line. However, when we displayed this format on the monitor, with each parameter type (class name) as a link, the declaration wasn't very readable.

Links on a Web page are displayed in a blue font which is already a little more difficult to read than a black font. When many links are concatenated together on one line (some methods have as many as eight parameters), they become difficult to distinguish. After much evaluation, we settled on a format where each *type/parameter name* pair is on a separate line.

4.4.3 Links to inherited methods and variables

The issue of whether to include links from within a class to the methods inherited by that class was a particularly difficult problem. Looking at the class hierarchy, a programmer can easily determine it's superclasses. However, it is undeniable that having immediate access to the inherited methods and variables would also be valuable.

The problem lies with the fact that quite a lot of methods are inherited, few of which the programmer may actually need to see. In Java, every class ultimately

---

[4] To qualify a class name is to precede it with the name of one or more of its superclasses. The full ancestry provides essential information but can be quite lengthy, depending on the depth of subclassing.

inherits from class Object. It was clear that we didn't want to include Object's methods with every class, and for a brief time we thought that this exclusion might make the problem manageable.

From a formatting perspective, we tried several alternatives; adding inherited methods at the end of the index list, integrating them alphabetically into the index of methods, or using a different color code to differentiate them. But in the end, any of these options simply made the list of methods too unwieldy. More essential information was being obscured.

It did occur to us to add an @inherited tag and let the programmer decide which inherited methods should be included. But this approach would lead users to assume that the important inherited methods were being called out. If they did not see such a callout they were likely to assume that the method they sought did not exist. Similarly, if a method were added to a superclass, did the programmer who created it become responsible for updating the documentation for all subclasses? In the end, we decided not to include links to inherited items but agreed that the issue should be revisited in future releases.

5. EXTENSIONS TO JAVADOC

While conducting the design analysis, we identified two components we wanted to include in the API document which could not be supported by Java's standard doc commenting syntax: embedded code samples and explanations for specific attributes of a method, such as return values and individual parameters.

As a result, we extended the javadoc program to support them. We used embedded HTML for the code samples and designed special doc tags to describe certain attributes of classes, methods, and variables. In the process, we realized that HTML could be used to enhance other doc comments as well, by adding markup for such formatting characteristics as italics and bold.

5.1 Embedded HTML

Code samples are extremely useful to programmers trying to understand a new language. We use standard HTML to embed code samples. Javadoc parses the HTML and includes it when it creates the generated API document (which of course, is all formatted in HTML). When the markup is passed through the browser's HTML formatter, the code sample is correctly displayed on the Web page.

5.2 Doc Tags

Several "doc tags" were designed for programmers to include within an existing code comment (at their discretion). These tags support "See Also" references for a class or method (@see), parameter definitions (@param), descriptions for exceptions (@exception), and explanations of a method's return value (@return). Tags for author (@author) and version (@version) were also created.

Tags must start at the beginning of a line. The subsections below list the valid tags for classes, variables, and methods and provide examples.

5.2.1 Doc tags for a class:

Class comments may contain @see tags only.

```
@see classname
```

Creates a hyperlinked "See Also" entry to the class.

```
@see classname#method-name
```

Creates a hyperlinked "See Also" entry to the method in the specified class.

An example of a class comment:

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
 * @see             awt.BaseWindow
 * @see             awt.Button
 * @version         1.2 12 Dec 1994
 * @author          Sami Shaio
 */
class Window extends BaseWindow {
          ...
}
```

5.2.2 Doc tags for a variable:

Variable comments can contain only the @see tag.

An example of a variable comment:

```
/**
 * The X-coordinate of the window
 * @see window#1
 */
int x = 1263732;
```

5.2.3 Doc tags for a method:

Method comments may contain @see tags, as well as @param, @return, and @exception:

```
@param parameter-name description...
```

Adds a parameter to the "Parameters" section. The description may be continued on the next line. Note: The first use of @param within a code comment creates a section labeled “Parameters” in the documentation for that method.

```
@return  description...
```

Adds a "Returns" entry, which contains the description of the return value.

```
@exception exception-name description...
```

Adds a "Throws" entry, which contains the name of the exception that may be thrown by the method. The exception is linked to its class documentation.

An example of a method comment:

```
/**
 * Return the character at the specified index.
 * An index ranges from <tt>0</tt> to
 * <tt>length() - 1</tt>.
 * @param index   The index of the desired
 * character
 * @return        The desired character
 * @exception    StringIndexOutOfRangeException
 * When the index is not in the range
 * <tt>0</tt>to <tt>length() - 1</tt>.
 */
public char charAt(int index) {
          ...
```

6. IMPLEMENTATION

This section describes the final design of the API documentation for Java's alpha software release in December of 1994. Section 7 provides sample source code, markup, and output. The best way to understand the implementation is to visit the API document at http://java.sun.com/1.0alpha2/doc/api/packages.html.

There are three levels to the API:

All Packages - An index of links to all the packages in the Java API.

This Package - Linked indexes for all the classes, interfaces and exceptions within a package.

This Class - Linked indexes for the variables and methods within the class. At this level, the indexes include the declaration and a short comment. The indexes are followed, on the same Web page, by the detailed API for each of these sections.

The entries in the indexes are listed alphabetically to support browsing by name, while the entries in the detailed API are grouped as they are in the source code, to preserve logical groupings created by the programmer.

At the class level, color coded balls precede each entry to indicate the type of variable or method. (Magenta = instance variables, blue = static variables, yellow = constructors, red = instance methods, green = static methods.) The balls in the index are half-size, so users know immediately they are in the index section of the page.

6.1 How Hyperlinks are Used

At the top of every page of the document there are navigational anchors to the other levels and to Previous and Next (class or interface). At the top of each class/interface there is a hyperlinked drawing of the class hierarchy. This is followed by a link to the superclass and a link to the interfaces implemented by the class, if any.

Every exception thrown by a method is a link to the exception's class. Every "See Also" reference is also a link. When a method overrides a method in the superclass the document displays the entry, "Overrides: *method name* in *class name*." Both the method name and the class name are links.

Several features of the HotJava browser itself further support searching and navigation, including forward, back and home buttons and a search tool for locating items in any of the documentation we provide.

6.3 Generated Components

The decision to generate the documentation automatically made it possible to create pieces of information which could not easily have been gathered in any other way. These components add great value to the document and make it much more than just a highly readable subset of the source code. Probably the most important example is the hyperlinked hierarchy diagram for each class. Another example is the cross reference to overridden methods which links back to the method being overridden. (This is an essential piece of information for programmers.)

6.4 The Use of Graphical Images

We wanted the document to present an engaging visual interface. At the same time we didn't want the graphical elements to be a distraction or to slow down the loading time for a Web page. Because images on a page are cached, it is the number of different images and not their frequency which can slow down loading. We settled on a very sparse use of images; only the colored balls and some graphical headings in which text is aligned on a slight curve and underlined in yellow.

7. SAMPLE SOURCE, MARKUP AND DOCUMENTATION

Appendix A.2 contains the Java source code for the same class shown in Appendix A.1 (referenced earlier). However, the source code in A.2 adds embedded HTML and doc tags to the existing code comments. The HTML markup generated for this enhanced source code is shown in Appendix A.3. Appendix B illustrates the API documentation which results.

8. CONCLUSION

The automatically-generated API document described in this paper was published on-line for HotJava's alpha1 release in December of 1994 and again in March of 1995 for the alpha2 release.

The document which resulted from the design process is much more than a subset of the Java source code. It contains essential information that can not be readily obtained by reading the source code. It is on-line, hyperlinked, easily navigated and supports full text searching.

The Application Programming Interface document is an essential reference tool for anyone programming in the Java language. The better the document, the more likely it is to increase the acceptance of Java as a new programming language for executable applications. Users of both alpha releases indicate that they find it to be a very useful document.

8.1 Future Directions

The next generation of the HotJava browser will be based on a WYSIWYG HTML editor. That is, users will modify their Web pages from within the browser window itself. One possible direction for javadoc is to follow up on the finding by Brown and Childs [Brown 90] that programmers prefer a graphical user interface for literate programming tools. Perhaps javadoc should be integrated into this browser/editor so that programmers can do side-by-side editing of source and comments, which the editor would integrate for them. As part of the editing process, the programmer could compile the file and view its output in yet another portion of the browser window.

As the Java language becomes more ubiquitous, programmers will use javadoc to create their own documentation. Feedback from their experience will enable us to make enhancements to the program and chart its future development.

9. REFERENCES

[Bentley 86] Bentley J., "Programming Pearls: Sampling," CACM, May, 1986.

[Brown 90] Brown M. and Childs B. "An Interactive Environment for Literate Programming," Journal of Structured Programming, Vol. 11, No. 1, 1990.

[Cordes 91] Cordes D. and Brown M. "The Literate-Programming Paradigm," Computer, June 1991.

[ICON 95] ICON Programming Specification. http://www.cs.arizona.edu/icon/www/reference.html.

[Knuth 84] Knuth D., "Literate Programming," Computer Journal, May 1984.

[Knuth 91] Knuth D., *Literate Programming*, Center for the Study of Language and Information, Stanford University, 1992.

[Knuth 93] Knuth D. and Levy S., *The CWEB System of Structured Documentation*, Addison-Wessley, 1993.

[Lippman 91] Lippman S., *C++ Primer,* Addison-Wesley, 1991.

[NCSA 95] HTML Primer. http://www.ncsa.uiuc.edu/demoweb/html-primer.html

[PERL 95] PERL Programming Specification. http://www.metronet.com/0/perlinfo/perl5/manual/perl.html

[Ramsey 94] Ramsey N., Literate Programming Simplified. IEEE Software, September, 1994.

[SUN 95-1] HotJava Home Page. http://java.sun.com/

[SUN 95-2] Java API. http://java.sun.com/1.0alpha2/doc/api/packages.html

Acknowledgments

Arthur van Hoff was my primary colleague on the API documentation project. Arthur wrote the javadoc program and provided very useful review comments for this paper. Mark Scott Johnson, Herb Jellinek, and Kathy Walrath also made valuable review comments, as did the reviewers from the IWHD Program Committee.

APPENDIX A - SAMPLE SOURCE CODE AND MARKUP

A.1 Source Code for Stack.java

```
package containers;
/**
 * A simple FIFO container. It implements a fixed size
 * stack onto which objects can be pushed. The
 * following example prints the two strings in
 * reverse order.
*/
public class Stack {
    private Object stack[];
    private int sp;

    /**
     * Constructs a new stack of the given size.
     */
    public Stack(int size) {
          stack = new Object[size];
    }

    /**
     * Pushes an object onto the stack.
     */
    public synchronized void push(Object obj) {
          if (sp == stack.length) {
              throw new StackOverFlowException(this);
          }
          stack[sp++] = obj;
    }

    /**
     * Pops an object off the stack.
     */
    public synchronized Object pop() {
          if (sp == 0) {
              throw new StackUnderFlowException(this);
          }
          return stack[--sp];
    }

    /**
     * Returns the top object of the stack without
     * popping it off the stack.
     */
    public synchronized Object top() {
          if (sp == 0) {
              throw new StackUnderFlowException(this);
          }
          return stack[sp - 1];
    }
```

```
    /**
     * Returns the current depth of the stack.
     */
    public int depth() {
          return sp;
    }
}
```

A.2 Enhanced Source Code for Stack.java

```
package containers;
/**
 * A simple <em>FIFO</em> container. It implements a
 * fixed size stack onto which objects can be pushed.
 * The following example prints the two strings in
 * reverse order.
 * <pre>
 *       Stack s = new Stack(10);
 *       s.push("One");
 *       s.push("Two");
 *       System.out.println(s.pop());
 *       System.out.println(s.pop());
 * </pre>
 *
 * @see java.lang.Object
 * @author Arthur van Hoff
 * @version 1.0
 */
public class Stack {
    private Object stack[];
    private int sp;

    /**
     * Constructs a new stack of the given size.
     */
    public Stack(int size) {
          stack = new Object[size];
    }

    /**
     * Pushes an object onto the stack.
     * @param obj the object
     * @exception StackOverFlowException stack overflow
     * @see containers.Stack#pop
     */
    public synchronized void push(Object obj) {
          if (sp == stack.length) {
              throw new StackOverFlowException(this);
          }
          stack[sp++] = obj;
    }
```

```
    /**
     * Pops an object of the stack.
     * @return         the object
     * @exception StackUnderFlowException stack under
     * flow
     * @see containers.Stack#push
     */
    public synchronized Object pop() {
          if (sp == 0) {
              throw new StackUnderFlowException(this);
          }
          return stack[--sp];
    }

    /**
     * Returns the top object of the stack without
     * popping it off the stack.
     * @return the top of the stack
     */
    public synchronized Object top() {
          if (sp == 0) {
              throw new StackUnderFlowException(this);
          }
          return stack[sp - 1];
    }

    /**
     * Returns the current depth of the stack.
     * @return the depth of the stack
     */
    public int depth() {
          return sp;
    }
}
```

A.3 HTML Markup for Stack.java

```
<html>
<head>
<a name="_top_"></a>
<title>
Class containers.Stack
</title>
</head>
<body>
<a name="_top_"></a>
<pre>
<a href="packages.html">All Packages</a>
<a href="containers.html">This Package</a>
<a href="containers.html">Previous</a>
<a href="containers.html">Next</a></pre>
```

```
<hr>
<h1>
Class containers.Stack</h1>
<pre>
<a
href="java.lang.Object.html#_top_">java.lang.Object</a>
   |
   +----containers.Stack
</pre>
<hr>
<dl>
<dt>
public class <b>
Stack</b>
<dt>
extends
<a href="java.lang.Object.html#_top_">Object</a>
</dl>
A simple <em>FIFO</em> container. It implements a fixed
size stack onto which objects can be pushed. The
following example prints the two strings in reverse
order.
<pre>
          Stack s = new Stack(10);
          s.push("One");
          s.push("Two");
          System.out.println(s.pop());
          System.out.println(s.pop());
</pre>
<dl>
<dt>
<b>See Also:</b>
<dd>
<a href="java.lang.Object.html#_top_">Object</a><dt>
<b>Author:</b>
<dd>
Arthur van Hoff<dt>
<b>Version:</b>
<dd>
1.0
</dl>
<hr>
<a name="index"></a>
<h2>
<img src="images/constructor-index.gif" width=275
height=38>
</h2>
<dl>
<dt>
<img src="images/yellow-ball-small.gif" width=6
height=6>
<a href="#Stack(int)"><b>Stack</b></a>(int)
<dd>
```

```
Constructs a new stack of the given size.
</dl>
<h2>
<img src="images/method-index.gif" width=207 height=38>
</h2>
<dl>
<dt>
<img src="images/red-ball-small.gif" width=6 height=6>
<a href="#depth()"><b>depth</b></a>()
<dd>
Returns the current depth of the stack.
<dt>
<img src="images/red-ball-small.gif" width=6 height=6>
<a href="#pop()"><b>pop</b></a>()
<dd>
Pops an object off the stack.
<dt>
<img src="images/red-ball-small.gif" width=6 height=6>
<a
href="#push(java.lang.Object)"><b>push</b></a>(Object)
<dd>
Pushes an object onto the stack.
<dt>
<img src="images/red-ball-small.gif" width=6 height=6>
<a href="#top()"><b>top</b></a>()
<dd>
Returns the top object of the stack without popping it
off the stack.
</dl>
<a name="constructors"></a>
<h2>
<img src="images/constructors.gif" width=231 height=38>
</h2>
<a name="Stack"></a>
<a name="Stack(int)">
<img src="images/yellow-ball.gif" width=12 height=12>
</a>
<b>
Stack
</b>
<pre>
  public Stack(int size)
</pre>
<dl>
<dd>
Constructs a new stack of the given size.
</dl>
<p>
<a name="methods"></a>
<h2>
<img src="images/methods.gif" width=151 height=38>
</h2>
<a name="push(java.lang.Object)">
```

```
<img src="images/red-ball.gif" width=12 height=12>
</a>
<a name="push">
<b>
push
</b>
</a>
<pre>
  public synchronized void push(<a
href="java.lang.Object.html#_top_">Object</a> obj)
</pre>
<dl>
<dd>
Pushes an object onto the stack.
<dl>
<dt>
<b>Parameters:</b>
<dd>
obj
-
          the object
<dt>
<b>Throws:</b> <a
href="containers.StackOverFlowException.html#_top_">Sta
ckOverFlowException</a>
<dd>
 stack overflow
<dt>
<b>See Also:</b>
<dd>
<a href="#pop">pop</a>
</dl>
</dl>
<p>
<a name="pop()">
<img src="images/red-ball.gif" width=12 height=12>
</a>
<a name="pop">
<b>
pop
</b>
</a>
<pre>
  public synchronized <a
href="java.lang.Object.html#_top_">Object</a> pop()
</pre>
<dl>
<dd>
Pops an object off the stack.
<dl>
<dt>
<b>Returns:</b>
<dd>
```

```
            the object
<dt>
<b>Throws:</b> <a
href="containers.StackUnderFlowException.html#_top_">
StackUnderFlowException</a>
<dd>
 stack under flow
<dt>
<b>See Also:</b>
<dd>
<a href="#push">push</a>
</dl>
</dl>
<p>
<a name="top()">
<img src="images/red-ball.gif" width=12 height=12>
</a>
<a name="top">
<b>
top
</b>
</a>
<pre>
  public synchronized
<a href="java.lang.Object.html#_top_">Object</a> top()
</pre>
<dl>
<dd>
Returns the top object of the stack without popping it
off the stack.
<dl>
<dt>
<b>Returns:</b>
<dd>
the top of the stack
</dl>
</dl>
<p>
<a name="depth()">
<img src="images/red-ball.gif" width=12 height=12>
</a>
<a name="depth">
<b>
depth
</b>
</a>
<pre>
  public int depth()
</pre>
<dl>
<dd>
Returns the current depth of the stack.
<dl>
```

```
<dt>
<b>Returns:</b>
<dd>
the depth of the stack
</dl>
</dl>
<p>
<hr>
<pre>
<a href="packages.html">All Packages</a>
<a href="containers.html">This Package</a>
<a href="containers.html">Previous</a>
<a href="containers.html">Next</a></pre>
</body>
</html>
```

APPENDIX B - API DOCUMENTATION

Note: This example shows one Web page, broken into sections for this paper.

All Packages   This Package   Previous   Next

Class containers.Stack

```
java.lang.Object
   |
   +----containers.Stack
```

---

public class **Stack**
extends Object

A simple *FIFO* container. It implements a fixed size stack onto which objects can be pushed. The following example prints the two strings in reverse order.

```
Stack s = new Stack(10);
s.push("One");
s.push("Two");
System.out.println(s.pop());
System.out.println(s.pop());
```

See Also:
Object

Author:
Arthur van Hoff

Version:
1.0

Constructor Index

- **Stack**(int)
  Constructs a new stack of the given size.

Method Index

- **depth**()
  Returns the current depth of the stack.
- **pop**()
  Pops an object of the stack.
- **push**(Object)
  Pushes an object onto the stack.
- **top**()
  Returns the top object of the stack without poping it of the stack.

Constructors

Stack

```
public Stack(int size)
```

Constructs a new stack of the given size.

Methods

push

```
public synchronized void push(Object obj)
```

Pushes an object onto the stack.

Parameters:
obj – the object

Throws: StackOverFlowException
stack overflow

See Also:
pop

pop

```
public synchronized Object pop()
```

Pops an object of the stack.

Returns:
the object

Returns:
: the object

Throws: StackUnderFlowException
: stack under flow

See Also:
: push

● top

```
public synchronized Object top()
```

Returns the top object of the stack without poping it of the stack.

Returns:
: the top of the stack

● depth

```
public int depth()
```

Returns the current depth of the stack.

Returns:
: the depth of the stack
