Javadoc Tool

A tool for generating API documentation in HTML format from doc comments in source code.

Javadoc

What is Javadoc?

Javadoc, part of the standard Java JDK, is a computer software tool for generating API documentation into HTML format from Java source code. Javadoc is the industry standard for documenting Java classes.

Traditionally, when you write a program you must write a separate document recording how the program works.
When the program changes, the separate documentation must change as well. For large programs that are constantly being changed and improved, it is a tedious and error-prone task to keep the documentation consistent with the actual code.

Javadoc is a tool that helps with this problem.

Instead of writing and maintaining separate documentation, the programmer writes specially-formatted comments in the Java code itself. Javadoc takes these comments and transforms them into documentation in HTML (web page) format.

General Format of Javadoc Comments

Javadoc comments have the following format:

```
/**
 * 1) Summary sentence.
 * 2) More general information about the
 * program, class, method or variable which
 * follows the comment, using as many lines
 * as necessary.
 *
 * 3) Zero or more “tags” to specify more specific kinds
 * of information
 */
```

You start out a *javadoc* comment with the normal beginning of comment delimiter (/*) followed by another (*).

```
/**
```

All following lines start with an asterisk (generally lined up under the first asterisk in the first line).

```
/**
 *
 *
```

The last line contains just the normal end of comment delimiter (*/).

```
/**
 *
 *
 */
```

General Format of Javadoc Comments (cont.)

The first sentence is a "summary sentence". This should be a short description of the program, class, method, or variable that can stand on its own. Even though it's referred to as a "summary sentence", it is often a phrase rather than a complete sentence, as in the example shown below. It can extend over several lines if needed; just don't include any blank lines. The summary sentence is terminated by the first period (“.”), so avoid using abbreviations such as "e.g." in this sentence.

Following the summary sentence, you may include more text, general descriptions, etc. to give more details. Again, don't include any blank lines.

Following this general description, there should be a blank line, and then a sequence of "tags" (these tags are described later). Different tags will be appropriate for different situations. Here is an example of a *javadoc* comment without tags which describes the variable declared immediately below it:

```
/**
 * The number of students in the class.  This variable must not be
 * negative or greater than 200.
 */
public int numStudents;
```

Note: You still can and should have regular comments within methods, describing local variables and the computing going on inside the methods. There is, however, no javadoc format for these comments. Use // or /...*/, whichever you prefer.*

Javadoc Tags

Javadoc parses special “tags” that are recognized when they are embedded within a *javadoc* comment. These tags enable you to auto-generate a complete, well-formatted API from your source codes. The tags start with an "at" sign (@) and are case-sensitive.

Tags must start at the beginning of a line. Keep tags with the same name together within a *javadoc* comment. For example, put all @author tags (“@author” tag is described later) together so that javadoc can tell where the list ends.

The general form of a tag is:

```
@tagname information
```

The name of the tag specifies what kind of information you are providing and the “information” provides the details for that tag. For example, the “author” tag is used to tell us who wrote a class or program, as in

```
@author William Shakespeare
```

Each tag should start on a new line. The tag comments can extend into multiple lines if necessary, but there should be no blank lines in between tags.

Class and Interface - Documentation Tags

The following are the most common tags that appear in the documentation comment for a class or interface. Class/Interface documentation tags include:

- `@see` *(described on page 15)*
- `@since` *(described on page 14)*
- `@deprecated` *(described on page 16)*
- `@author` *(described on page 13)*

An example of a class *javadoc* comment:

Note that HTML tags can be used to format your resulting text

```
/**
 * A class representing a window on the screen.
 * Usage example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
 * @author     Bart Simpson
 * @since      Version 1.0
 * @see        java.awt.BaseWindow
 * @see        java.awt.Button
 */
class Window extends BaseWindow {
   ...
}
```

Class and Interface - Documentation Tags (cont.)

Class/Interface-level comments provide a description of the class. They are placed above the code that declares the class. Class/Interface -level comments generally contain author tags, and a description of the class. Another example class/interface -level *javadoc* comment is below:

```
/**
 * The Inventory class contains the amounts of all the
 * inventory in the CoffeeMaker system.<br> The types of
 * inventory in the system include coffee, milk, sugar
 * and chocolate.
 *
 * @author Joe Java
 */
```

The resulting HTML Javadoc:

Class Inventory

```
java.lang.Object
  └ Inventory
```

```
public class Inventory
extends java.lang.Object
```

The Inventory class contains the amounts of all the inventory in the CoffeeMaker system.
The types of inventory in the system include coffee, milk, sugar and chocolate.

Author:
Joe Java

[…]

Field - Documentation Tags

The following are the most common tags that appear in the documentation comment for a field. Field documentation tags include:

- `@see` *(described on page 15)*
- `@since` *(described on page 14)*
- `@deprecated` *(described on page 16)*
- `{@value}` *(described on page 16)*

An example of a field *javadoc* comment:

```
/**
 * The X-coordinate of the component.
 *
 * @see #getLocation()
 */
private int xCoord;
```

Field - Documentation Tags (cont.)

Field-level comments describe the data attributes of the class. Another example of these field-level *javadoc* comments is below:

```
/**
 * Inventory for coffee - should be zero or more.
 *
 * @see #getAmtCoffee()
 */
private int amtCoffee;
```

The resulting HTML Javadoc:

[…]

Field Summary

private int	`amtCoffee` Inventory for coffee - should be zero or more.

[…]

Field Detail

amtCoffee

`private int` **`amtCoffee`**

Inventory for coffee - should be zero or more.

See Also:

`getAmtCoffee()`

[…]

Constructor and Method - Documentation Tags

The following are the most common tags that appear in the documentation comment for a constructor or method (Note that the `@return` tag cannot appear in a constructor). Method/Constructor documentation tags include:

- `@see` *(described on page 15)*
- `@since` *(described on page 14)*
- `@deprecated` *(described on page 15)*
- `@param` *(described on page 13)*
- `@return` *(described on page 14)*
- `@throws and @exception` *(described on page 14)*

An example of a method javadoc comment:

```
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *              if the index is not in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
   ...
}
```

Constructor and Method - Documentation Tags (cont.)

Constructor/Method-level comments describe the methods and constructors. Method and constructor comments may contain tags that describe the parameters of the method. Method comments may also contain return tags. Another example of these javadoc comments is below (and on the following page):

```
/**
 * Adds to the units of coffee in the Inventory.
 * Negative numbers are ok, as they represent deductions from inventory.<br>
 * The method then returns the resulting amount of coffee.
 *
 * @param newCoffeeIn The number of units to add to the inventory
 * @return The int resulting inventory value
 * @throws DataValidationException If the resulting coffee value would be less than zero
 */

public int addAmtCoffee(int newCoffeeIn) throws DataValidationException
{
   if (coffee - newCoffeeIn < 0)
       throw new DataValidationException("Resulting value invalid: " +  (coffee - newCoffeeIn));

   coffee += newCoffeeIn;
}
```

Constructor and Method - Documentation Tags (cont.)

The resulting HTML Javadoc:

[…]

Method Summary

int	addAmtCoffee (int newCoffeeIn) Adds to the units of coffee in the Inventory.

[…]

Method Detail

addAmtCoffee

```
public int addAmtCoffee(int newCoffeeIn)
                 throws DataValidationException
```

Adds to the units of coffee in the Inventory. Negative numbers are ok, as they represent deductions from inventory.
The method then returns the resulting amount of coffee.

Parameters:

newCoffeeIn - The number of units to add to the inventory

Returns:

The int resulting inventory value

Throws:

`DataValidationException` - If the resulting coffee value would be less than zero

[…]

Summary of Javadoc Tags and Their Usage

@author *name-text*

Adds an "Author" entry with the specified *name-text* to the generated docs when the -author *javadoc* option is used. A *javadoc* comment may contain multiple `@author` tags. You can specify one name per `@author` tag or multiple names per tag. In the former case, the *javadoc* tool inserts a comma (,) and space between names. In the latter case, the entire text is simply copied to the generated document without being parsed. For example:

```
@author Harrison Jones
```

This generates a reference as:

Author:

Harrison Jones

@param *name description*

The `@param` tag is followed by the name ***(not** data type)* of the parameter, followed by a description of the parameter. By convention, the first noun in the description is the data type of the parameter. (Articles like "a", "an", and "the" can precede the noun.) Additional spaces can be inserted between the name and description so that the descriptions line up in a comment block. Dashes or other punctuation should not be inserted before the description, as the *javadoc* tool already inserts one dash.The description begins with a lowercase letter if it is a phrase (contains no verb), or an uppercase letter if it is a sentence. End the phrase with a period only if another phrase or sentence follows it. For example:

```
@param observer  the image observer to be notified
```

This generates a reference as:

Parameters:

observer - the image observer to be notified

When writing the descriptions, in general, start with a phrase and follow it with sentences if they are needed:

- When writing a phrase, do not capitalize and do not end with a period:
  ```
  @param x  the x-coordinate, measured in pixels
  ```
- When writing a phrase followed by a sentence, do not capitalize the phrase, but end it with a period to distinguish it from the start of the next sentence:
  ```
  @param x  the x-coordinate. Measured in pixels.
  ```
- If you prefer starting with a sentence, capitalize it and end it with a period:
  ```
  @param x  Specifies the x-coordinate, measured in pixels.
  ```
- When writing multiple sentences, follow normal sentence rules:
  ```
  @param x  Specifies the x-coordinate. Measured in pixels.
  ```

Summary of Javadoc Tags and Their Usage (cont.)

@return *description*

Adds a "Returns" section with the *description* text. This text should describe the return type and permissible range of values. This tag is valid only in a javadoc comment for a method.

Omit `@return` for methods that return void and for constructors; include it for all other methods. Having an explicit `@return` tag makes it easier for someone to find the return value quickly. Whenever possible, supply return values for special cases (such as specifying the value returned when an out-of-bounds argument is supplied). Use the same capitalization and punctuation as you used in `@param`. For example:

```
@return the number of integers read
```

This generates a reference as:

Returns:

the number of integers read

@throws *class-name description* / **@exception** *class-name description*

The `@throws` and `@exception` tags are synonyms. Adds a "Throws" subheading to the generated documentation, with the *class-name* and *description* text. The *class-name* is the name of the exception that may be thrown by the method. This tag is valid only in the doc comment for a method or constructor. Multiple `@throws` tags can be used in a given doc comment for the same or different exceptions.

To ensure that all checked exceptions are documented, if a `@throws` tag does not exist for an exception in the throws clause, the *javadoc* tool automatically adds that exception to the HTML output (with no description) as if it were documented with `@throws` tag. For example:

```
@throws DataValidationException if the resulting coffee value would be less than zero
```

This generates a reference as:

Throws:

DataValidationException - if the resulting coffee value would be less than zero

@since *since-text*

Adds a "Since" heading with the specified *since-text* to the generated documentation. The text has no special internal structure. This tag is valid in any doc comment: overview, package, class, interface, constructor, method or field. This tag means that this change or feature has existed since the software release specified by the *since-text*. For example:

```
@since Version 1.0
```

This generates a reference as:

Since:

Version 1.0

Summary of Javadoc Tags and Their Usage (cont.)

@see *reference*

Adds a "See Also" heading with a link or text entry that points to *reference*. A javadoc comment may contain any number of `@see` tags, which are all grouped under the same heading. This tag is valid in any doc comment: overview, package, class, interface, constructor, method or field. For inserting an in-line link within a comment to a package, class or member, use `{@link}`.

@see "*string*"

Adds a text entry for *string*. No link is generated. The *string* is a book or other reference to information not available by URL. The *javadoc* tool distinguishes this from the previous cases by looking for a double-quote (") as the first character. For example:

```
@see "The Java Programming Language"
```

This generates a reference as:

See Also:

"The Java Programming Language"

@see *label*

Adds a link as defined by *URL#value*. The *URL#value* is a relative or absolute URL. The *javadoc* tool distinguishes this from other cases by looking for a less-than symbol (<) as the first character. For example:

```
@see <a href="spec.html#section">Java Spec</a>
```

This generates a reference as:

See Also:

Java Spec

@see *package.class#member label*

Adds a link, with visible text *label*, that points to the javadoc documentation for the specified package/class/member. The *label* is optional; if omitted, the link appears as the visible text. Use javadoc flag "-noqualifier" to globally remove the package name from this visible text. Use the label when you want the visible text to be different from the auto-generated visible text. For example:

```
@see String#equals(Object)
```

This generates a reference as:

See Also:

String.equals(Object)

Summary of Javadoc Tags and Their Usage (cont.)

@deprecated

Adds a comment indicating that this API should no longer be used (even though it may continue to work). The *javadoc* tool moves the *deprecated-text* ahead of the main description, placing it in italics and preceding it with a bold warning: "Deprecated". This tag is valid in all doc comments: overview, package, class, interface, constructor, method and field.
The first sentence of *deprecated-text* should at least tell the user when the API was deprecated and what to use as a replacement. The *javadoc* tool copies just the first sentence to the summary section and index. Subsequent sentences can also explain why it has been deprecated. You should include a {@link} tag that points to the replacement API. For example:

```
 * @deprecated  As of JDK 1.1, replaced by
 *              {@link #setBounds(int,int,int,int)}
```

This generates a reference as:

Deprecated. *As of JDK 1.1, replaced by* *setBounds(int,int,int,int).*

{@value *package.class#field***}**

When `{@value}` is used in the javadoc comment of a `static final` field, it displays the value of that constant. For example:

```
/**
 * The value of this constant is {@value}.
 */
public static final int MAX_COFFEE = 199;
```

This generates a reference as:

MAX_COFFEE
`public static final int` **`MAX_COFFEE`**

The value of this constant is 199.

See Also:
Constant Field Values

Summary of Javadoc Tags and Their Usage (cont.)

@link *reference label*

Inserts an in-line link with visible text *label* that points to the documentation for the *reference* (a package, class or member name of a referenced class). This tag is valid in all doc comments: overview, package, class, interface, constructor, method and field, including the text portion of any tag (such as `@return`, `@param` and `@deprecated`).

The `{@link}` tag generates an in-line link rather than placing the link in the "See Also" section. Also, the `{@link}` tag begins and ends with curly braces to separate it from the rest of the in-line text. There is no limit to the number of {@link} tags allowed in a sentence. You can use this tag in the main description part of any documentation comment or in the text portion of any tag (such as @deprecated, @return or @param).

{@link *package.class label* **}**

Here, *package.class* is the name of the class you wish to link to. The class name must include the package name if the target class is not in the same package. *Label* is optional. It is the text that will be displayed in the document. If the label is skipped, the result document will display the full class name. For example:

```
* Refer to the {@link Coffee Coffee} class for Coffee-specific details.
```

This generates a link as:

Refer to the Coffee class for Coffee-specific details.

{@link *#member label* **}**

Here, *#member* is the signature of the target method (found within the same class), including the method name and the parameter types. *Label* is optional. It is the text that will be displayed in the document. If the label is skipped, the result document will display the member name. For example:

```
* Refer to the {@link #addAmtCoffee(int) addAmtCoffee(int)} method for modifying inventory levels.
```

This generates a link as:

Refer to the addAmtCoffee(int) method for modifying inventory levels.

{@link *package.class#member label* **}**

Here, *package.class* is the full name of the class where the method resides. It must include the package name if the class is not in the same package. *#member* is the field or method signature where this comment links to. *Label* is optional. It is the text that will be displayed in the document. If the label is skipped, the result document will display the member name. For example:

```
* Refer to the {@link Coffee#getNumCoffeePerUnit() getNumCoffeePerUnit()} method for coffee unit sizes.
```

This generates a link as:

Refer to the `getNumCoffeePerUnit()` method for coffee unit sizes.

Javadoc

A Sample Java Class "BubbleSorter" – With Full Javadoc Comment Notation

```
package samples;

import java.util.ArrayList;
import java.util.Scanner;

/**
 * This program reads a list of up to 10 integers, sorts them in ascending
 * order, and prints them out.  It uses the bubble sort algorithm.
 * <br><br>
 * Input: The integers to be sorted.  These are read from the standard input.
 * The program repeatedly prompts for an integer, then asks the user if
 * s/he is done.  This repeats until the user is done, or until the
 * maximum number of integers is reached.  If the user enters characters
 * that are not integer format, the program will ask him/her to try
 * again.
 * <br><br>
 * Output: The sorted list of integers, on the standard output.
 * <br><br>
 * Libraries Used: java.util ArrayList & Scanner classes.
 * <br><br>
 * Known Limitations: When the user answers the "Done?" question, any
 * character other than "N" or "n" is considered a yes.
 *
 * @author Harrison Jones
 * @author Hamid Al Shayeb
 * @since version 1.0
 * @see java.util.Scanner
 * @see java.util.ArrayList
 */
public class BubbleSorter
{

    /**
     * The maximum number of integers the program will accept ({@value})
     *
     * @since version 1.8
     */
    private static final int MAX_SIZE = 10;
```

Class/Interface Documentation with Tags

Field Documentation with Tags

Javadoc

Sample Java Class "BubbleSorter" - Using Javadoc Comment Notation

```
/**
 * The "private" constructor. This is defined as private because as
 * all methods & data attributes are static it does not make much sense
 * to allocate individual BubbleSorter instances. Making the constructor
 * private" eliminates the ability to allocate an instance.
 *
 * @since version 2.0
 */
private BubbleSorter() {}

/**
 * Prompts for and reads an ArrayList of Integers from the standard
 * input.  Repeatedly prompts for an integer, then asks the user if
 * s/he is done.  This repeats until the user is done, or until the
 * maximum number of integers is reached.  If the user enters characters
 * that are not integer format, the method will ask him/her to try
 * again. <br><br>
 * Limitation: If the user answers the "Done?" question with
 * any character but "N" or "n", it is considered a yes.
 *
 * @param inputArray the array into which the integers are read
 * @return the number of integers read
 * @throws DataValidationException if the ArrayList<Integer> parameter is null.
 */
private static int readArray(ArrayList<Integer> inputArray) throws DataValidationException
{
    // Throw DataValidationException if we get a null ArrayList - cannot use that!
    if (inputArray == null) throw new DataValidationException("null ArrayList encountered");

    int intCount = 0; // number of integers read so far
    String doneAnswer; // user's answer to the "Done?" question

    // Use the new java Scanner class to read user input from the keyboard.
    Scanner scanner = new Scanner(System.in);

    // Set doneAnswer to make sure the loop will execute once
    doneAnswer = "N";
```

Method/Constructor Documentation with Tags

Regular Comments

Javadoc

Sample Java Class "BubbleSorter" - Using Javadoc Comment Notation

```
        // Repeat this loop until the user says done OR until we get the maximum
        // number of integers.
        while (doneAnswer.toUpperCase().trim().equals("N") && intCount < MAX_SIZE)
        {
            System.out.print("Enter an integer: ");

            inputArray.add(scanner.nextInt());
            intCount++;

            // Test intCount here to avoid a "Done?" question if we've
            // already read the maximum number of integers.  In this case, we'll
            // be stopping anyway.
            if (intCount < MAX_SIZE)
            {
                System.out.print("Done? (Y/N): ");
                doneAnswer = scanner.next();
            } // end if
        } // end while

        return intCount;
    } // end inputArray

    /**
     * Sorts an ArrayList of Integers in ascending order using the
     * "bubble sort" algorithm.
     * <br><br>
     * Assumptions:
     *     1. Size is not negative
     *     2. The ArrayList has at least 'size' elements
     *
     * @param theArray the array to be sorted
     * @param size     the number of elements in the array to be sorted
     * @throws DataValidationException if the ArrayList<Integer> parameter is null.
     * @throws DataValidationException if the size parameter is > MAX_SIZE.
     */
```

Regular Comments

Method/Constructor Documentation with Tags

Javadoc

Sample Java Class "BubbleSorter" - Using Javadoc Comment Notation

```
private static void sortArray(ArrayList<Integer> theArray, int size) throws DataValidationException
{
    // Throw DataValidationException if we get a null ArrayList - cannot use that!
    if (theArray == null) throw new DataValidationException("null ArrayList encountered");

    // Throw DataValidationException if we get a size > MAX_SIZE - cannot use that!
    if (size > MAX_SIZE) throw new DataValidationException("size greater than allowed max.");

    // The bubble sort algorithm makes size-1 passes through the ArrayList.  In
    // each pass, it swaps adjacent pairs of elements if they are in the
    // wrong order.
    int pass;       // The number of the pass we're currently on
    int pairStart;  // The position of the first element of the pair being tested and possibly swapped
    int tempInt;    // A temporary integer used for swapping integers


    for (pass = 0; pass < size - 1; pass++)
    {
        for (pairStart = 0; pairStart < size - pass - 1; pairStart++)
        {
            // If this pair is out of order, swap them
            if (theArray.get(pairStart) > theArray.get(pairStart + 1))
            {
                tempInt = theArray.get(pairStart);
                theArray.set(pairStart, theArray.get(pairStart + 1));
                theArray.set((pairStart + 1), tempInt);
            } // end if
        } // end for pairStart
    } // end for pass
} // end sortArray
```

Regular Comments

Javadoc

Sample Java Class "BubbleSorter" - Using Javadoc Comment Notation

```
/**
 * Prints an ArrayList of Integers on the standard output, one per
 * line.
 * <p>Assumptions: size is not negative, and the ArrayList has at least 'size'
 * elements
 *
 * @param theArray the array to be printed
 * @param size     the number of elements in the array to be printed
 * @throws DataValidationException if the ArrayList<Integer> parameter is null.
 * @throws DataValidationException if the size parameter is > MAX_SIZE.
 */

private static void printArray(ArrayList<Integer> theArray, int size) throws DataValidationException
{
    // Throw DataValidationException if we get a null ArrayList - cannot use that!
    if (theArray == null)
         throw new DataValidationException("null ArrayList encountered");

    // Throw DataValidationException if we get a size > MAX_SIZE - cannot use that!
    if (size > MAX_SIZE)
         throw new DataValidationException("size greater than allowed max.");

    for (int i = 0; i < size; i++)
    {
        System.out.println(theArray.get(i));
    }

} // end
```

Method/Constructor Documentation with Tags

Javadoc

Sample Java Class "BubbleSorter" - Using Javadoc Comment Notation

```
    /**
     * The main method for the program.  Creates an array of integers,
     * then calls other methods to read, sort, and print them.
     *
     * @param args The command-line arguments.  Not used in this program,
     *             but required by Java
     */
    public static void main(String args[])
    {
      try
      {
        // The array that will hold the integers to be sorted
        ArrayList<Integer> sortingList = new ArrayList<Integer>();
        int arraySize; // the number of integers actually in the array

        // Read the array
        arraySize = readArray(sortingList);

        // Sort the array
        sortArray(sortingList, arraySize);

        // Print the sorted array
        System.out.println(); // skip a line
        System.out.println("Sorted Result:");
        printArray(sortingList, arraySize);
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    } // end main

} // end class Bubble
```

Method/Constructor Documentation with Tags

Regular Comments

Generating HTML Javadoc Documentation

To run *javadoc* to auto-generate *javadoc* documentation for your classes, you must execute the "javadoc" command.

This is actually a java application found in the standard JDK distribution (located in the JAVA_HOME "bin" directory). Assuming you have this location in your environment "path", then simply invoking the "javadoc" executable with certain command-line flags will auto-generate the javadoc documentation for your classes.

Javadoc invocation:

```
javadoc <command-line-flags> <package/directory, java file, or list of java files>
```

Common Command-Line Javadoc Flags

```
-public                     Show only public classes and members
-protected                  Show protected/public classes and members (default)
-package                    Show package/protected/public classes and members
-private                    Show all classes and members
-author                     Include @author paragraphs
-classpath <pathlist>       Specify where to find user class files
-d <directory>              Destination directory for output files
```

Javadoc invocation examples:

```
C:\Example> javadoc -d docs -classpath . -author -private *.java
C:\Example> javadoc -protected Vehicle.java Radio.java
C:\Example> javadoc -d documents -author -public MyProj
```

Generating HTML Javadoc Documentation (cont.)

Example: Executing javadoc on the sample BubbleSorter class and related exception class (source code is located in a package/folder called “samples”):

```
C:\Example> javadoc -d docs -classpath . -author -private samples
Loading source files for package samples...
Constructing Javadoc information...
Standard Doclet version 1.5.0_08
Building tree for all the packages and classes...
Generating docs\samples/\BubbleSorter.html...
Generating docs\samples/\DataValidationException.html...
Generating docs\samples/\package-frame.html...
Generating docs\samples/\package-summary.html...
Generating docs\samples/\package-tree.html...
Generating docs\constant-values.html...
Generating docs\serialized-form.html...
Building index for all the packages and classes...
Generating docs\overview-tree.html...
Generating docs\index-all.html...
Generating docs\deprecated-list.html...
Building index for all classes...
Generating docs\allclasses-frame.html...
Generating docs\allclasses-noframe.html...
Generating docs\index.html...
Generating docs\help-doc.html...
Generating docs\stylesheet.css...

C:\Example>
```

Resulting Javadoc Documentation – Sample BubbleSorter Class from Earlier

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** | **FRAMES** **NO FRAMES** **All Classes**
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Class BubbleSorter

```
java.lang.Object
   └ BubbleSorter
```

```
public class BubbleSorter
extends java.lang.Object
```

This program reads a list of up to 10 integers, sorts them in ascending order, and prints them out. It uses the bubble sort algorithm.

Input: The integers to be sorted. These are read from the standard input. The program repeatedly prompts for an integer, then asks the user if s/he is done. This repeats until the user is done, or until the maximum number of integers is reached. If the user enters characters that are not integer format, the program will ask him/her to try again.

Output: The sorted list of integers, on the standard output.

Libraries Used: java.util ArrayList & Scanner classes.

Known Limitations: When the user answers the "Done?" question, any character other than "N" or "n" is considered a yes.

Since:
version 1.0
Author:
Harrison Jones, Hamid Al Shayeb

Javadoc

See Also:

Scanner, ArrayList

Field Summary

private static int	**MAX_SIZE** The maximum number of integers the program will accept (10)

Constructor Summary

private	**BubbleSorter**() The "private" constructor.

Method Summary

static void	**main**(java.lang.String[] args) The main method for the program.
private static void	**printArray**(java.util.ArrayList<java.lang.Integer> theArray, int size) Prints an ArrayList of Integers on the standard output, one per line.
private static int	**readArray**(java.util.ArrayList<java.lang.Integer> inputArray) Prompts for and reads an ArrayList of Integers from the standard input.
private static void	**sortArray**(java.util.ArrayList<java.lang.Integer> theArray, int size) Sorts an ArrayList of Integers in ascending order using the "bubble sort" algorithm.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

MAX_SIZE

```
private static final int MAX_SIZE
```

The maximum number of integers the program will accept (10)

Since:
version 1.8
See Also:
Constant Field Values

Constructor Detail

BubbleSorter

```
private BubbleSorter()
```

The "private" constructor. This is defined as private because as all methods & data attributes are static it does not make much sense to allocate individual BubbleSorter instances. Making the constructor private" eliminates the ability to allocate an instance.

Since:
version 2.0

Method Detail

readArray

```
private static int readArray(java.util.ArrayList<java.lang.Integer> inputArray)
                      throws DataValidationException
```

Prompts for and reads an ArrayList of Integers from the standard input. Repeatedly prompts for an integer, then asks the user if s/he is done. This repeats until the user is done, or until the maximum number of integers is reached. If the user enters characters that are not integer format, the method will ask him/her to try again.

Limitation: If the user answers the "Done?" question with any character but "N" or "n", it is considered a yes.

Parameters:

inputArray - the array into which the integers are read

Returns:

the number of integers read

Throws:

DataValidationException - if the ArrayList parameter is null.

---

sortArray

```
private static void sortArray(java.util.ArrayList<java.lang.Integer> theArray,
                              int size)
                       throws DataValidationException
```

Sorts an ArrayList of Integers in ascending order using the "bubble sort" algorithm.

Assumptions:

1. Size is not negative
2. The ArrayList has at least 'size' elements

Parameters:

theArray - the array to be sorted

size - the number of elements in the array to be sorted

Throws:

DataValidationException - if the ArrayList parameter is null.

DataValidationException - if the size parameter is > MAX_SIZE.

---

printArray

```
private static void printArray(java.util.ArrayList<java.lang.Integer> theArray,
                               int size)
                        throws DataValidationException
```

Prints an ArrayList of Integers on the standard output, one per line.

Assumptions: size is not negative, and the ArrayList has at least 'size' elements

Parameters:

theArray - the array to be printed

size - the number of elements in the array to be printed

Throws:

DataValidationException - if the ArrayList parameter is null.

DataValidationException - if the size parameter is > MAX_SIZE.

main

```
public static void main(java.lang.String[] args)
```

The main method for the program. Creates an array of integers, then calls other methods to read, sort, and print them.

Parameters:

args - The command-line arguments. Not used in this program, but required by Java

Package **Class** **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** | **FRAMES** **NO FRAMES** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD