Introduction to Javadoc

1 Why Javadoc?

One of the key design goals of Java is the ability to cleanly separate interface from implementation. You should be able to use code by looking at its interface, without having to read the source code.

To be able to do this in practice, the somebody needs to provide high-quality documentation of the interface to its users (that is, the programmers who write code that uses it). This documentation needs to be:

- comprehensive
- consistent with the actual behaviour system.
- clearly and consistently presented.

The traditional way to do this was to hire technical writers to produce binders full of interface documentation. Professional tech writers produce well-written and presented documentation, but the resulting documentation is very expensive to write and was often incomplete or out of date.

Meanwhile, the implementers of interfaces documented their code to assist themselves and future maintainers. Maintainers need all the information in the interface documentation binders as well as material to explain complex bits of the implementation. So the same information often had to be written twice.

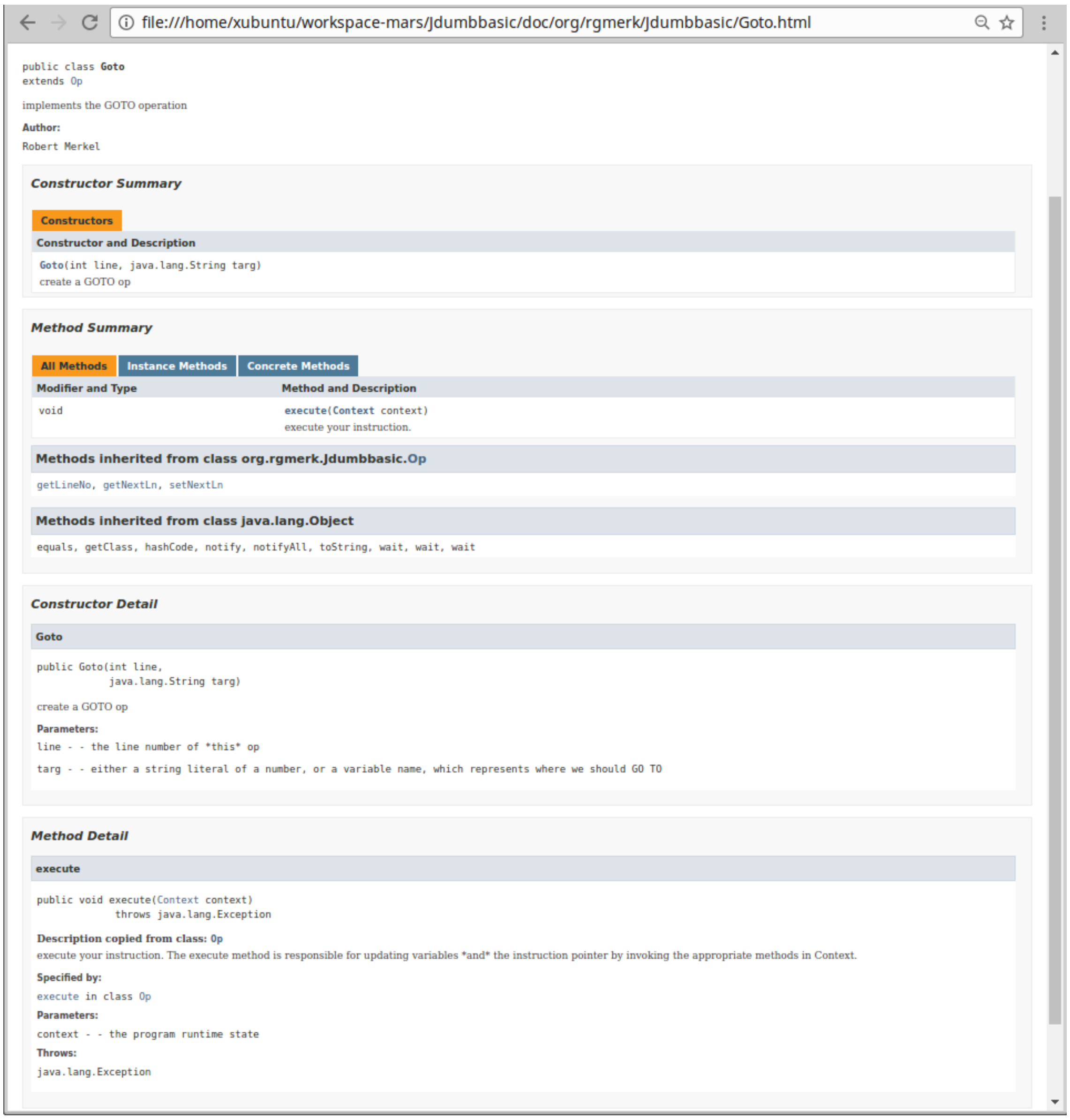
Where separate interface documentation is unavailable, interface users had to read the source code and its associated comments. As well as being time consuming (and sometimes impossible if the source code is not available) it can lead interface users to depend on implementation details that maintainers wish to change in future. This is a bad thing.

Javadoc helps to solve these problems. If you structure the comments in your code correctly, you can automatically export good-looking, consistently formatted interface user documentation in a variety of formats based directly on those comments and the structure of the Java source itself.

For instance, Figure 1 shows some commented source code, and Figure 2 shows the resulting, easy-to-read and professional looking HTML documentation.

```java
 package org.rgmerk.Jdumbbasic;

/**
 * implements the GOTO operation
 * @author Robert Merkel <robert.merkel@monash.edu>
 *
 */
public class Goto extends Op {

  private String target; // either a string representing a numeric constant, or a variable name
  /**
  * create a GOTO op
  * @param line - the line number of *this* op
  * @param targ - either a string literal of a number, or a variable name, \
  which represents where we should GO TO
  */

  public Goto(int line, String targ) {
    super(line);
    target=targ;
    return;
  }

  @Override
  public void execute(Context context) throws Exception {
    int newline = context.getValue(target);
    context.setIp(newline);
    return;
  }
}
```

Figure 1: A Java class with Javadoc comments



file:///home/xubuntu/workspace-mars/Jdumbbasic/doc/org/rgmerk/Jdumbbasic/Goto.html

```
public class Goto
extends Op
```

implements the GOTO operation

Author:

Robert Merkel

Constructor Summary

Constructors

Constructor and Description
`Goto(int line, java.lang.String targ)` create a GOTO op

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
void	`execute(Context context)` execute your instruction.

Methods inherited from class org.rgmerk.Jdumbbasic.Op

`getLineNo, getNextLn, setNextLn`

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Goto

```
public Goto(int line,
            java.lang.String targ)
```

create a GOTO op

Parameters:

`line - - the line number of *this* op`

`targ - - either a string literal of a number, or a variable name, which represents where we should GO TO`

Method Detail

execute

```
public void execute(Context context)
             throws java.lang.Exception
```

Description copied from class: Op

execute your instruction. The execute method is responsible for updating variables *and* the instruction pointer by invoking the appropriate methods in Context.

Specified by:

`execute in class Op`

Parameters:

`context - - the program runtime state`

Throws:

`java.lang.Exception`

Figure 2: HTML documentation generated from Javadoc comments

In this guide, we'll explain how to format your comments so that Javadoc tools can read and process them. However, neither Javadoc itself nor this guide explains what you should put in your comments! You should read ”How to comment your code” for more information on that.

2 Adding Javadoc comments

2.1 What Java elements can have a Javadoc comment?

Javadoc comments can be attached to:

- A Java package[1]
- A Java class.
- A Java interface[2]
- Any public class member, including attributes and methods (and constructors).

Private methods and attributes are ignored by Javadoc - they are part of the implementation, not the interface, and any comments left there are for class maintainers, not interface users.

2.2 Making a comment a Javadoc comment

Making a comment visible to Javadoc is easy.

Just place your comment directly *above* the element it applies to. Use *block comments*, and add an extra * directly after the initial /*, as you will see in the Java source code in Figure 3:

```
/**
 * This is a Javadoc comment for method "foo".  It should explain what "foo" does
 */

public void foo(int fred) {
  ...
}

/*
 * This is NOT a Javadoc comment for method "bar" as it lacks the /** at the start:
 */
public int bar(String ginger) {
   ...
 }

 // This is also not a Javadoc comment for method "baz"
 public int baz() {
    /** This is not a Javadoc comment either as it appears within a method*/
    int burble = y + bazhelper();
    // and nor is this
    return burble;
 }

 /**
  * this is NOT a Javadoc comment because bazhelper is a private method
  */

 private int bazhelper() {
 ...
 }
```

Figure 3: Examples of Javadoc and not-Javadoc comments

2.3 Tags

For each Java element that Javadoc can be used to document, there are some types of information that you'll typically want to record. For instance, for a Java method, you'll want to document things like what the return value is, and what each of the parameters should contain.

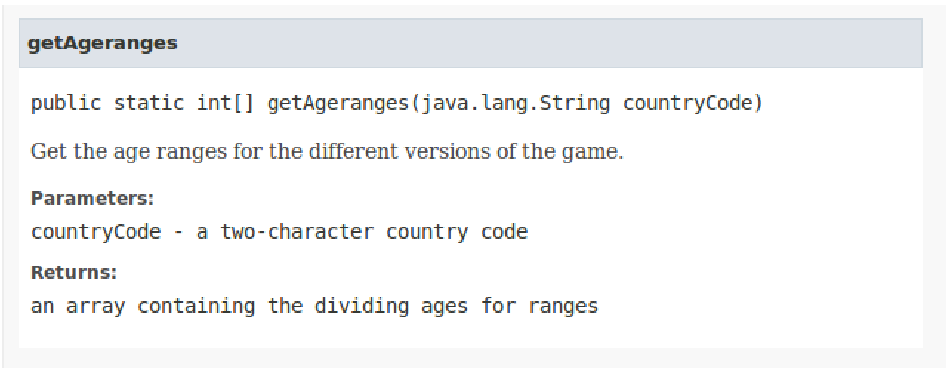
[1]We will discuss packages later in FIT2099. In short, they are a collection of classes that perform related functions. They are used to simplify organization of large software systems

[2]If you're not familiar with Interfaces, they are something like a completely abstract class.

Javadoc allows you to tag parts of your comments so that the processor can identify and format them appropriately. For instance, for a method, you will usually want to say something about what the input parameters are and what the method returns. You can use the `@param` and `@return` tags for this purpose. For instance, Figure 4 shows Javadoc comments for a method. The Javadoc processor, with the tags giving structure to the information in the comments, produces the output shown in Figure 5.

```
/**
 * Get the age ranges for the different versions of
 * the game.
 *
 * @param countryCode a two-character country code
 * @return an array containing the dividing ages for ranges
 */
public static int [] getAgeranges(String countryCode) {
   int [] agerange= new int[2];
    if (countryCode.equals("US")) {
      agerange[0]=8;
      agerange[1]=15;
    } else {
      agerange[0]=7;
      agerange[1]=14;
    }

    return agerange;
}
```

Figure 4: A Javadoc comment with tags



getAgeranges

```
public static int[] getAgeranges(java.lang.String countryCode)
```

Get the age ranges for the different versions of the game.

Parameters:

`countryCode` - a two-character country code

Returns:

an array containing the dividing ages for ranges

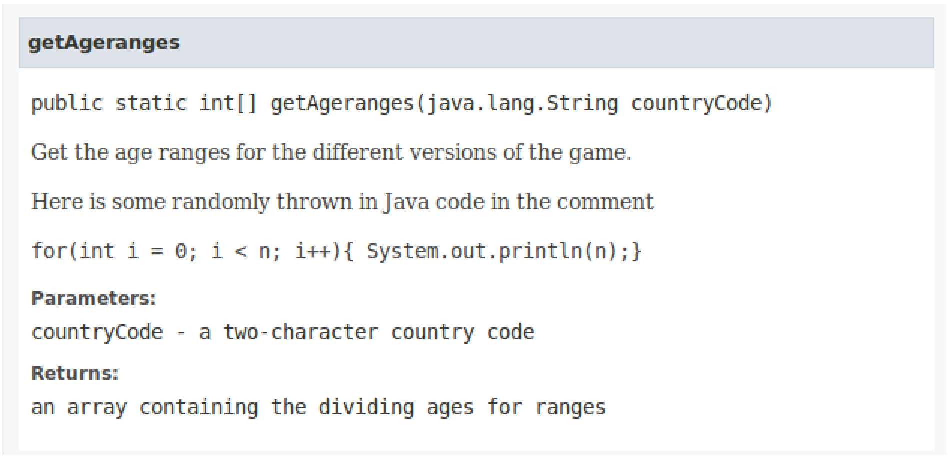
Figure 5: HTML output for Javadoc comment with tags

.

You can include some HTML formatting in Javadoc comments. To make Javadoc comments readable as plain text in a source file and as HTML in generated documentation, Javadoc includes some inline tags, such as the `@code` tag (Figure 6), which produces the output in Figure 7.

```
/**
 * Get the age ranges for the different versions of
 * the game.
 * <P>
 * Here is some randomly thrown in Java code in the comment
 * <p>
 * {@code for(int i = 0; i < n; i++){ System.out.println(n);}}
 *
 * @param country_code a two-character country code
 * @return an array containing the dividing ages for ranges
 */

public static int [] getAgeranges(String country_code) {
```

Figure 6: Using inline tags



getAgeranges

```
public static int[] getAgeranges(java.lang.String countryCode)
```

Get the age ranges for the different versions of the game.

Here is some randomly thrown in Java code in the comment

`for(int i = 0; i < n; i++){ System.out.println(n);}`

Parameters:

`countryCode - a two-character country code`

Returns:

`an array containing the dividing ages for ranges`

Figure 7: Output using inline tags

The full list of supported tags is available in the official Java documentation[3], but here is a list of some of the most commonly used Javadoc tags:

[3] `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html`

Tag	**Elements commonly used with**	**What it is used to tag**
`{@author` *author string*`}`	class, interface, package	The writer/maintainer of this code.
`{@param` *parameter description*`}`	method	parameter to a method
`{@return` *return description*`}`	method	return value of a method.
`{@version` *version string*`}`	class, package	version of this element
`{@see` *searchstring*`}`	*any*	Cross-reference to another Javadoc comment, an external link, or some other external reference See Section 2.5
`{@deprecated` *explanation*`}`	*any*	used when this element is *deprecated* - that is, it should not be used in new code. Text should explain what to use instead.
`{@throws` *exception-name explanation*`}`	method	describes an exception that this method might throw, and the reasons why it might do so
`{@code code-text}`	*inline*	write code verbatim within a Javadoc comment. Will be formatted as code in output
`{@literal literal-text}`	*inline*	write text including HTML literals (eg <, >) as it appears in the comment

2.4 Package Javadoc

If you're not familiar with Java packages, you can safely skip this section for now.

In Java, collections of related classes can be organized into packages. They have a hierarchical structure like nested folders on a disk, and, indeed, Eclipse and most other IDEs store all the classes in a package in the same folder.

To document Java packages in Javadoc, you should edit the `package-info.java` file. Eclipse, and most other IDEs, will create this file for you when you create a new package. A package-info.java file contains only one non-comment item, a declaration containing the full name of the package. If your IDE created it for you, leave it alone.

Above that, you can write a Javadoc comment describing the package:

```java
/**
* A package to determine game versions for international markets.
*
* @author Robert Merkel <robert.merkel@monash.edu>
* @version 1.0
*/
package org.rgmerk.versioner;
```

2.5 Referencing

Javadoc has extensive facilities for providing references to other documentation, including:

- References to Javadoc documentation for other elements within your codebase
- URLs
- "Offline" sources such as books [4]

However, the syntax for this referencing can get a little complex. It's all extensively documentated in the official Javadoc documentation, so look there!

2.6 Custom Tags

Sometimes, you may want to add your own tags for a particular Javadoc document. For instance, say you're writing a library for handling graphs, and you want to indicate the algorithmic complexity of each method. In Javadoc, you can define your own `@complexity` block tag for this and instruct the Javadoc doclet how to process it into HTML. See the section on the `-tag` argument in the Javadoc documentation [5].

3 Producing Javadoc documentation

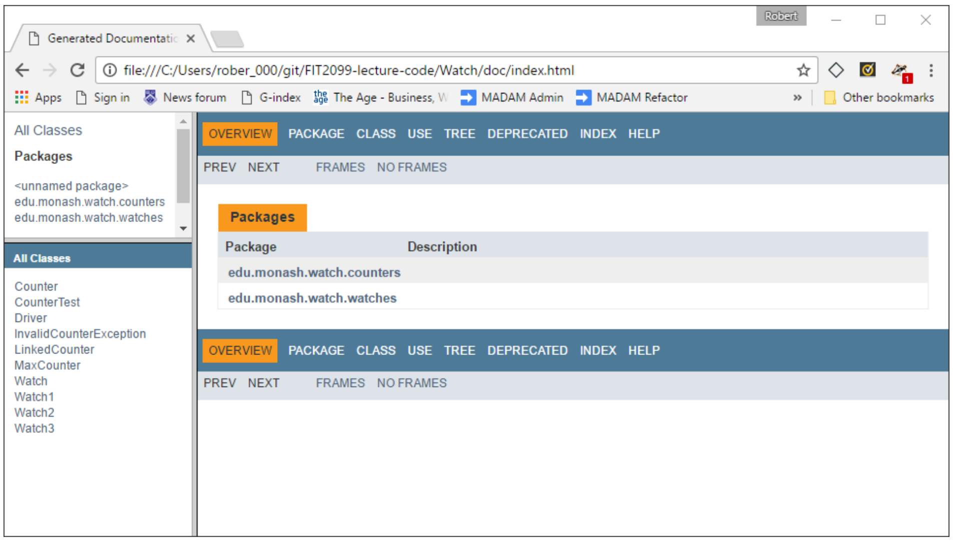
Tools that read Javadoc documentation and produce output in other formats are known as *doclets*, and there are a number of different ones available. However, the standard one that comes with Java reads Javadoc and produces a collection of linked HTML files, which is adequate for beginners!

To produce Javadoc documentation from an Eclipse project, select *Project → Generate Javadoc....* Choose the project you wish to generate Javadoc for. By default, the generated documentation will be placed in a doc subfolder of the project.

You can view the generated documentation from within Eclipse itself or from a web browser. Open `index.html` for an index of all the generated documentation.

[4] If you are unfamiliar with "books", please consult `https://en.wikipedia.org/wiki/Book` for a brief introduction :-P

[5] `http://docs.oracle.com/javase/1.5.0/docs/tooldocs/windows/javadoc.html#tag`



Package	Description
edu.monash.watch.counters	
edu.monash.watch.watches	

Figure 8: An example index page for generated documentation