14. Javadoc

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

Documentation is a not very exciting but critical part of programming. We need to write down what we've done (and how) so that both we and others can understand it.

- **Programs and code have a long life** – it's rare for code to be owned only by one person, or even by one group of people, over it's whole life.
- **We forget how our own code works** – it's not just other people: given a few weeks we need to relearn how things we've written work.

Kinds of Documentation

We can distinguish between several types of documentation.

- **How it works** – this is documentation that explains how our code works. It's for people who work on maintaining it and developing it.
- **Version History** – this is a list of all the changes that were made to code from one version, or release, to the next. It's meant for both users and developers.
- **What it does** – this is documentation mainly meant for those who are using the code and don't need to know how it works.

Different Types, Different Approaches

There are several ways we can do each of these – but we'll just look at one or two in each case.

How it Works – Good Code + Standard Commenting

In order to explain how your code works you should write and lay it out in a clear way, put readability above showing off, and choose your names carefully. You should also comment your code. There are no strict rules to this but:

- Comment each variable declaration if it's not obvious
- Comment each logical block of code – for example, each method (though see below).
- Comment anything tricky – if you found it hard to write, or can see that it's not obvious how it works, then comment it.

Self-Documenting Code

Occasionally (though not so much anymore) you get statements like:

"I don't need to comment my code – it's self-documenting"

By this, the programmer means that they believe they are so good at writing clear code that no further commenting is necessary. This is, as well as being

arrogant, an extreme position to take – it's true good, well-laid out designed code with good identifier names doesn't need as much commenting. But it's a fallacy to claim it never does. So don't do it:-)

Version History – Version Control System (and Javadoc)

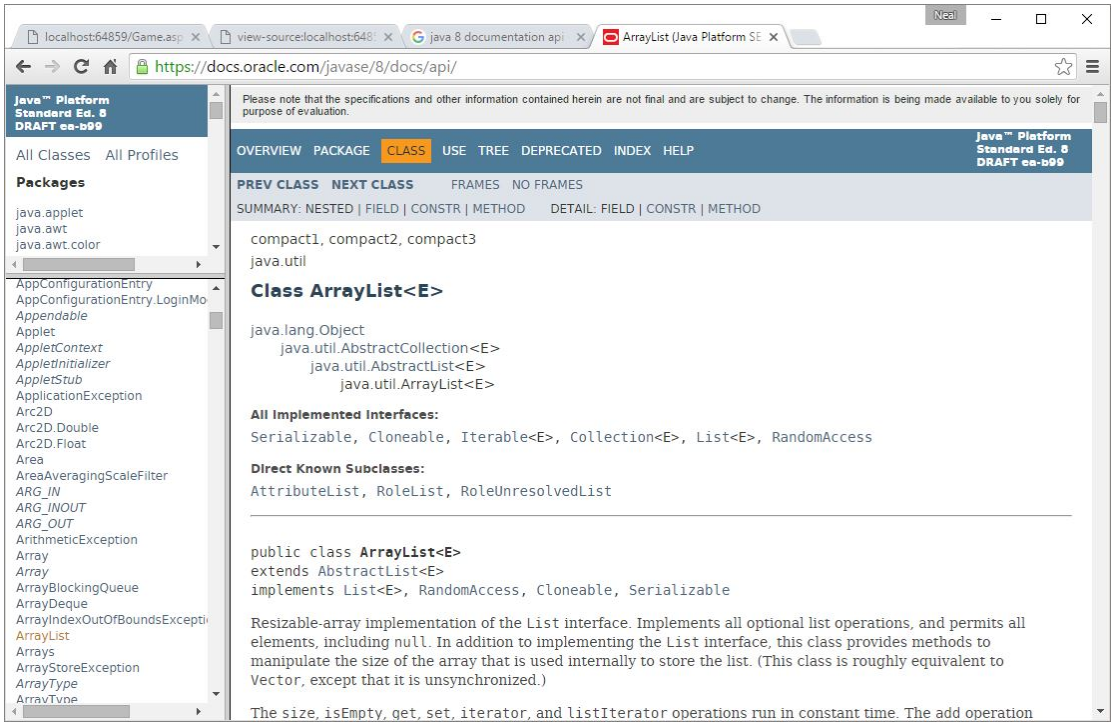
In a 'real' application (or, say, a 2nd year software engineering project; or 3rd/4th year project) you are likely to be using a *source code control system*: a piece of software that maintains a history of your code (including the ability to roll back to previous versions), and also – usually – includes tools so multiple programmers can collaborate on applications. Systems like this include *git* and *subversion* (and there are others). These systems also include tools that let you enter version information and, often, maintain version numbering systems (so you don't have to). It's also possible to use the tools we're going to talk about next to do this.

What it Does – Javadoc

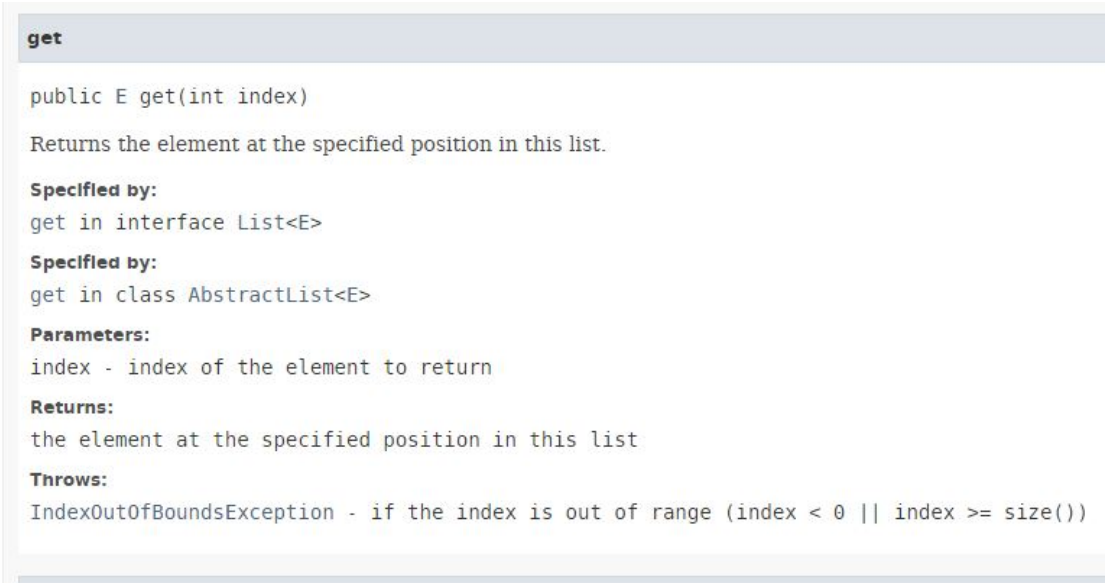
The traditional way of doing 'what it does' documentation is to write separate documents – but this is not good for three reasons:

- **Wasteful** – at least some of the things you need to write in these documents are already in the code.
- **Goes out of date** – if you change the code you need to change the documentation and you'll often forget.
- **Extra documents to maintain** – as well as the code, you now have more documents to deal with.

If you've ever looked up Java library documentation, and we've done this in lectures, you will have seen things like this:



This is the start of the documentation for the `ArrayList` class, which we've used a fair bit. If you scroll down a bit you get to:



get

public E get(int index)

Returns the element at the specified position in this list.

Specified by:
get in interface List<E>

Specified by:
get in class AbstractList<E>

Parameters:
index - index of the element to return

Returns:
the element at the specified position in this list

Throws:
IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

which is the documentation for one of the `get()` methods. Clearly, someone has been to a lot of trouble to write all this – and they have, but not as much as you think, because it's generated from the code with a few 'special' comments.

If you've been using Netbeans, whenever you created a new class, you would see stuff like this at the top of the code:

```
/**
 *
 * @author neal
 */
public class JavaApplication13 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

First it just looks like 'blank' or 'template' comments – but you can see the text starting with @. If you look closer, you can see that instead of the comment starting `/*` they start `/**` - these are *javadoc* comments which are processed by a special tool called `javadoc`, which is included in the Java system when you install it. They way it works is that it processes the special comments, using the words marked with @ to highlight and extract certain information, together with information from the source code, to produce

exactly the documentation shown in the figures above. All the Java library documentation is done this way – by javadoc comments embedded in the source code.

- **Less wasteful** – as we'll see information that can be extracted from the source code is: we don't have to write it out again.
- **More likely to update** – it doesn't force you to update it, but you are more likely to if you are looking at it at the same time you're editing the code
- **No extra documents** – obviously, as it's in the code.

Public Things Only

Javadoc only applies to public parts of the code (also actually protected etc – but not private). It's only meant to document what the code *does*, not *how*.

Javadoc Example

Here's an example series of Java code files with javadoc comments using the enumerated type card classes. First the suite enumerated type:

```
/**
 * The Suite enumerated type represents the suites
 * in a deck of cards.
 */
public enum Suite {
     /* This is an ordinary comment - we've commented
     each suite with its own javadoc entry. In real life
     for something this simple, that may be excessive */

      /** represents the Spades suite */
     SPADES,
     /** represents the Clubs suite */
     CLUBS,
     /** represents the Hearts suite */
     HEARTS,
      /** represents the Diamonds suite */
     DIAMONDS
}
```

Note the 'ordinary' comment we've left in. This will produce documentation that looks like this:

Enum Suite

```
java.lang.Object
  └ java.lang.Enum<Suite>
      └ Suite
```

All Implemented Interfaces:
java.io.Serializable, java.lang.Comparable<Suite>

```
public enum Suite
extends java.lang.Enum<Suite>
```

The Suite enumerated type represents the suites in a deck of cards.

Enum Constant Summary
CLUBS represents the Clubs suite
DIAMONDS represents the Diamonds suite
HEARTS represents the Hearts suite
SPADES represents the Spades suite

Method Summary	
static Suite	valueOf(java.lang.String name) Returns the enum constant of this type with the specified name.
static Suite[]	values() Returns an array containing the constants of this enum type, in the

Notice it includes all the javadoc comment text we wrote, but we didn't have to repeat the names of the enumerated types, or the name of the type itself (also note it documents the 'free' methods we didn't actually write but which are present).

Here's the `CardName` type – we've cut some out to shorten it and but the *annotations* in bold:

```
/**
 * The CardName enumerated type represents the names
 * and values of cards in a deck.
 */

public enum CardName {
      ACE(1),     // This time we'll leave out individual comments
      TWO(2),     // for each element of the eum
      ….
      KING(10);

      //We can have an ordinary comment to explain what this
      //is but *not* a javadoc one - it's a *private* field
      //and nobody *using* the class needs to know about it
      private final int value;

      // Same again - the constructor isn't needed outside
      // the enum, so no need for a javadoc comment
      CardName(int cardValue){
            value = cardValue;
      }

      //The method below on the other hand, *is* called
      //from outside the class

      /**
       * Returns the value associated with a particular card.
       * @return the value associated with a card
       * @deprecated
       * Use <code>cardScore</code> instead.
       * <p>
       * The name of method is too close to <code>values</code>
       * which returns an array containing the <code>enum</code
       * objects.
       */
      @Deprecated //What's this??
      public int value() {
            return cardScore();
      }

      //Notice we can put HTML tags like <p> (and any others we want)
      //in to format the comment. We use the <code> tag to format
      //code correctly.

      /**
       * Returns the value associated with a card.
       * @return the value associated with a card
       *<p>
       * Use in preference to <code>value</code>.
       * @see #value()
       */
      public int cardScore() {
            return value;
      }
}
```

The method part looks like this:

values

```
public static CardName[] values()
```

Returns an array containing the constants of this enum type, in the order they are declar

```
for (CardName c : CardName.values())
    System.out.println(c);
```

Returns:

an array containing the constants of this enum type, in the order they are declared

---

valueOf

```
public static CardName valueOf(java.lang.String name)
```

Returns the enum constant of this type with the specified name. The string must match e
(Extraneous whitespace characters are not permitted.)

Parameters:

`name` - the name of the enum constant to be returned.

Returns:

the enum constant with the specified name

Throws:

`java.lang.IllegalArgumentException` - if this enum type has no constant with the
`java.lang.NullPointerException` - if the argument is null

If you look you can see that the words starting with @ mark sections in the documentation. We're going to talk about these in a bit more detail below – but we'll mention one here: `@Deprecated` – this means that this public method is no longer encouraged; that it's been replaced by something else (in this case by `cardScore()` – and that `cardScore()`'s javadoc comment has a reference to the one it's replacing using `@see`). We *deprecate* methods when we have thought of a better way to do something that's public – we can't just take it out because people will be relying on it. So we mark it deprecated, don't update it, but leave it in long enough for people to update their code (usually years).

HTML in Javadoc

If you're familiar with HTML, you can see that our comments contain HTML elements – for example, `<p>` to mark a new paragraph, `<code>` to mark text to be rendered in a fixed-width 'courier' type font (code is usually represented in this way). You can use any HTML elements you want in javadoc.

Example Javadoc Class

The two examples above are enumerated types – but more typically we would write a class. So here's the `Card` class.

```
/**
```

```
 * This class represents a card using the {@link Suite} and
 * {@link CardName} enumerated types.
 * <p>
 * With appropriate definitions of <code>Suite</code>
 * and <code>CardName</code>,this class should be suitable
 * for any kind of card that can be characterized by a suite,
 * a name and a numeric value.
 */

public class Card {

	private final Suite suite;
	private final CardName name;

	/**
	 * Creates an immutable Card object.
	 * @param suiteVal the card's suite
	 * @param cardVal the card's name
	 */
	public Card(Suite suiteVal, CardName cardVal){
		name = cardVal;
		suite = suiteVal;
	}

	/**
	 * The suite of the Card object represented by the
	 * Suite enumerated type.
	 *
	 * @return the value of the Suite
	 */
	public Suite getSuite() {
		return suite;
	}

	/**
	 * The name of the Card object represented by the CardName
	 * enumerated type.
	 *
	 * @return the name of the Suite
	 */
	public CardName getName() {
		return name;
	}

	/**
	 * The numeric value of the Card object represented by the
	 * value within the appropriate CardName enumerated type
	 * object.
	 *
	 * @return the value of the Suite
	 */
	public int getValue() {
		return name.cardScore();
	}
```

```
	/**
	 * The Card object as a String formatted in the traditional
	 * way.
	 *
	 * @return the Card as a String
	 */
	public String toString() {
		return (name + " of " + suite);
	}
}
```

As you can see, there are annotations for parameters (`@param`) and return values (`@return`) – there is also `@link`, which puts in a hyperlink to the class (or enum etc.) referred to.

- `@return` – generates the text Returns: followed by the text we wrote in the comment.
- `@param` – generates the name of the parameter followed by the text we wrote (notice we don't have to put the parameter name in the comment; javadoc gets that all from the source code along with the names and types of all the methods, constructors etc.
- `@link` – generates a hyperlink. So `{@link Suite}` generates a clickable link to the documentation for Suite.

There are many other annotations we can use – for example, in the Netbeans code at the top of the page we used @author for the code's author – but these are the main ones.

Generating Javadoc

To generate javadoc, you just use the javadoc tool – just type

```
javadoc codefile.java
```

where codefile.java is the name of the file you want to generate documentation for. If your code includes references to other Java files (e.g. creates objects of that class etc.) then those files will also have javadoc generated. So, for example, in this case I typed:

```
javadoc Deck.java
```

and because `Deck.java` refers to `Card.java`, `CardName.java` and `Suite.java`, the documentation for them gets automatically generated too. The javadoc output is basically HTML with a few other things (a short file of javascript and a cascading style sheet) to manage the layout and formatting. Typically, you would make it available on a *web server* for users of the code.