OBJECT ORIENTATION FUNDAMENTALS IN JAVA – PART 4

Debugging, Assertions, and Exceptions

At present our system "works", but it is actually quite fragile. It is easy to cause a run-time error.

Question: What happens if someone tries to create a `Watch` for which one of the counters has a negative maximum value?

Let's try it. We'll change code that adds the third watch in `Driver` to

```
        watches.add(new Watch(new int[] {24, 60, -60, 1000} ));
```

The system compiles and runs. Everything goes well until it starts to test the third watch. At that point it crashes, with the following output:

```
03:19
##############################
Testing Watch: Watch
00:00Exception in thread "main" java.util.UnknownFormatConversionException: Conversion =
'N'
    at java.util.Formatter$FormatSpecifier.conversion(Formatter.java:2691)
    at java.util.Formatter$FormatSpecifier.<init>(Formatter.java:2720)
    at java.util.Formatter.parse(Formatter.java:2560)
    at java.util.Formatter.format(Formatter.java:2501)
    at java.util.Formatter.format(Formatter.java:2455)
    at java.lang.String.format(String.java:2940)
    at edu.monash.fit2099.counters.MaxCounter.toString(MaxCounter.java:29)
    at java.lang.String.valueOf(String.java:2994)
    at java.lang.StringBuilder.append(StringBuilder.java:131)
    at edu.monash.fit2099.watches.Watch.display(Watch.java:45)
    at edu.monash.fit2099.watches.Watch.testWatch(Watch.java:57)
    at Driver.main(Driver.java:18)
```

We see banner telling us we're testing an object of class `Watch`, and in fact we see the `00:00` of the first two counters. Then we see the message:

```
Exception in thread "main" java.util.UnknownFormatConversionException: Conversion = 'N'
```

This tells us that an *exception* has occurred.[37] It also tells us the *type* of the exception: `java.util.UnknownFormatConversionException`, and a message from that exception: `Conversion = 'N'`. This is followed by a *stack trace* (in red).[38] An exception indicates that something unexpected has occurred, which the part of the program it occurred in can't handle.

Debugging

Using the stack trace

Before we talk about exceptions in detail, let us consider all of the information we have just been given. The stack trace shows us what the program was executing when the exception occurred, and how it got there. In this case the method executing was

[37] https://docs.oracle.com/javase/tutorial/essential/exceptions/
[38] http://programming.guide/java/stack-trace.html

`java.util.Formatter$FormatSpecifier.conversion(…)`, at line 2691 of the library class `Formatter.java`.

This is not part of the code we have written. It is very unlikely that we have discovered a bug in the Java libraries: they have been used millions of times, and are very, very unlikely to be at fault. Suspecting the libraries should be an absolute last resort. This is an example of a common challenge in debugging: the system has failed at a point remote in the code from the error which is ultimately responsible for the failure. This makes it harder to find the bug. Soon we will discuss the "Fail Fast" principle that helps to address this issue.

The stack trace provides us with information that can help us find where the actual bug is. We can trace the path of execution back down the stack. We see that `java.util.Formatter$FormatSpecifier.conversion(…)` was called by `java.util.Formatter$FormatSpecifier.<init>(…)`, which was called by `java.util.Formatter.parse(…)`, and so on all the way back to `Driver.main(…)`.

The first method that we wrote that appears in the stack trace is `edu.monash.fit2099.counters.MaxCounter.toString()`. In Eclipse, if we click on the `MaxCounter.java:29` that appears between the parentheses, it will take us to the line of code that called the next method, `java.lang.String.format(…)`. This is the line

```
        return String.format(fieldFormat, this.getValue());
```

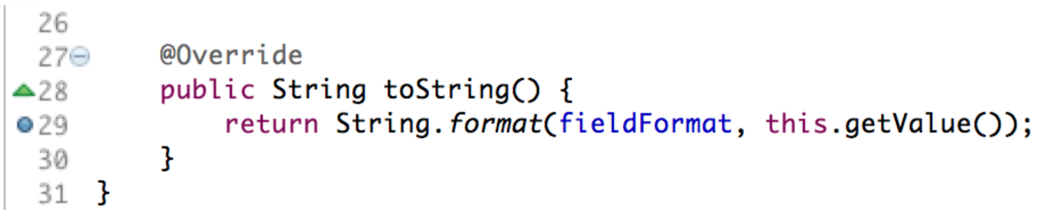
It seems likely that this line passed parameters into `String.`*`format`*`(…)` that eventually caused the crash. Before we investigate further, we will comment out the lines that create the `Watch1` and `Watch2` objects in `Driver`, as we know the problem is not there.

Using the Eclipse Debugger

At this stage, a common thing for beginner programmers to do would be to add some code before this line that prints the values of the arguments to `String.`*`format`*`(…)` to the console. In this case, we could certainly do that. There is, however, always a risk in doing this: every time you touch the code, you risk changing its behaviour, and/or introducing new bugs. In many real-world programming situations, moreover, there is often no console to print to.

A better solution is to use the programming environment's debugger. Almost all modern IDEs come with a built-in debugger that allows you to do things like set break points, execute code step-by-step, inspect the values of variables, and so on. It is very much worth your time learning to use your debugger.

In Eclipse, there in much debugger functionality available under the Run menu. You can also set a breakpoint by double-clicking to the left of the line number in the editor. Let's do that for the line of interest in `MaxCounter`.

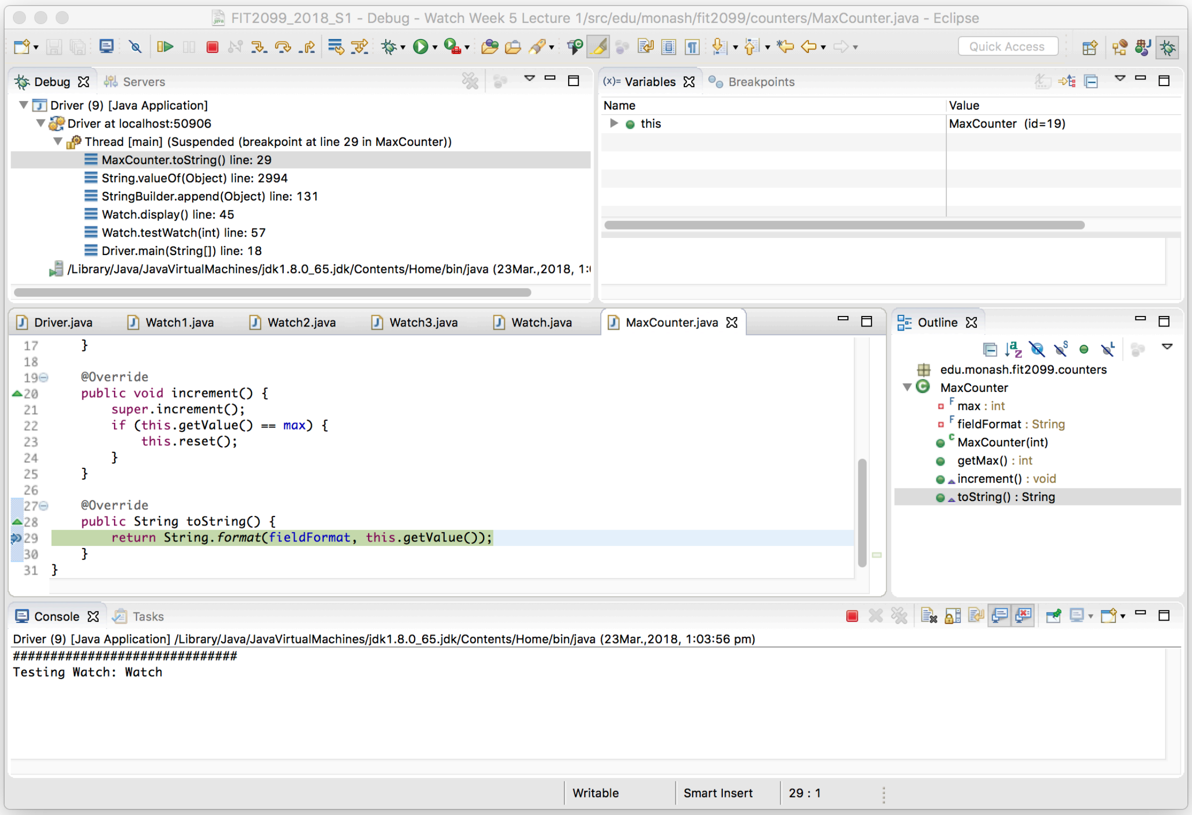


```
26
27⊖     @Override
28      public String toString() {
29          return String.format(fieldFormat, this.getValue());
30      }
31  }
```

The breakpoint is indicated by a blue circle in the left margin in the Eclipse editor.

We can then run the program in debug mode by selecting Debug from the Run menu. This will switch Eclipse to the Debug Perspective (you will have to enable this the first time you do it). The program will run until it reaches the breakpoint.

When the program stops, we see an Eclipse window like this



```
Driver (9) [Java Application]
  Driver at localhost:50906
    Thread [main] (Suspended (breakpoint at line 29 in MaxCounter))
      MaxCounter.toString() line: 29
      String.valueOf(Object) line: 2994
      StringBuilder.append(Object) line: 131
      Watch.display() line: 45
      Watch.testWatch(int) line: 57
      Driver.main(String[]) line: 18
  /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (23Mar.,2018, 1:
```

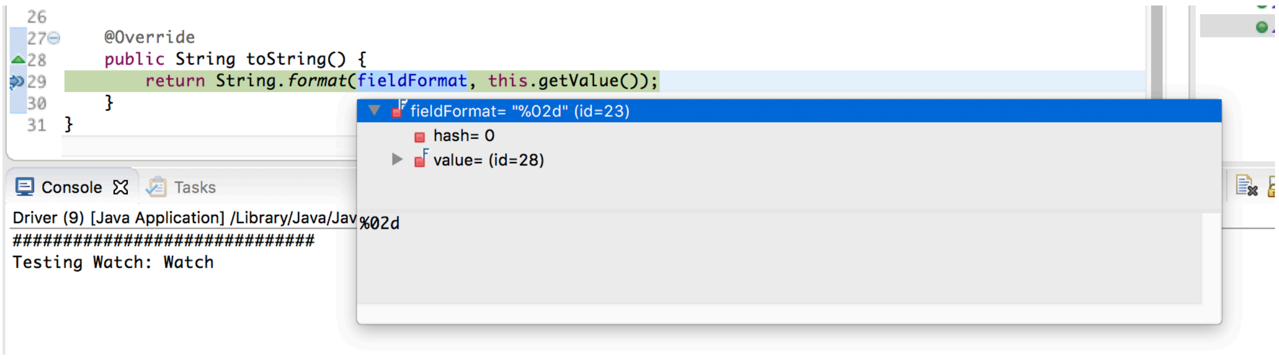
Name	Value
this	MaxCounter (id=19)

```java
    }

    @Override
    public void increment() {
        super.increment();
        if (this.getValue() == max) {
            this.reset();
        }
    }

    @Override
    public String toString() {
        return String.format(fieldFormat, this.getValue());
    }
}
```

```
edu.monash.fit2099.counters
  MaxCounter
    max : int
    fieldFormat : String
    MaxCounter(int)
    getMax() : int
    increment() : void
    toString() : String
```

```
Driver (9) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (23Mar.,2018, 1:03:56 pm)
##############################
Testing Watch: Watch
```

If we hover the pointer over any variable or attribute in the code, it will show us information about it. If we do this for `fieldFormat`, we see



```java
    @Override
    public String toString() {
        return String.format(fieldFormat, this.getValue());
    }
}
```

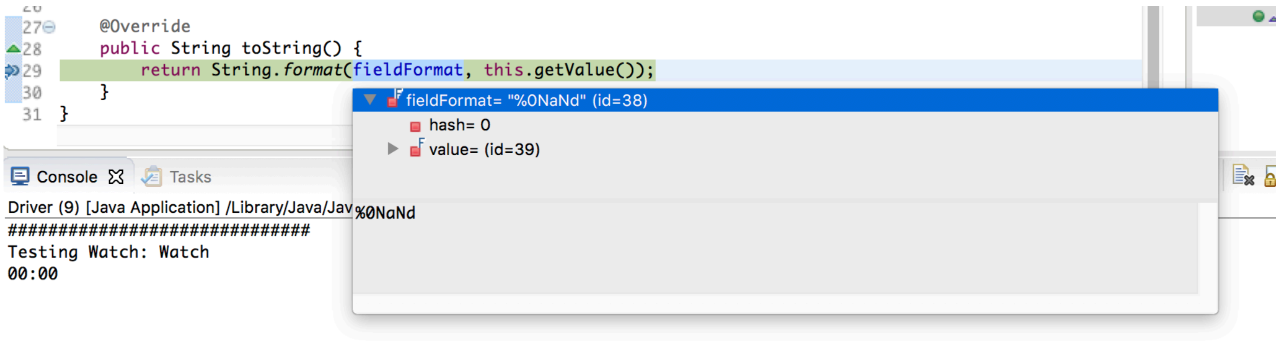
```
fieldFormat= "%02d" (id=23)
    hash= 0
    value= (id=28)
%02d
```

```
Driver (9) [Java Application] /Library/Java/Jav
##############################
Testing Watch: Watch
```

This shows us that `fieldFormat` has the value `%02d`, which is what we expect for the first counter in our `Watch`, which has a maximum value of `24`.

We now want to start the program again from where it stopped. We can do this by selecting Resume from the Run menu (or using the corresponding icon on the toolbar). When we do this, the program prints the `00` for the first counter, and then stops again at the breakpoint. We inspect `fieldFormat` again, and see that it has the value `%02d`, which is what we expect for the second counter in our `Watch`, which has a maximum value of `60`.

We resume again. The program prints prints the `:00` for the second counter, and then stops again at the breakpoint. We inspect `fieldFormat` again, and see



```
    @Override
    public String toString() {
        return String.format(fieldFormat, this.getValue());
    }
}
```

```
fieldFormat= "%0NaNd" (id=38)
    hash= 0
    value= (id=39)
%0NaNd
```

```
Driver (9) [Java Application] /Library/Java/Jav
##############################
Testing Watch: Watch
00:00
```

`fieldFormat` has the value `%0NaNd` — this is clearly wrong. What has happened?

First we must ask ourselves how `fieldFormat` got its value. We know it is set in the `MaxCounter` constructor, with the line

```
        fieldFormat = "%0" + String.format("%.0f", fieldWidth) + "d";
```

So, `String.format("%.0f", fieldWidth)` must have produced the string `NaN`. If we set a breakpoint at that line and debug the program again until we get to the creation of the third counter, we see that `fieldWidth` itself has the value `NaN`.

At this point, if you don't know already, you are wondering what `NaN` is. We could do a quick Google, or look at the Java documentation for the type `double`.[39] This would lead us to discover that the Java `double` is an IEEE 754 floating point value[40], and that `NaN` means "Not a Number".

`fieldWidth` is calculated in the line

```
        double fieldWidth = Math.ceil(Math.log10(max));
```

so the problem must have occurred there. Hovering over `max`, we see that it has the value `-60`. At this point I can use my mathematical knowledge to say "A-ha! You can't take the logarithm of a negative number. There is no number n for which 10^n is negative". If I didn't know that, however, I could look at the documentation for `Math.log10(…)` and see[41]

log10

`public static double log10(double a)`

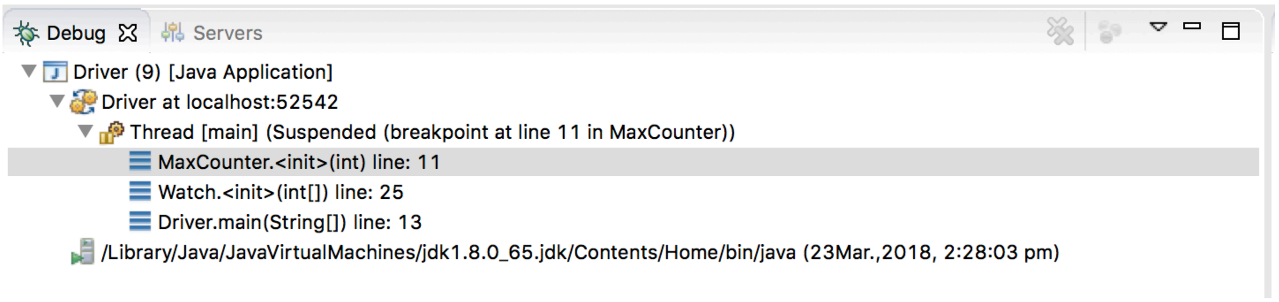
Returns the base 10 logarithm of a `double` value. Special cases:

- If the argument is NaN or less than zero, then the result is NaN.
- If the argument is positive infinity, then the result is positive infinity.
- If the argument is positive zero or negative zero, then the result is negative infinity.
- If the argument is equal to 10^n for integer *n*, then the result is *n*.

So, now we know where the `NaN` came from, that ultimately caused the string formatter to throw an exception. We tried to take the logarithm of a negative number. The next question is where did that negative number come from?

[39] https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html - jls-4.2.3
[40] https://en.wikipedia.org/wiki/IEEE_754
[41] https://docs.oracle.com/javase/7/docs/api/java/lang/Math.html - log10(double)

We have found the problem is due to a parameter passed to the `MaxCounter` constructor. The stack trace we saw on page 44 does not include any call to that constructor. We need to know where the problematic call came from. Another stack trace would be useful at this point. Thankfully, the debug perspective already gives us that, in the sub-window at top left:



```
Driver (9) [Java Application]
  Driver at localhost:52542
    Thread [main] (Suspended (breakpoint at line 11 in MaxCounter))
      MaxCounter.<init>(int) line: 11
      Watch.<init>(int[]) line: 25
      Driver.main(String[]) line: 13
  /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java (23Mar.,2018, 2:28:03 pm)
```

We see that we got here from a call to the `Watch` constructor, that came from line 13 of `Driver`. Double-clicking on that takes us the ultimate source of the bug:

```
12          //watches.add(new Watch2());
13          watches.add(new Watch(new int[] {24, 60, -60, 1000} ));
14
```

We see the `-60` that caused the problem!

Fail Fast

Whilst it was useful to learn about how to use a stack trace and the debugger in the previous section, it was undeniably quite a challenge to track the bug down. Why was this so? It was because the exception that actually caused the program to crash occurred a long way away in time (from a computer's perspective) and space (in code) from the actual bug that was the source of the problem. Our lives would have been a lot easier if the program had complained at the time and place that the actual problem occurred.

This brings us to another principle of good design and good coding:

Fail Fast

The Fail Fast Principle says that a system should fail immediately and visibly when something is wrong.[42]

This might at first seem counter-intuitive? Wouldn't this make our system more fragile? Shouldn't we try to prevent failure? In fact, the opposite is true – particularly for software. Systems that fail fast make it easier to find and fix errors, and make it more likely that the error will be found. This means that the resultant program will be *more robust* in the end, because it is more likely that errors will have been found before it is shipped, and less likely that they will have been introduced in the first place.

It is perhaps easiest to understand this using an example. The problem we saw above was ultimately called by a client passing a negative value into the constructor for `Watch`, which in turn passed it to the constructor of `MaxCounter` (sometimes via the `LinkedCounter` constructor). The designer of `MaxCounter` knows that `Math.`*`log10`*`(max)` is called in the constructor, so `max` can't be negative. We could and should add documentation stating this. Even better, however, we can cause the program to fail if this condition is violated.

[42] https://martinfowler.com/ieeeSoftware/failFast.pdf

Assertions

Java provides a mechanism for the programmer to *assert* something that should be true at a certain point in the code. This is called an assertion.[43] If the assertion is violated, an `AssertionError` is thrown[44]. We can add an assertion to the `MaxCounter` constructor to assert that the maximum value must be greater than zero:

```
public MaxCounter(int max) {
    assert max > 0 : "Maximum value of a MaxCounter must be greater than zero";
    this.max = max;
    // create a format string with the correct field width for this counter
    double fieldWidth = Math.ceil(Math.log10(max));
    fieldFormat = "%0" + String.format("%.0f", fieldWidth) + "d";
}
```

The line

```
    assert max > 0 : "Maximum value of a MaxCounter must be greater than zero";
```

is of the form

```
assert Expression1 : Expression2 ;
```

where:

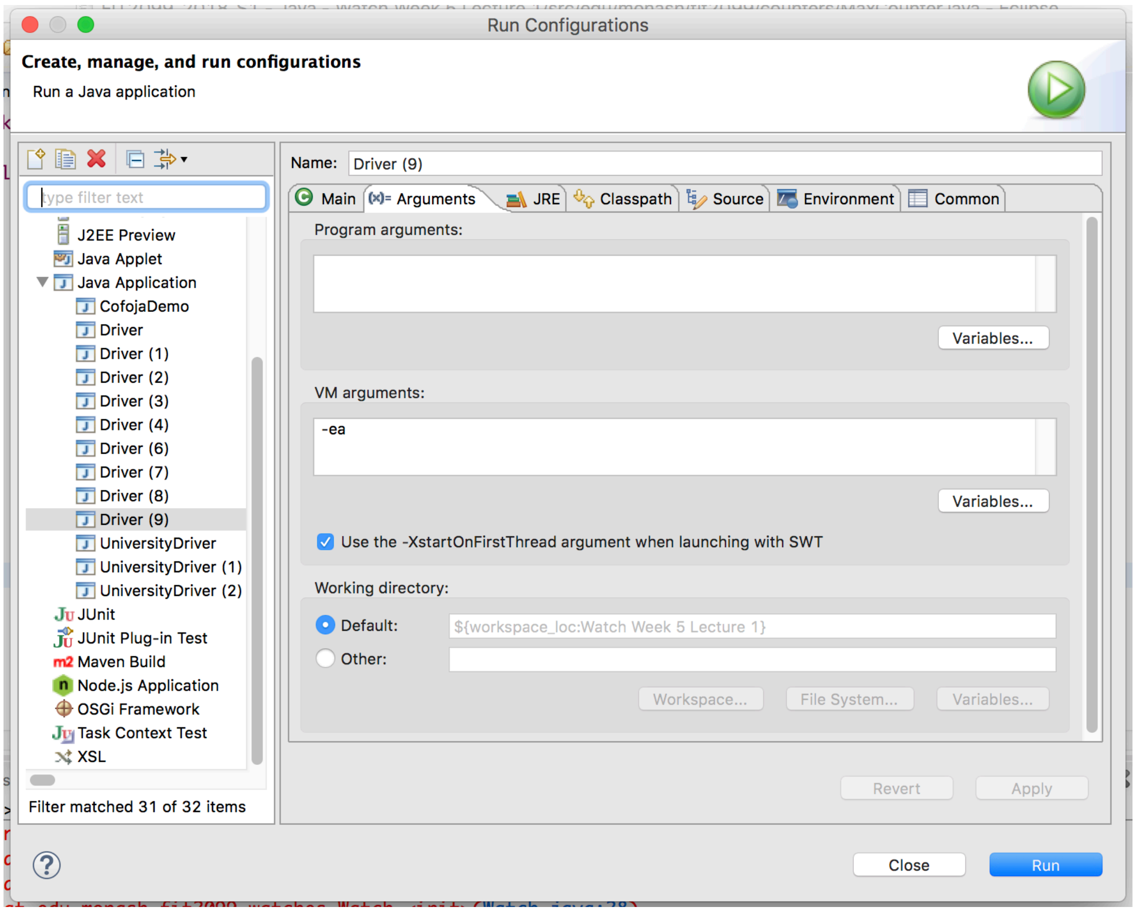
- `Expression1` is a boolean expression.
- `Expression2` is an expression that has a value. (It cannot be a call to method that is declared `void`.)

The `: Expression2` part is optional, but very useful, as it allows us to provide a detailed explanation of the error, which is not always clear simply from reading `Expression1`.

[43] https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html
[44] An AssertionError is very like an exception.

Enabling assertions

When we first run this, however, nothing happens: we still get the same exception we saw on page 44. This is annoying. It is because assertion checking is not enabled by default in the Java virtual machine (VM). We need to use a switch, `-enableassertions` or `-ea`, when we start the Java VM. We can do this in Eclipse by right-clicking on our project, selecting Run As, and then selecting Run Configurations… . This brings up a dialog box, in which we can enter the switch in the `Arguments` tab:



Name: Driver (9)

Program arguments:

VM arguments: -ea

[✓] Use the -XstartOnFirstThread argument when launching with SWT

Working directory:
[● Default:] ${workspace_loc:Watch Week 5 Lecture 1}
Other:

Filter matched 31 of 32 items

Once we have done this, when we run our program it fails immediately, before any output is produced, with the message:

```
Exception in thread "main" java.lang.AssertionError: Maximum value of a MaxCounter must
be greater than zero
    at edu.monash.fit2099.counters.MaxCounter.<init>(MaxCounter.java:9)
    at edu.monash.fit2099.counters.LinkedCounter.<init>(LinkedCounter.java:8)
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:28)
    at Driver.main(Driver.java:13)
```

It has failed fast. There is an informative message telling us exactly what the problem is:

```
Maximum value of a MaxCounter must be greater than zero
```

The stack trace is exactly the one we need, showing the sequence of constructor calls that led to the problem. It starts from line 13 of `Driver`, which is what we eventually tracked down the hard way using the debugger. **FF** has made our life a lot easier.

Question: can you see other places where our Watch system could be made to crash by a client?

Assertions for the `Watch` constructor

There are several other ways that a client can cause a crash. What happens if we pass a null array reference to `Watch(int[] maxValues)`?

```
        watches.add(new Watch(null));
```

This compiles, because passing a null reference is sometimes something that a programmer has a good reason to do. It crashes our system though, with the exception:

```
Exception in thread "main" java.lang.NullPointerException
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:25)
    at Driver.main(Driver.java:13)
```

Line 25 of Watch is:

```
        MaxCounter lastCounter = new MaxCounter(maxValues[0]);
```

We are trying to access the item at position `0` in the array `maxValues`, but `maxValues` is `null` – we are thus trying to dereference a null reference – in this case to use an array that doesn't exist.[45] This causes a `java.lang.NullPointerException` to be thrown.[46]

We can add an assertion to the `Watch(…)` constructor to make it clear that this is not a valid way to call this constructor:

```
    public Watch(int[] maxValues) {

        assert maxValues != null : "Null reference passed to Watch constructor.";
...
```

Now the program crashes with the message:

```
Exception in thread "main" java.lang.AssertionError: Null reference passed to Watch
constructor.
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:25)
    at Driver.main(Driver.java:13)
```

This hasn't added much – an experienced Java programmer would have worked out straight away what the problem was.

Can we find other ways to crash our system? What happens if we try to create a `Watch` using a zero-length array?

```
        watches.add(new Watch(new int[] {} ));
```

This crashes the system with the exception:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:25)
    at Driver.main(Driver.java:13)
```

[45] To "dereference a reference" means to access the thing that the reference refers to. You will also often see the expression "dereference a pointer" in programming literature.
[46] It is somewhat confusing that the term "Pointer" is used rather than "Reference". Java doesn't have pointers, where as C and C++ do. Pointers and references are very similar concepts – something that tells you where to find another thing.

This time we have a `java.lang.ArrayIndexOutOfBoundsException` – we have tried to access an element of an array in a position that doesn't exist (in this case position `0`).

We can also add an assertion that checks for this:

```
    public Watch(int[] maxValues) {

        assert maxValues != null : "Null reference passed to Watch constructor.";
        assert maxValues.length >= 1 : "Must pass at least one counter maximum value in
array parameter";
```

Question: does the order of these assertions matter?

Now the system crashes with the message:

```
Exception in thread "main" java.lang.AssertionError: Must pass at least one counter
maximum value in array parameter
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:26)
    at Driver.main(Driver.java:13)
```

We now have a more meaningful message. Note also the assertions act as documentation of the code too. They make it clear to any reader of the `Watch(…)` code how it should be used.

`Watch` as a well-behaved client of `MaxCounter` and `LinkedCounter`

`Watch` is a client of the classes `MaxCounter` and `LinkedCounter` – it makes use of the services they supply. To be a well-behaved client, `Watch` should ensure that it uses the methods of its suppliers correctly.

A properly documented version of `MaxCounter` (which we will do soon), would tell us that the argument to its constructor must be greater than zero. As a good client, `Watch` should ensure that the values it passes to the `MaxCounter` and `LinkedCounter` constructors are greater than zero. We can add an assertion to do this:

```
     public Watch(int[] maxValues) {

        assert maxValues != null : "Null reference passed to Watch constructor.";
        assert maxValues.length >= 1 : "Must pass at least one counter maximum value in
array parameter";

        MaxCounter lastCounter = new MaxCounter(maxValues[0]);
        this.addCounter(lastCounter);
        for (int i = 1; i < maxValues.length; i++) { // notice we start from 1, not 0
            assert maxValues[i] > 0 : "Counter maximum values must be greater than zero.
maxValues[" + i + "] = " + maxValues[i];
            MaxCounter thisCounter = new LinkedCounter(maxValues[i], lastCounter);
            this.addCounter(thisCounter);
            lastCounter = thisCounter; // notice we can assign a LinkedCounter to a
MaxCounter
        }
    }
```

Now when we try to construct a `Watch` with a negative maximum value as on page 44, we get the message:

```
Exception in thread "main" java.lang.AssertionError: Counter maximum values must be
greater than zero. maxValues[2] = -60
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:31)
    at Driver.main(Driver.java:13)
```

We now have an even more informative error message

```
Counter maximum values must be greater than zero. maxValues[2] = -60
```

that specifies exactly which value in the array passed to the constructor caused the problem. Notice that the message accompanying the `AssertionError` does not have to be just a `String` literal. It can be the result of evaluating attributes, local variables, or methods.

We also have a shorter stack trace, because we have caught the problem before the `LinkedCounter` and `MaxCounter` constructors were called. This makes it even easier to track down the line of code that is at fault.

Exceptions

As we have seen, assertion checking has to be enabled by using a switch when starting the Java VM. Most languages have a similar mechanism. Checking assertions adds computational overhead at run time, so it is more efficient to run deployed code without them. Also, we should not expect to see assertions being violated in a deployed system – ideally they are tools to aid in development. The messages are aimed at developers, not end users.

Sometimes, however, things go wrong at run time that mean that the method executing cannot do its job – something exceptional has occurred. Sometimes this is due to a programming error, as with the examples above for which we used assertions. At other times it can be due to something completely beyond the control of the author of the program. For example, a network connection may go down when a method is trying to use it; a hard disk may become full and so unwritable; the process may run out of memory. In such cases, the program needs a way of reporting the problem, and cleaning up gracefully. Simply terminating the program is not a good approach. Sometimes the problem can be resolved – but the method in question has no way of knowing what the client wants to do, or can do.

A little history

There was a time when languages did not have exceptions. The only way to report that something had gone wrong was to use the return value of the function that had been called. For example, here is the documentation for the `malloc(…)` function in C, which is used to allocate a block of memory for use by a data structure.[47]

```
void* malloc (size_t size);
```

Allocate memory block
Allocates a block of size bytes of memory, returning a pointer to the beginning of the block. The content of the newly allocated block of memory is not initialized, remaining with indeterminate values. If size is zero, the return value depends on the particular library implementation (it may or may not be a null pointer), but the returned pointer shall not be dereferenced.

Parameters
`size`
Size of the memory block, in bytes. `size_t` is an unsigned integral type.

Return Value
On success, a pointer to the memory block allocated by the function. The type of this pointer is always `void*`, which can be cast to the desired type of data pointer in order to be dereferenceable. If the function failed to allocate the requested block of memory, a null pointer is returned.

[47] http://www.cplusplus.com/reference/cstdlib/malloc/

Look at the very last sentence:

If the function failed to allocate the requested block of memory, a null pointer is returned.

The function is used like this:

```
fibonacci_numbers = (long *)malloc(NUM_FIBOS*sizeof(long));
```

This is asking for a block of memory that can hold `NUM_FIBOS` things the size of the type `long`. The return value is typecast to be a pointer to a `long`. The block of memory would typically be used for an array of `NUM_FIBOS` long integers. The closest Java equivalent would be:

```
long fibonacciNumbers[] = new long[NUM_FIBOS];
```

When everything goes well, the value returned and stored in variable `fibonacci_numbers` is a pointer to the block of memory requested. If it fails, however, `fibonacci_numbers` has the value `NULL`. As soon as the program tries to use the memory at that address, it will crash with a segmentation fault. Here is some example C code:

```
long *fibonacci_numbers;
long last = 1;
long before_last = 1;

fibonacci_numbers = (long *)malloc(NUM_FIBOS*sizeof(long));
fibonacci_numbers[0] = before_last;
fibonacci_numbers[1] = last;
/* fill array with Fibonacci numbers */
for (i = 2; i < NUM_FIBOS; i++) {
    fibonacci_numbers[i] = last + before_last;
    before_last = last;
    last = fibonacci_numbers[i];
}
```

If `malloc(…)` fails for some reason,[48] this will crash at the line

```
fibonacci_numbers[0] = before_last;
```

It is illegal to try to write to the memory address `0`, which is what `NULL` ultimately is in C.

In this toy example, it is easy to work out what has happened. Sometimes however, a null pointer like this can exist for a long time before it is eventually used somewhere, quite possibly in a completely different part of the program. It can be very hard to track down the cause of such a crash, remote in time and space from the ultimate cause.

What the programmer should have done is something like this:

```
if ((fibonacci_numbers = (long *)malloc(NUM_FIBOS*sizeof(long))) == NULL ) {
    /* do what ever is necessary to clean up and exit this routine */
    ...
}
```

In this paradigm, we should always check the return value of any function that can fail and signals that using its return value.[49]

[48] In the old days, I could have made this happen easily by making `NUM_FIBOS` too large. Today, with GBs of RAM, and an operating that will swap memory to disk, I had to ask for so much memory (`NUM_FIBOS = 9999999999`) that the OS killed the process before `malloc(…)` returned.

Though the practice of using return values to signal errors was very common and is still used, it is problematic. Why? First, because programmers often don't do the right thing and check the return value for error signals. They assume everything worked, and execution continues – until one day it doesn't.

Also, it is in general considered bad design to give special meanings to certain parts of the range of a value. Meilir Page-Jones calls this "Hybrid Coupling" or "Connasence of Convention"[50]. Steve McConnell, in his classic "Code Complete", states the principle:[51]

Avoid variables with hidden meanings

McConnell says:

> [A] way in which a variable can be used for more than one purpose is to have different values for the variable mean different things. For example:
>
> - The value in the variable `pageCount` might represent the number of pages printed, unless it equals -1, in which case it indicates that an error has occurred.
> - The variable `customerId` might represent a customer number, unless its value is greater than 500,000, in which case you subtract 500,000 to get the number of a delinquent account.
> - The variable `bytesWritten` might be the number of bytes written to an output file, unless its value is negative, in which case it indicates the number of the disk drive used for the output.
>
> Avoid variables with these kinds of hidden meanings. The technical name for this kind of abuse is "hybrid coupling" (Page-Jones 1988). The variable is stretched over two jobs, meaning that the variable is the wrong type for one of the jobs. In the `pageCount` example, `pageCount` normally indicates the number of pages; it's an integer. When `pageCount` is -1, however, it indicates that an error has occurred; the integer is moonlighting as a boolean!

Language designers came up with a solution to this problem: a way of signalling that something has gone wrong, that doesn't overload the value of any variable with extra meanings, and that can't be ignored: *exceptions*.

Exceptions in Java

When a method needs to signal that something has gone wrong and it can't complete its task successfully, it does so by *throwing an exception*. The method that called the method that threw the exception can either *catch* the exception and try to deal with it, or it can throw it back to the method that called it. If the exception gets all the way back to `main(…)` without being caught, the program crashes – as we saw in the examples in the Debugging section above.

We could replace the assertions we used to check the parameters passed to the `Watch` and `MaxCounter` constructors with code that throws an exception. One reason for doing this is that some problems really shouldn't be possible to avoid if assertions are not enabled.

[49] https://wiki.sei.cmu.edu/confluence/display/c/EXP12-C.+Do+not+ignore+values+returned+by+functions
[50] Meilir Page-Jones, *Fundamentals of Object-Oriented Design in UML*, p. 217, Addison-Wesley, 2000.
[51] Steve McConnell, Code Complete, 2nd Edition, p. 256, Microsoft Press, 2004

The start of `MaxCounter(…)` is now:

```
    public MaxCounter(int max) throws Exception {

        if (max <= 0) {
            throw new Exception("Maximum value of a MaxCounter must be greater than 
zero.");
        }
...
```

We have replaced the assertion with an `if` statement that checks if max is invalid. If so, it creates a new `Exception` object and throws it. We have also had to change the signature of the constructor to indicate that it can throw an `Exception`.

Java has a built-in `Exception` class that provides several constructors for creating `Exception` objects that contain various kinds of information about the exception.[52] Here we have used the version that takes a `String` argument that is used as a message to describe what went wrong. Just as with other classes, we can create subclasses of `Exception`. This allows us to communicate the *type* of exception that has been thrown, and perhaps also to add more attributes and operations.

You may have noticed that as soon as we made these changes to `MaxCounter`, Eclipse started to complain about several other classes: all of those that call `MaxCounter(…)`. This is because any code that calls a method that can throw a (checked) exception must declare what it is going to do with it. It can either deal with it itself, or simply say that it will pass it on to the method that called it.

To fix this, we need to add throws `Exception` to the signature of everything that directly or indirectly calls the `MaxCounter` constructor – all the watch constructors, the `LinkedCounter` constructor, and `main(…)` in `Driver` (or make them catch the exception). When we have done this and run the program with a negative counter value, the program crashes with the message:

```
Exception in thread "main" java.lang.Exception: Maximum value of a MaxCounter must be 
greater than zero.
    at edu.monash.fit2099.counters.MaxCounter.<init>(MaxCounter.java:11)
    at edu.monash.fit2099.counters.LinkedCounter.<init>(LinkedCounter.java:8)
    at edu.monash.fit2099.watches.Watch.<init>(Watch.java:33)
    at Driver.main(Driver.java:13)
```

Notice the type of Exception is now `java.lang.Exception` rather than `java.lang.AssertionError`. This will occur even if assertions are not enabled.

[52] https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Catching exceptions

Rather than simply passing the exception on, the calling method has the option of trying to deal with it. It does this by *catching* the exception. For example, we could change `Driver` to:

```
    public static void main(String[] args) {

        ArrayList<Watch> watches = new ArrayList<Watch>();

        // watches.add(new Watch1());
        // watches.add(new Watch2());
        try {
            watches.add(new Watch(new int[] {24, 60, -60, 1000} ));
        }
        catch (Exception e) {
            System.out.println("Watch construction failed with message: \n\t"
                    + e.getMessage()
                    + "\nLet's not bother with this watch for now."
            );
        }
```

When we run the system now, it doesn't crash. The output of the program is:

```
Watch construction failed with message:
    Maximum value of a MaxCounter must be greater than zero.
Let's not bother with this watch for now.
##############################
```

Notice that we no longer need to specify that `main(…)` can throw an exception, because it has been caught.

This is quite an artificial example – usually simply ignoring the exception and carrying on is not an option. When catching an exception, it is more common to do something such as:

- error recovery, for example:
  - wait and try again (might make sense for a transient problem such as a network connectivity issue)
  - try a different way to achieve the method's goal
- prompting the user to make a decision
- propagating the error up to a higher-level handler[53]
  - often after doing some cleaning up, e.g. closing files, saving the program state, writing to a log file, etc.

Unchecked exceptions

The `Exception` object used above is an example of a checked exception. Checked exceptions form part of the API of a method – the signature tells us the method can throw an exception, and the type of exception it throws. This alerts clients of the method that it can fail, and they then have the chance to catch the exception and deal with it.

Java also has another type of exception: unchecked exceptions.[54] These are subclasses of `RuntimeException` or `Error`, and they do not have to be specified in the signature of a method in which they might arise, or otherwise caught. They are used for situations in which it is not reasonable to expect the client to be able to recover from or handle the exception –

[53] For more details, see https://docs.oracle.com/javase/tutorial/essential/exceptions/chained.html

[54] https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

typically because the are the result of programmer error, rather than user input or the external environment.

We can change the `MaxCounter` constructor to throw an unchecked exception. It turns out that Java has a built-in subclass of `RuntimeException` that is exactly what we need: `IllegalArgumentException`. Using this subtype lets us convey more information about what sort of thing has gone wrong. Client code may be able to catch and handle some types of exception, but not others.

```
    public MaxCounter(int max) {

        if (max <= 0) {
            throw new IllegalArgumentException("Maximum value of a MaxCounter must be 
greater than zero.");
        }
...
```

Notice that we no longer have to specify that this constructor can throw an exception. Nor do we have to specify that everything that calls it can do so. We can remove the `throws` clause from the constructors of the `Watch` subclasses and `LinkedCounter`.

Unchecked exceptions can still be caught. When we run our program after these changes, we still get

```
Watch construction failed with message:
    Maximum value of a MaxCounter must be greater than zero.
Let's not bother with this watch for now.
##############################
```

Do not fall into the temptation of using unchecked exceptions because it is “easier”. As the Oracle tutorial says:[55]

> Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

[55] https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

Documenting Code with Javadoc

Code is often worked on by many people and has a very long lifespan. These people need information about how the code works, and why it is written the way it is.[56] It is thus vitally important to document your code properly. The easiest and best place to document the details of an individual class is in the class itself. Documentation in the class is essential when we need to maintain or extend the class.

Other programmers using your class, however, should not be required to look at the source code – it's slow and can lead to them relying on implementation details you want the freedom to change later.[57] For library classes, the source code may not even be available. It is thus vitally important to be able to produce separate documentation for the *interface* of the class: those class members that are visible from outside the class.

Javadoc simplifies this process – if you format your comments correctly, you can generate well-formatted API documentation automatically. This is how the Java API documentation we have frequently referenced is generated.

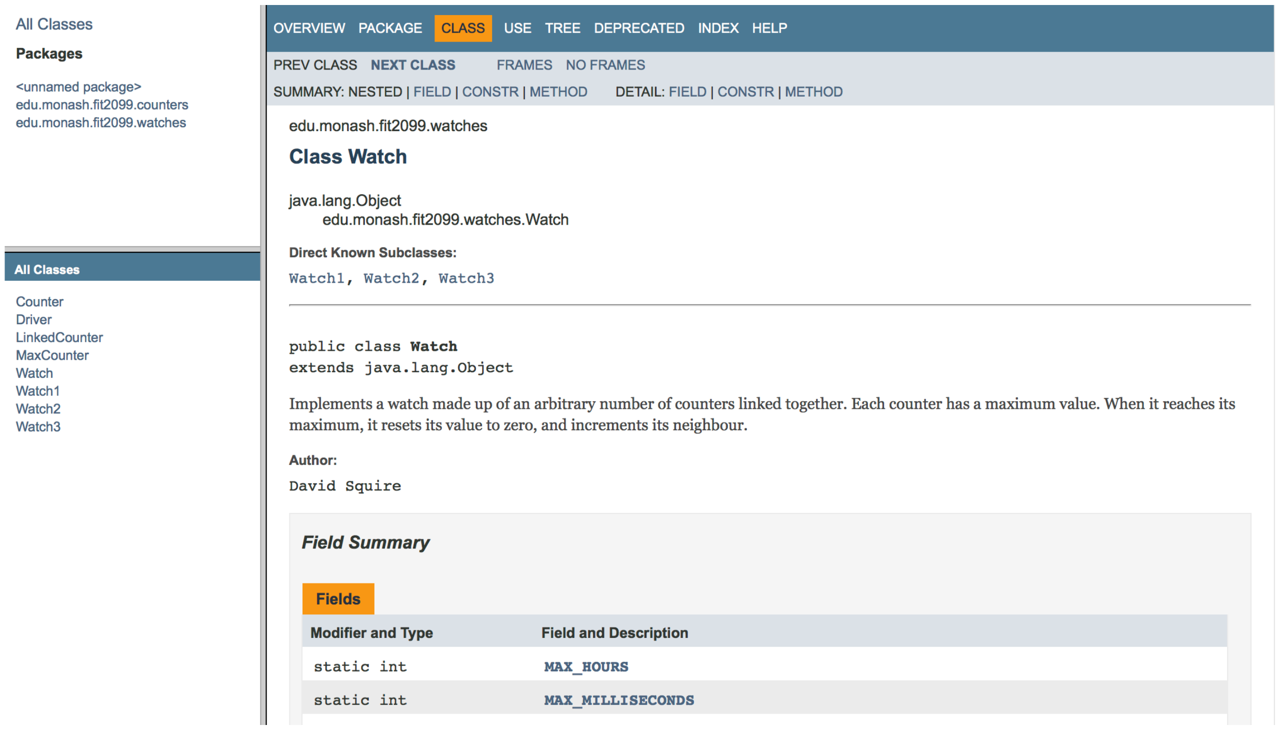
There are separate documents on Moodle that provide an introduction to Javadoc, and other information about documenting code. Let us now see how we can use Javadoc to document the `Watch` class, and generate the HTML API documentation from the code.

First we will add documentation for the class itself. We do this in a comment immediately before the class declaration:

```java
package edu.monash.fit2099.watches;

import java.util.ArrayList;

import edu.monash.fit2099.counters.*;

/**
 * Implements a watch made up of an arbitrary number of counters linked together.
 * Each counter has a maximum value. When it reaches its maximum, it resets its
 * value to zero, and increments its neighbour.
 *
 * @author David Squire
 *
 */
public class Watch {

    static public final int MAX_HOURS = 24;
...
```

[56] The person that needs the documentation could even be you in a few year's time. There's an old saying: "The day you write your code, you and God understand how it works. Six months later, only God does."
[57] *Question: can you think of ways in which a client of your class could rely on implementation details even if the attributes and methods involved are private?*

If we then select Generate Javadoc… from the Project menu in Eclipse, the API documentation for our system is automatically generated. If we view it in the web browser and navigate to the Watch documentation, we see


All Classes
Packages
<unnamed package>
edu.monash.fit2099.counters
edu.monash.fit2099.watches
All Classes
Counter
Driver
LinkedCounter
MaxCounter
Watch
Watch1
Watch2
Watch3
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

edu.monash.fit2099.watches

Class Watch

java.lang.Object
    edu.monash.fit2099.watches.Watch

Direct Known Subclasses:

Watch1, Watch2, Watch3

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

Implements a watch made up of an arbitrary number of counters linked together. Each counter has a maximum value. When it reaches its maximum, it resets its value to zero, and increments its neighbour.

Author:

David Squire

Field Summary

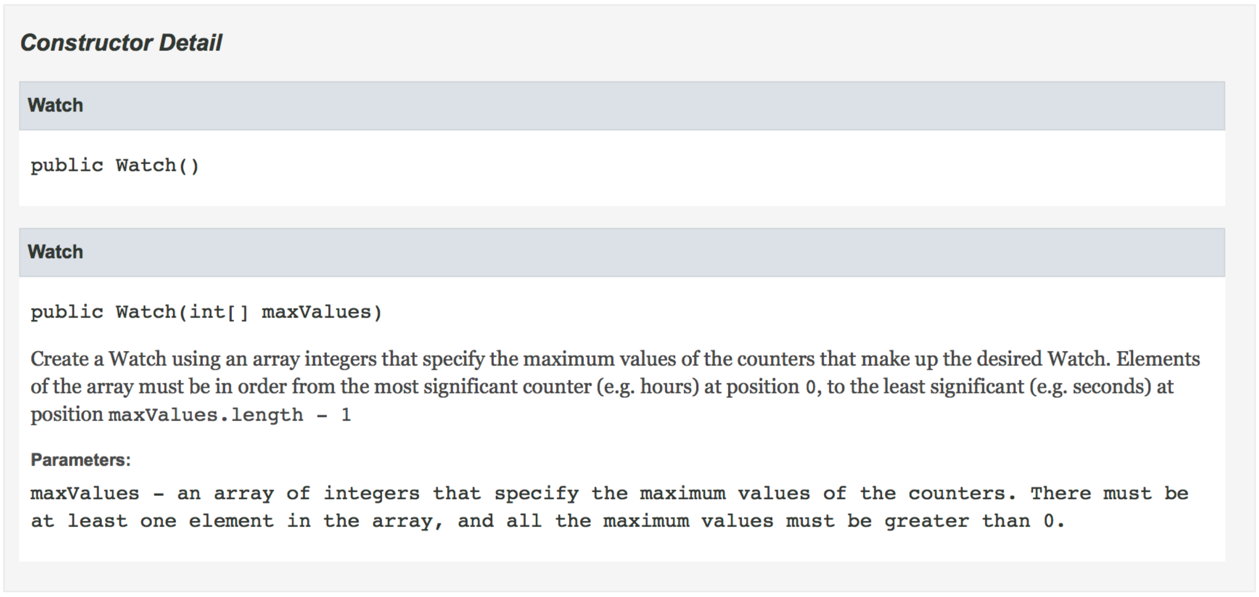
Fields

Modifier and Type	Field and Description
static int	MAX_HOURS
static int	MAX_MILLISECONDS

Now we will add documentation for the constructor. We do this immediately before its declaration:

```
/**
* Create a Watch using an array integers that specify the maximum values of the
* counters make up the desired Watch. Elements of the array must be in order from
* the most significant counter (e.g. hours) at position {@code 0}, to the least
* significant (e.g. seconds) at position {@code maxValues.length - 1}
*
* @param maxValues an array of integers that specify the maximum values of the
* counters. There must be at least one element in the array, and all the maximum
* values must be greater than 0.
*
*/
public Watch(int[] maxValues) {

   assert maxValues != null : "Null reference passed to Watch constructor.";
...
```

When we generate the Javadoc HTML from this, we see



Constructor Detail

Watch

`public Watch()`

Watch

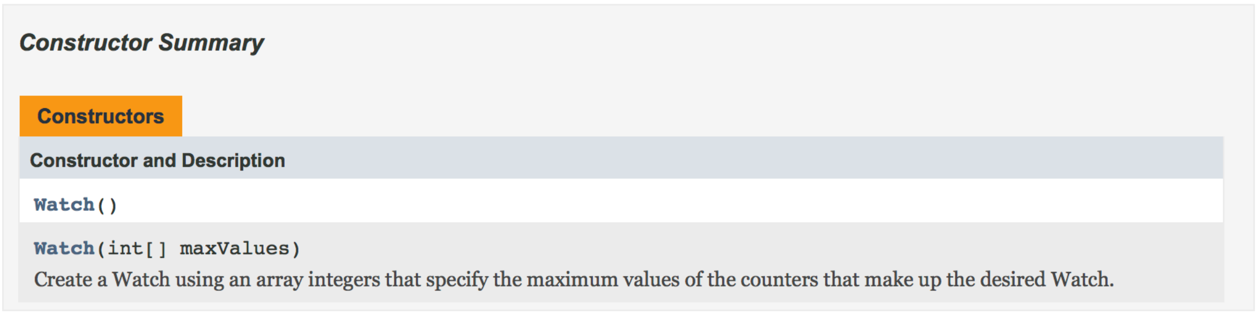
`public Watch(int[] maxValues)`

Create a Watch using an array integers that specify the maximum values of the counters that make up the desired Watch. Elements of the array must be in order from the most significant counter (e.g. hours) at position `0`, to the least significant (e.g. seconds) at position `maxValues.length - 1`

Parameters:

`maxValues - an array of integers that specify the maximum values of the counters. There must be at least one element in the array, and all the maximum values must be greater than 0.`

Notice that in the Constructor Summary section, only the first sentence is used



Constructor Summary

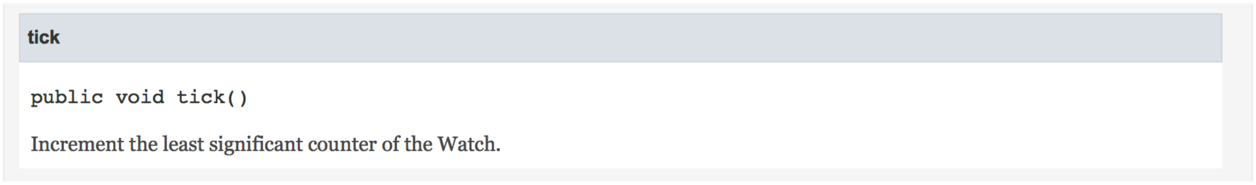
Constructors

Constructor and Description
`Watch()`
`Watch(int[] maxValues)` Create a Watch using an array integers that specify the maximum values of the counters that make up the desired Watch.

We can do the same thing for the methods. For example, for `tick()` we have:

```
/**
 * Increment the least significant counter of the Watch.
 */
public void tick() {
    getLeastSignificantCounter().increment();
}
```

This produces



tick

`public void tick()`

Increment the least significant counter of the Watch.

This is a very easy way of putting documentation in the best place, and having publically viewable API documentation generated as well. Another example of **DRY**.