OBJECT ORIENTATION FUNDAMENTALS IN JAVA – PART 3

Named Constants in Java

Back on page 7, we introduced the design principle that we should avoid excessive use of literals. We made a start on addressing this issue by moving the literals that specified the maximum values of the `minutes` and `hours` counters out of the body of the `tick()` method and into the constructor of the watch classes when we introduced `MaxCounter` in `Watch2`. The original Java Code Conventions standard[20], however, states in section 10.3:

> *Numerical constants (literals) should not be coded directly, except for -1, 0, and 1, which can appear in a for loop as counter values.*

This is a common and good convention applied in many languages – there is nothing specifically OO or Java about it. For any constant other than these fundamental ones, it is not clear *why* the constant has the value it does. We are currently in violation of the convention, with literals `60` and `24` appearing in several places in our code. In `Watch3`, for example, `60` appears twice – once for the `minutes` counter, and once for the `seconds` counter. Are these values related? Can I change one without changing the other?[21] It is difficult to know.

`static final` fields

The solution to these issues is to give such literals a name and type which are declared once **DRY**, and can never change. In Java this is done by using two modifiers when declaring fields that represent constants: `static` and `final`.

The `static` modifier indicates that the class member it qualifies is shared by all instances of a class. A class can have both static fields and static methods.[22] The `final` modifier indicates that the value of the field it qualifies can never change. A `static final` field is thus used to represent a constant: something common to all instances of the class that never changes.

We can use this approach to define that constants needed by our watches at the beginnings of those classes. For Watch3, we have:

```
public class Watch3 extends Watch {
...
    static final int MAX_HOURS = 24;
    static final int MAX_MINUTES = 60;
    static final int MAX_SECONDS = 60;

    public Watch3() {

        hours = new MaxCounter(MAX_HOURS);
        this.addCounter(hours);
        minutes = new LinkedCounter(MAX_MINUTES, hours);
        this.addCounter(minutes);
        seconds = new LinkedCounter(MAX_SECONDS, minutes);
        this.addCounter(seconds);
    }
...
```

[20] Now to some extent superseded by the Google Java Style Guide: https://google.github.io/styleguide/javaguide.html

[21] These values are conventions, not facts of nature. During the French Revolution, Decimal Time was indeed officially introduced, though it was never widely used.

[22] https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html

Notice that we have not used the access level modifier `private` for these `final` fields. We could have, but there is often no need. Final fields can't be modified after creation, so there is no danger of unauthorised modification as there is with non-final fields (which should all be private). You might still want to keep them private to hide implementation details though, or to avoid privacy leaks (if their types are not primitive).

The names of these static fields appear in *italics* in Eclipse, as do names of static methods. This is a useful visual hint that these members are special, and that there are limitations on what you can do with them – static fields can't be modified; static methods (a.k.a. class methods) can't change the values of non-static fields in that class (can you see why?).

In our system, the constants used for the maximum values of counters are the same for all our watches – **DRY**! They can thus also be pulled up into the `Watch` abstract class:[23]

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

import java.util.ArrayList;

import edu.monash.fit2099.counters.Counter;

public abstract class Watch {

    static final int MAX_HOURS = 24;
    static final int MAX_MINUTES = 60;
    static final int MAX_SECONDS = 60;

    public abstract void tick();

    private ArrayList<Counter> counters = new ArrayList<Counter>();

    protected void addCounter(Counter newCounter) {
        counters.add(newCounter);
    }

    public void display() {
        String prefix = "";
        for (Counter thisCounter : counters) {
            System.out.print(prefix
          + String.format("%02d", thisCounter.getValue()));
            prefix = ":";
        }
        System.out.println();
    }

    public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            display();
            tick();
        }
    }
}
```

[23] The statement that prints the value of `thisCounter` in display is broken across two lines. This is done so that the second line starts with the operator +. This is a good convention for readable code. See https://google.github.io/styleguide/javaguide.html#s4.5-line-wrapping

Indirect Dependencies

One of the design principles we have mentioned in the preceding parts is that we should reduce dependencies whenever possible **ReD**!

Some dependencies in code are obvious. For example, class `A` depends on class `B` if class `A` has members or variables that are of type `B`. We can also see such dependencies at the level of particular attributes or methods. A compiler can see these dependencies, and thus tools exist that can help us find and manage them. We can call these *direct dependencies*.

Other kinds of dependency are invisible to the compiler. They exist because of the meaning of various elements of the code to humans, and only a human programmer can know they exist. For example, the values of variables or literals in different parts of the program can be related to each other, and must be changed simultaneously if correctness is to be maintained. These are the most dangerous dependencies. Automated tools can't help us to find them, and it is easy for them to be overlooked when code is maintained over long periods of time – particularly by multiple programmers. *Indirect dependencies* such as these must be very carefully documented – or even better eliminated: **ReD**. The use of named constants eliminates one source of such dependencies – a literal that must be the same in multiple places in the code because it represents the same thing.

The dependency between the maximum value of a `Counter` and how it is displayed

We have signalled several times in the earlier parts of this document our discomfort with the dependency between the literals used to specify the maximum values of the counters, and the literal used in the format string in the display() method to specify the field width and zero-padding of the counter value:

```
System.out.print(prefix + String.format("%02d", thisCounter.getValue()));
```

The format specifier `%02d` indicates that we are formatting (`%`) a decimal integer (`d`) of a particular width (`2`), and that we want leading zero-padding (`0`).[24] The key problem for us is the width `2`. This line of code is effectively assuming that *all* counters have a width of `2`. That is not a good assumption.

In fact, the field width depends on the maximum value of the `Counter` which `thisCounter` references. That is determined when the counter object is created in the constructors of the classes `Watch2` and `Watch3`. For `Watch1`, the maximum values of the counters are not stored anywhere – they are just constants embedded in the `tick()` method.

[24] https://docs.oracle.com/javase/tutorial/java/data/numberformat.html Format strings like this date back to languages from the 1960s, and came to Java via C.

What happens if we modify `Watch3` to have milliseconds as its least significant counter?

```
package edu.monash.fit2099.watches;
import edu.monash.fit2099.counters.LinkedCounter;
import edu.monash.fit2099.counters.MaxCounter;

public class Watch3 extends Watch {

    private LinkedCounter milliseconds;
    private LinkedCounter seconds;
    private LinkedCounter minutes;
    private MaxCounter hours;

    public Watch3() {

        hours = new MaxCounter(MAX_HOURS);
        this.addCounter(hours);
        minutes = new LinkedCounter(MAX_MINUTES, hours);
        this.addCounter(minutes);
        seconds = new LinkedCounter(MAX_SECONDS, minutes);
        this.addCounter(seconds);
        milliseconds = new LinkedCounter(MAX_MILLISECONDS, seconds);
        this.addCounter(milliseconds);
    }

    public void tick() {
        milliseconds.increment();
    }

}
```

The system compiles and runs, but when the milliseconds counter value is less than 100, the output looks like this:

```
00:00:00:97
00:00:00:98
00:00:00:99
00:00:00:100
00:00:00:101
00:00:00:102
```

The zero-padding is wrong. For the milliseconds counter, the field width should be 3. A quick-and-dirty solution would be to create version of the `display()` method in `Watch3` that overrides the version in `Watch`. Hopefully that idea is setting off alarm bells in your head, and setting your code-smell detecting nose twitching. That would violate **DRY** – which is often the consequence quick-and-dirty hacks. We would also need to give `Watch` subclasses access to the `counters` attribute.

To solve this design problem, we need to ask a few questions. First, how can we get the value we need – the required field width – where we need it. What knows it?

First, let's refactor `Driver` to use an `ArrayList` rather than an array of watches. We'll also ask the watches we iterate over to tell us their class name, rather than constructing it using a counter: **Red**! Remember to explore the methods available for an object by using your IDE (e.g. Eclipse) – I did not know about `getSimpleName()` until I looked for it.

```
import java.util.ArrayList;

import edu.monash.fit2099.watches.*;

public class Driver {

    public static void main(String[] args) {

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

        watches.add(new Watch1());
        watches.add(new Watch2());
        watches.add(new Watch3());

        System.out.println("##############################");
        for (Watch watch : watches) {
            System.out.println("Testing Watch: "
          + watch.getClass().getSimpleName());
            watch.testWatch(200);
            System.out.println("##############################");
        }
    }

}
```

The counters in `Watch2` and `Watch3` all know their maximum values. It's stored in their `max` attribute. If we add a `getMax()` method to `MaxCounter`, we can access that value in `display()`, and use it to set the field width.

Question: what is the mathematical function that relates the maximum value of a counter to its required field width?

There is no way to do this for `Watch1` though. It uses the basic `Counter`, which has no `max` attribute . We could introduce another abstract watch class as a subclass of `Watch` – perhaps `MaxWatch` – to cope with this, but that would add unnecessary complication. Perhaps at this stage it's time to say goodbye to the `Watch1` implementation.

Delegation

`Watch1` has exactly the same functionality as `Watch2`. It's tempting simply to delete the class once and for all. In this case, we could easily remove all references to `Watch1` from `Driver`. What if there were *other* code that depended on `Watch1` though? In large systems this could well be the case. Deleting `Watch1` would break that code. If `Watch1` were in library code, we would have no way of knowing what code elsewhere might be using it.

This is a fairly common problem. We want to keep the type `Watch1` around so as not to break code that depends on it, but not its separate (and out-dated) implementation. One way to solve the problem is to give `Watch1` an attribute of type `Watch2`, and to *delegate* all behaviour to that `Watch2` object.[25]

[25] Delegation is useful in many other situations too. It's not just for preserving legacy types. It's crucial to design concepts such dependency injection, or implementing inheritance via composition.

Watch1 is now:

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

public class Watch1 extends Watch {

    private Watch2 myWatch2;

    public Watch1() {
        myWatch2 = new Watch2();
    }

    public void tick() {
        myWatch2.tick();// delegation
    }

    @Override
    public void display() {
        myWatch2.display(); // delegation
    }

}
```

Importantly, there is no longer any reference to the base `Counter` class in `Watch` or any of its subclasses. This means that we can now change all references to `Counter` in `Watch` to `MaxCounter` without breaking anything. In particular:

```
private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();
```

The container `counters` now contains references to `MaxCounter` objects. This means that anything that iterates over it can use the methods of `MaxCounter`. We can thus rewrite `display()` to use the `getMax()` method of each counter.

Computing the field width using the `Math` class

The Java platform provides a class `Math` as part of the `java.lang` package.[26] `Math` provides basic mathematical operations such as logarithm, square root, and trigonometric functions.

The mathematical function we need to compute the field width is the base 10 logarithm: $\log_{10} 10 = 1$, $\log_{10} 100 = 2$, $\log_{10} 1000 = 3$, and so on; $\log_{10}(x)$ returns the power to which 10 must be raised to give x. We can use the `Math` class to compute this with the expression `Math.log10(x)`. A counter with a maximum value of 1000 should have a field width of 3. In fact, any counter with a maximum value between 101 and 1000 needs a field width of 3.

Now, $\log_{10} 500 = 2.699$. So, what we need for the field width is the smallest integer greater than or equal to $\log_{10}(max)$. Finding this is also a standard mathematical operation, known as computing the *ceiling* of a number. `Math` provides a `ceil()` operation that takes a `double` as its argument and returns a `double`. We need just the integer part of this in our format string – and we can use another format string to get it![27]. We can format this double using `%.0f` to format a floating point number (`f`) with zero decimal places (`.0`).

[26] https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

[27] There are other ways we could have done this, including using the *number classes*. See https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html to read about these *wrapper classes*, and the concepts of *boxing* and *unboxing*.

This allows us to rewrite `Watch` as

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

import java.util.ArrayList;

import edu.monash.fit2099.counters.*;

public abstract class Watch {

    static final int MAX_HOURS = 24;
    static final int MAX_MINUTES = 60;
    static final int MAX_SECONDS = 60;
    static final int MAX_MILLISECONDS = 1000;

    public abstract void tick();

    private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();

    protected void addCounter(MaxCounter newCounter) {
        counters.add(newCounter);
    }

    public void display() {
        String prefix = "";
        for (MaxCounter thisCounter : counters) {
            double fieldWidth = Math.ceil(Math.log10(thisCounter.getMax()));
            // create a format string with the correct field width for this counter
            String fieldFormat = "%0" + String.format("%.0f", fieldWidth) + "d";
            System.out.print(prefix + String.format(fieldFormat,
thisCounter.getValue()));
            prefix = ":";
        }
        System.out.println();
    }

    public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            display();
            tick();
        }
    }
}
```

Running this, we see that the output is now correct:

```
Testing Watch: Watch3
00:00:00:000
00:00:00:001
00:00:00:002
00:00:00:003
```

We have succeeded in eliminating the literal integer 2 from the format string for the counters, and, even more importantly, the indirect dependencies this introduced between `Watch` and all of its subclasses have been eliminated **ReD**. We now have a solution that formats the values of counters correctly no matter what their maximum values.

Making Counters responsible for their own string representation

You may have noticed that sometimes we have been able to concatenate something that is not a String with a String. In page 18, for example, we had the line:

```
System.out.println("Testing Watch" + (i+1));
```

When `i` is equal to `0`, this produced the output:

```
Testing Watch1
```

Here we have an integer expression `(i+1)` that is being concatenated with the string `"Testing Watch"`. Why does this work? It does so because Java, like many languages, does *implicit type conversion* in some situations.[28] In this case, it decides that when an integer is used in a `String` context, what is wanted is a `String` containing the digits representing that integer as a decimal (i.e. in base 10). This is a reasonable thing to do in many cases, and is thus the default behaviour.[29]

In many situations, however, there is no obvious way to represent an object as a `String`. Let's see what happens, for example, if we try to print `MaxCounter` objects directly in the `display()` method of `Watch`. We change display to:

```
public void display() {
    String prefix = "";
    for (MaxCounter thisCounter : counters) {
        System.out.print(prefix + thisCounter);
        prefix = ":";
    }
    System.out.println();
}
```

Perhaps surprisingly, this does compile and run. The output looks like this:

```
Testing Watch: Watch1
edu.monash.fit2099.counters.MaxCounter@4e25154f:edu.monash.fit2099.counters.LinkedCounte
r@70dea4e
edu.monash.fit2099.counters.MaxCounter@4e25154f:edu.monash.fit2099.counters.LinkedCounte
r@70dea4e
...
```

Java has provided a `String` representation for the `MaxCounter` objects. It consists the full name of the class of the object, the `@` symbol, and the hashcode of the object in hexadecimal.[30] This is basically never the representation wanted in any real program output. So, where did this representation come from?

---

[28] https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

[29] We have seen several examples of how to use `String.format(...)` to get different string representations of integers and floating point numbers when we don't want the default behaviour.

[30] This is the default implementation of the `toString()` method of `Object`. See https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html - toString--

The Java universal base class `Object`

In Java every class is, either directly or indirectly, a subclass of the base class `Object`. `Object` is the ultimate superclass of all classes.[31] All classes inherit methods from `Object` that are useful in many contexts. We used one of these, `getClass()` on page to get the class names of the watches we were testing. You may have used another of these methods, `clone()`, in labs. The full list of the methods of `Object` is:

```
protected Object clone() throws CloneNotSupportedException
```
Creates and returns a copy of this object.

```
public boolean equals(Object obj)
```
Indicates whether some other object is "equal to" this one.

```
protected void finalize() throws Throwable
```
Called by the garbage collector on an object when garbage collection determines that there are no more references to the object

```
public final Class getClass()
```
Returns the runtime class of an object.

```
public int hashCode()
```
Returns a hash code value for the object.

```
public String toString()
```
Returns a string representation of the object.

The `notify()`, `notifyAll()`, and `wait(…)` methods of `Object` all play a part in synchronizing the activities of independently running threads in a program. We won't cover threads in this unit:

```
public final void notify()
public final void notifyAll()
public final void wait()
public final void wait(long timeout)
public final void wait(long timeout, int nanos)
```

The `String` representation of `MaxCounter` we saw above is what the default implementation of `toString()` returns – when an object is used in a `String` context, Java implicitly calls its `toString()` method to perform the type conversion. We can put classes in charge of the string representation of objects of that type by overriding the `toString()` method.

[31] You can read about the class `Object` at https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html There has been considerable criticism of the choice of the Java designers to call this *class* "Object".

A `MaxCounter` that provides its own `String` representation

We can redesign `MaxCounter` by moving the code for formatting the counter value out of `Watch` and into `toString()` in `MaxCounter`. We've given `MaxCounter` a `fieldFormat` attribute, as this only needs to be computed once, and thus this can be done in the constructor.

```
package edu.monash.fit2099.counters;

public class MaxCounter extends Counter {

    private final int max;
    private final String fieldFormat;

    public MaxCounter(int max) {
        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";
    }

    public int getMax() {
        return max;
    }

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

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

The version of `display()` in `Watch` we saw on page 35 now produces the desired output.

For completeness, we should also give `Counter` an appropriate version of `toString()`. The basic `Counter` doesn't have a maximum value, so the only sensible thing to do is to return the `String` representation of the value of the counter:[32]

```
@Override
public String toString() {
    return Integer.valueOf(this.getValue()).toString();
}
```

[32] There is a simpler, but arguably dirtier, way of doing this:

```
    return "" + this.getValue();
```

The Java Numbers Classes

Above we have used one the Java numbers *wrapper classes,* `Integer`.[33] The wrapper classes "wrap" primitive data types (e.g. `int`, `double`, etc.) so that they can be used as objects. There are several reasons for wanting to do this. You might want to use the primitive in a context where an object is expected (e.g. with a generic data structure). You might want to access constants (e.g. `Integer.MAX_VALUE`) or class methods defined by the wrapper class (e.g. `Integer.`*`parseInt`*`(String s)`).

Sometimes the compiler wraps a primitive automatically (known as *boxing*); sometimes it unwraps automatically (known as *unboxing*).[34] Sometimes you need to do it explicitly, as above.

Make sure you familiarise yourself with the methods provided by the wrapper classes for the primitive types. You will find them useful in many contexts.

Specify `Watch` structure in the constructor argument

Consider the state of our remaining "interesting" watch classes, `Watch2` and `Watch3`. The only things left in the concrete subclasses are the constructors and the `tick()` method. The constructors are different because the two watches use different numbers of counters, with different maximum values. The `tick()` methods are very similar. They differ for two reasons: the counter each needs to increment first is different, and `Watch2` doesn't know about `LinkedCounter`. We can rewrite Watch2 to use LinkedCounter without changing its behaviour:

```
package edu.monash.fit2099.watches;
import edu.monash.fit2099.counters.LinkedCounter;
import edu.monash.fit2099.counters.MaxCounter;

public class Watch2 extends Watch {

    private MaxCounter minutes;
    private MaxCounter hours;

    public Watch2() {

        hours = new MaxCounter(MAX_HOURS);
        this.addCounter(hours);
        minutes = new LinkedCounter(MAX_MINUTES, hours);
        this.addCounter(minutes);
    }

    public void tick() {
        minutes.increment();
    }

}
```

Notice now that the `tick()` methods are `Watch2` are now very similar. In fact, they both consist of single line that does the same thing conceptually: it increments the least significant counter of the watch. For `Watch2`, that is `minutes`, for `Watch3` it is `milliseconds`.

[33] https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html
[34] https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

In our current design, all watches have an `ArrayList` of `MaxCounter` objects. The last counter added to that list is the one we need to increment in `tick()`. We can give `Watch` a method to return the counter in this position, so that the concrete watch subclasses can increment it:

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

import java.util.ArrayList;

import edu.monash.fit2099.counters.*;

public abstract class Watch {

    static final int MAX_HOURS = 24;
    static final int MAX_MINUTES = 60;
    static final int MAX_SECONDS = 60;
    static final int MAX_MILLISECONDS = 1000;

    public abstract void tick();

    private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();

    protected void addCounter(MaxCounter newCounter) {
        counters.add(newCounter);
    }

    protected MaxCounter getLeastSignificantCounter() {
        return counters.get(counters.size() - 1);
    }

    public void display() {
        String prefix = "";
        for (MaxCounter thisCounter : counters) {
            System.out.print(prefix + thisCounter);
            prefix = ":";
        }
        System.out.println();
    }

    public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            display();
            tick();
        }
    }
}
```

Once we have done this, we can rewrite the tick methods in the subclasses as

```
    public void tick() {
        getLeastSignificantCounter().increment();
    }
```

This is now identical in `Watch2` and `Watch3`. It can thus be pulled up into the superclass `Watch`. **DRY**!

The only thing left in the concrete watch classes now is the constructors. When we look at them, they are very similar. Surely we can generalise what they are doing, and thus remove more duplicated code?

Each constructor starts by creating a `MaxCounter` with a particular maximum value. Then it creates one or more `LinkedCounter` objects, each with its own maximum value, and links it the last counter created. All we need to know to do this is the maximum values, in the correct order. We can write a constructor that takes a list of maximum values as its argument, and then do what is needed. Let's do that in `Watch3`. The new constructor is

```
    public Watch3(int[] maxValues) {

        MaxCounter lastCounter = new MaxCounter(maxValues[0]);
        this.addCounter(lastCounter);
        for (int i = 1; i < maxValues.length; i++) { // notice we start from 1, not 0
            MaxCounter thisCounter = new LinkedCounter(maxValues[i], lastCounter);
            this.addCounter(thisCounter);
            lastCounter = thisCounter; // we can assign a LinkedCounter to a MaxCounter
        }
    }
```

We can test this by (temporarily) using this constructor to create a `Watch3` in `Driver`:

```
import java.util.ArrayList;

import edu.monash.fit2099.watches.*;

public class Driver {

    public static void main(String[] args) {

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

        watches.add(new Watch1());
        watches.add(new Watch2());
        watches.add(new Watch3( new int[] {
                Watch.MAX_HOURS,
                Watch.MAX_MINUTES,
                Watch.MAX_SECONDS,
                Watch.MAX_MILLISECONDS,
        } ));

        System.out.println("##############################");
        for (Watch watch : watches) {
            System.out.println("Testing Watch: " + watch.getClass().getSimpleName());
            watch.testWatch(200);
            System.out.println("##############################");
        }
    }

}
```

Here we have used something you have not seen before: we have created an array of integers to use as the argument to the `Watch3` constructor and initialised with constants when we

constructed it.[35] We also needed to make the constants defined in the `Watch` class publically visible to do this, as `Driver` is in a different package.

Now, there is nothing specific to `Watch3` in this constructor. We can pull it up to `Watch`, and make `Watch` concrete (i.e. not abstract) – we can now create objects of type `Watch` itself. The latest version of `Watch` is

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

import java.util.ArrayList;

import edu.monash.fit2099.counters.*;

public class Watch {

    static public final int MAX_HOURS = 24;
    static public final int MAX_MINUTES = 60;
    static public final int MAX_SECONDS = 60;
    static public final int MAX_MILLISECONDS = 1000;

    private ArrayList<MaxCounter> counters = new ArrayList<MaxCounter>();

    public Watch() {
         /* needed so no-argument constructors in subclasses can call it.
          * A Watch created using this will do nothing, as it has no
          * counters.
          */
    }

    public Watch(int[] maxValues) {

        MaxCounter lastCounter = new MaxCounter(maxValues[0]);
        this.addCounter(lastCounter);
        for (int i = 1; i < maxValues.length; i++) { // notice we start from 1, not 0
            MaxCounter thisCounter = new LinkedCounter(maxValues[i], lastCounter);
            this.addCounter(thisCounter);
            lastCounter = thisCounter; // notice we can assign a LinkedCounter
                                                    // to a MaxCounter
        }
    }

    protected void addCounter(MaxCounter newCounter) {
        counters.add(newCounter);
    }


    protected MaxCounter getLeastSignificantCounter() {
        return counters.get(counters.size() - 1);
    }
```

[35] https://docs.oracle.com/javase/specs/jls/se7/html/jls-10.html - jls-10.6

```
public void display() {
        String prefix = "";
        for (MaxCounter thisCounter : counters) {
            System.out.print(prefix + thisCounter);
            prefix = ":";
        }
        System.out.println();
    }

   public void tick() {
        getLeastSignificantCounter().increment();
    }
}

   public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            display();
            tick();
        }
    }

}
```

Notice that we have had to create an no-argument constructor `Watch()` too. Java only automatically creates a default constructor when no other constructor is defined. The subclasses have no-argument constructors, and Java will complain if it can't find one in the superclass to call.[36] We have chosen to make the no-argument constructor for `Watch` that does nothing. A watch created using it will be very boring.

We can now rewrite the constructors for `Watch2` and `Watch3` to use this new `Watch(int [])` constructor. For example, `Watch2()` is now

```
    public Watch2() {
        super(new int[] {MAX_HOURS, MAX_MINUTES});
    }
```

and Watch3() is

```
    public Watch3() {
        super(new int[] {MAX_HOURS, MAX_MINUTES, MAX_SECONDS, MAX_MILLISECONDS});
    }
```

That's all that remains in those classes! We can now return `Driver` to how it was, creating its watches with

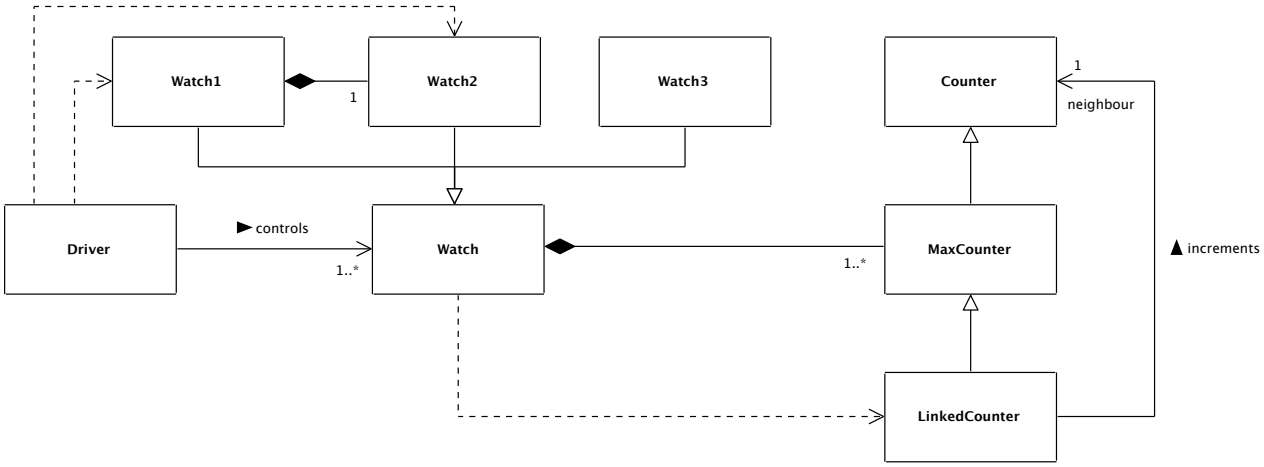
```
        watches.add(new Watch1());
        watches.add(new Watch2());
        watches.add(new Watch3());
```

We have succeeded in eliminating all the repeated code from the watch subclasses **DRY**. Importantly we have done this in such a way that nothing that depended on them had to change: `Driver` is still able to create and test its watches in exactly the same way. We should *always* do this when refactoring.

[36] https://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html

Class diagram for our current system

We haven't drawn a UML diagram of our design recently. Here is the current system:



We have supressed the package notation in this diagram to make things clearer. There are few things to note. Watch is now a *concrete* class (no longer abstract, nor an interface). There are no associations or dependencies between any of the `Watch` subclasses and the counters – none of them has any reference to the `Counter` class hierarchy any longer.

We have introduced another element of UML syntax: a special type of association to indicate *composition*. Composition is a kind of "part-of" relationship, shown in UML as an association with a black diamond at the end connected to the composite. The composite is made up of one or more components. We can read this diagram as saying that `Watch` one or more `MaxCounter` components, and `Watch1` has a single `Watch2` as a component.

Questions for next week

There are several places in our system where a client using classes could cause errors or unexpected behaviour by doing things we did not expect (and did not document!). For example, what would happen if a client tried to create a watch with one or more negative maximum values?

How could we improve our code to prevent or handle this sort of thing?