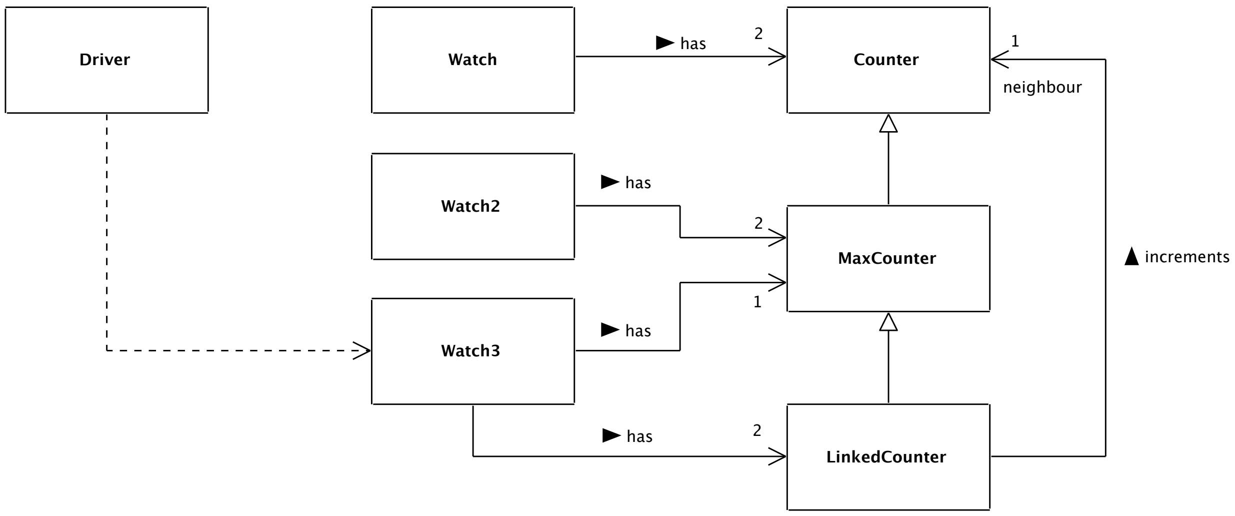
OBJECT ORIENTATION FUNDAMENTALS IN JAVA – PART 2

`Driver` and the `Watch` classes

So far, we have not talked about the `Driver` class in this document, even though we have been using it throughout. The truly complete class diagram for our system is:



We have shown a dependency between `Driver` and `Watch3`. The UML notation for a dependency is like that for an association, but with a dashed line. A dependency indicates that one class knows about another, but does not have an attribute of that type.

At this stage, the `Driver` class looks like this:

```
public class Driver {

    public static void main(String[] args) {

//          Watch myWatch = new Watch();
//          Watch2 myWatch = new Watch2();
        Watch3 myWatch = new Watch3();
        myWatch.testWatch(20000);
    }

}
```

It is a bit of a mess. We now have three kinds of watches, but we can only test one at a time. `Watch` and `Watch2` are not currently being used. Also, we have to edit the file in more than one place every time we add a new type of watch, to change the type of the variable `myWatch`, and the name of its constructor. How could we improve it?

We would like to have some way of keeping instances of all of the watch types around for testing, but to avoid duplicated code as much as possible. Remember: **DRY**!

Interfaces

Notice that all `Driver` needs to know about the `Watch` class is that it has a method `testWatch(…)` that takes an integer argument. It would be good if we could redesign `Driver` on this basis.

Java allows us to specify a type that declares what methods any class that implements that type must have, without providing any implementations for those methods. This is called an *interface*. A class that provides the implementations for the interface is said to *implement* that interface.[7]

Here is the Java code an interface Watch that meets the needs of Driver:

```
public interface Watch {

    public void testWatch(int numTicks);
}
```

Notice that the method `testWatch(int numTicks);` is simply *declared*. It is not *defined*. There are no curly braces {…} containing an implementation.

We can now refactor our code to use this interface. Before adding the interface `Watch`, we need to rename our existing `Watch` class as `Watch1`. We will then change the declarations of all our watch classes to say that they implement the interface `Watch`. For example, `Watch1` now begins:

```
public class Watch1 implements Watch {

    Counter minutes;
    Counter hours;

    public Watch1() {
    ...
```

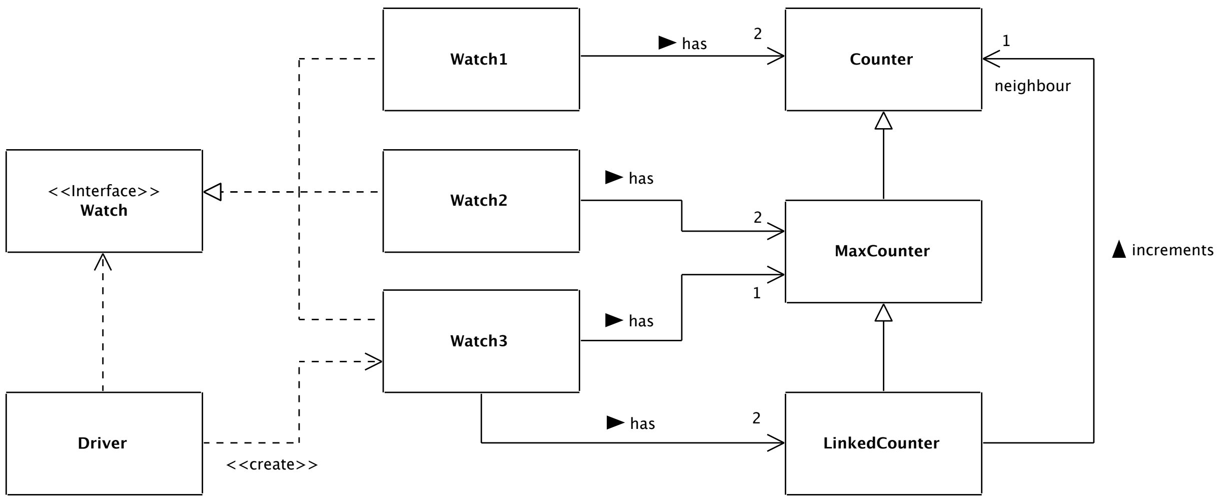
We can now change Driver so that it uses the interface type Watch for all the watches it creates:

```
public class Driver {

    public static void main(String[] args) {

//        Watch myWatch = new Watch1();
        Watch myWatch = new Watch2();
//        Watch myWatch = new Watch3();
        myWatch.testWatch(20000);
    }

}
```

Note that `Driver` no longer has any attributes of the specific watch types. `myWatch` is of type `Watch`. It still knows about the concrete watch types though, as it has to create objects of those types.

[7] Note that this is a common OO construct. It appears in many languages, sometimes with a different name. For example, in C++ one can create a class containing only *pure virtual functions* that has the same purpose. This is often called an "interface", even though there is no C++ keyword for the concept.

The class diagram now looks like this:



Some new UML notation has been introduced here. The class watch has the stereotype <<interface>>. Stereotypes allow existing UML elements (here the notation for a class) to be specialised. The relationship between the concrete watch types (`Watch1`, `Watch2`, and `Watch3`) and `Watch` is similar to the notation we saw for inheritance, but with a dashed line. This is the notation to indicate that these classes *implement* the interface `Watch`.[8]

The dependency between `Driver` and `Watch3` has been given the stereotype <<creates>> to indicate how `Driver` knows about `Watch3`.[9] Note that dependencies are often shown without stereotypes.

We are heading in the right direction, but we still need a way of testing all these different types of watches in an efficient manner. Of course we could simply copy-and-paste our code to create one of each type in turn and test it, but remember **DRY**!

Arrays

The array is perhaps the most fundamental data structure in the history of programming languages. A basic array contains a fixed number of items of the same type, in a fixed order.[10] Almost all programming languages have arrays built-in. Java does too.

[8] This is often called a "realizes" relationship. We can say, for example, that `Watch1` *realizes* the interface `Watch`.

[9] There are several other ways a class can depend on another in the absence of an attribute of that type. For example, it could have a local variable or a method argument of that type. Can you think of any other ways?

[10] See https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

We can use an array to contain references to each of our watches by redesigning `Driver` as follows:

```
public class Driver {

    public static void main(String[] args) {

        Watch[] watches = new Watch[3];

        watches[0] = new Watch1();
        watches[1] = new Watch2();
        watches[2] = new Watch3();

        System.out.println("##############################");
        for (int i = 0; i < watches.length; i++) {
            System.out.println("Testing Watch" + (i+1));
            watches[i].testWatch(200);
            System.out.println("##############################");
        }
    }

}
```

The line

```
        Watch[] watches = new Watch[3];
```

Declares a variable watches of type `Watch[]` – an array of items of type `Watch`. We initialize the variable on the same line by creating a `Watch` array of size 3. In Java, the size of an array is fixed once it has been created.

Next we create three watches, of types `Watch1`, `Watch2`, and `Watch2`, and store them in the array.

```
        watches[0] = new Watch1();
        watches[1] = new Watch2();
        watches[2] = new Watch3();
```

Notice that the array index starts at `0`. This is the convention in many computer languages (particularly those in the C family). Valid indices are thus in the range `0..2`.

Notice that we can store watches of different types in the array. This is because the array is able to contain things of type `Watch`, and all three concrete watch classes implement the interface `Watch`. The are thus all of type `Watch` (as well as of their concrete subclass). For example, an object instantiating class `Watch1` is of type `Watch1`, and is *also* of type `Watch`.

We can now use a for loop to iterate over all our watches and test them.

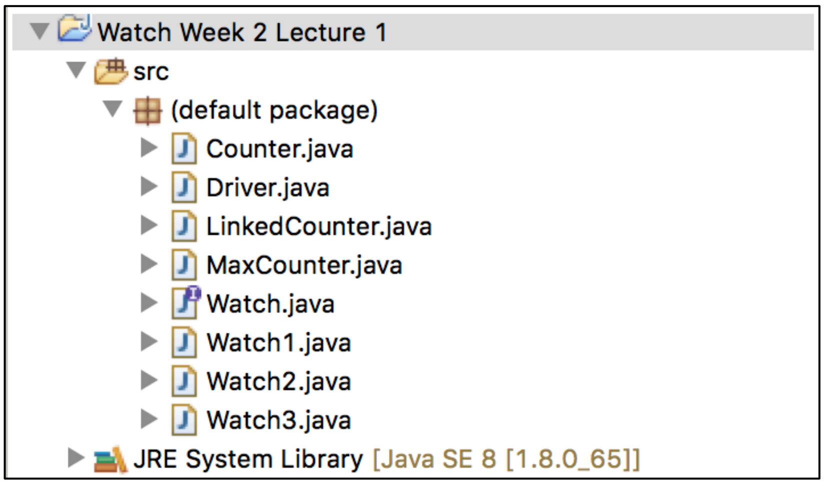
```
        System.out.println("##############################");
        for (int i = 0; i < watches.length; i++) {
            System.out.println("Testing Watch" + (i+1));
            watches[i].testWatch(200);
            System.out.println("##############################");
        }
```

Nothing in this loop knows anything about the concrete watch subclasses.

Notice that we don't use a literal to specify how many iterations the loop should have. We could have used a literal integer 3 in the end condition, but that would have needed to be updated whenever we changed the size of the array, meaning changes in multiple places. We always want to avoid this. Thankfully Java provides a mechanism to avoid this: arrays know their own size, and we can access this using the `length` member. We did this by using `watches.length` instead of 3. Always do this.

Packages

If we look at the files now in our `src` folder, we see that we have several different kinds of watch class, and several different kinds of counter:



```
▼ Watch Week 2 Lecture 1
    ▼ src
        ▼ (default package)
            ▶ Counter.java
            ▶ Driver.java
            ▶ LinkedCounter.java
            ▶ MaxCounter.java
            ▶ Watch.java
            ▶ Watch1.java
            ▶ Watch2.java
            ▶ Watch3.java
    ▶ JRE System Library [Java SE 8 [1.8.0_65]]
```

It is a good idea to group related classes together into a package. There are several reasons for doing this:

- It makes it easier to find related classes
- It gives us a mechanism to prevent name clashes. For example, the name `Counter` is quite common. It is likely that someone else somewhere has used that name. Placing our counter classes in a package with a very specific name means that we can distinguish them from any other classes elsewhere with the same class names.
- Often related classes necessarily depend on each other. When we can't eliminate dependencies, we want to place the elements that must depend on each other together inside an encapsulation boundary.

The final point above is related to a design principle that you will hear again and again in this unit, throughout your studies, and no doubt later in life:

Reduce dependencies as much as possible

This has the related principle:

Group elements that must depend on each other together inside an encapsulation boundary

We have already come across these ideas when discussing classes: the attributes and methods that implement an abstract data type often necessarily depend on each other. We thus group them together inside the encapsulation boundary of a class, and keep as much as possible of them private. Likewise, at the next level, classes that must depend on each other can be grouped together inside the encapsulation boundary of a package. We can also think of this at the level below, that of methods – instructions and variables that must be related to each other because they interact to implement some behaviour belong together inside the encapsulation boundary of a method.

The next related design principle is

Minimize dependencies that cross encapsulation boundaries

We can do this by using language constructs that make it *impossible* to create dependencies that cross boundaries. For example, in Java we can declare attributes and operations to be private to a class. If no access level modifier is used, the default visibility level in Java is *package-private*.[11]

It is a good habit to declare everything member of a class as private when you first create it. The decision to make a member public is a decision about the interface of the class exposed to the rest of the world, and should always be carefully considered. One reason for this is that it can be hard to change once it is done. As soon as something is visible, other programmers can write code that depends on it. If you then change it, you break other systems.

Another related design principle is

Declare things in the tightest possible scope

The tighter the scope in which something is visible, the less risk there is that something can depend on it, and thus the less the risk that it will be a future point of failure. So, for example: declare local variables in the loops where they are used, rather than at the beginning of a method; use local variables rather than attributes whenever possible; keep members that implement an abstraction together inside a class; keep classes that work together inside packages, and so on.

Packages in Java

Package Names

One of the motivations for using packages is to reduce the chance of name clashes. For this to work, it is necessary that different people or organisations use different package names. In Java, the convention is to use the reversed domain name of the organisation as the prefix for the package name, and then to rely on the organisation to avoid name clashes below that level.

For example, the Apache Foundation (https://apache.org) has published many java packages and libraries for public use – the Apache "commons".[12] One of these provides tools for reading CSV files[13]. The package name is `org.apache.commons.csv`. Notice that this starts with the reversed domain name of the organisation, `org.apache`, then comes a top-level package name within that organisation – `commons`, and then a specific package name – `csv`.

[11] See https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
[12] https://en.wikipedia.org/wiki/Commons
[13] Comma Separated Values – a common way of representing spreadsheet-style data in a plain text file.

Packages for Counters and Watches

We are at Monash University, which has the domain name monash.edu. We will thus use the prefix `edu.monash` for our packages. To avoid conflict with other units, we will use `fit2099` as the next part of our package names. Finally we will use the specific package names `counters` and `watches` for the two packages we need:

- `edu.monash.fit2099.counters`
- `edu.monash.fit2099.watches`

To create a package in Java, we declare that a class is a member of the package at the start of the class. At the start of the `Counter` class, for example, we have:

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

public class Counter {

    private int value = 0;

    public void reset() {
        value = 0;
    }
...
```

For `Watch` we have:

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

public interface Watch {

    public void testWatch(int numTicks);
}
```

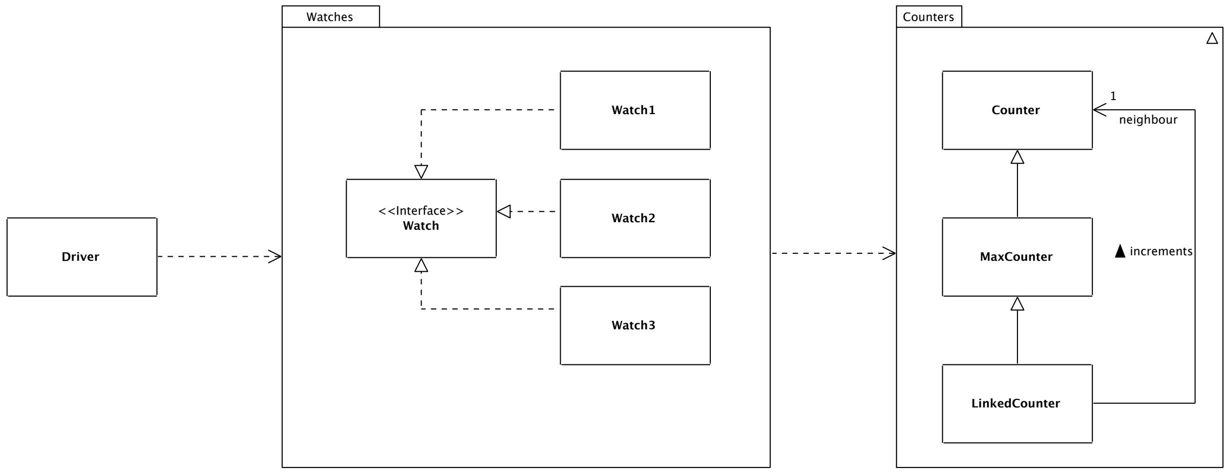
To indicate that a class makes use of classes from a package, we use an `import` declaration at the start of the class. For example, `Driver` has:

```java
import edu.monash.fit2099.watches.*;

public class Driver {

    public static void main(String[] args) {

        Watch[] watches = new Watch[3];
...
```

Notice the * above. That indicates that `Driver` is importing everything in the `edu.monash.fit2099.watches` package.

The easiest way to make all this happen in Eclipse is to first use the File menu to create the new packages. Then select the classes that belong in a package, and use the refactoring tools to move those classes to the desired package. This approach avoids much typing and many possible typos.

We can now redraw the UML diagram representing our system at a higher level, using the UML notation for a package. A package is depicted as a folder, with a tab. The name of the package can be in the tab (as below), or at the top of the main package body. The classes contained in the package can optionally be show (as below).



Abstract Classes

When we consider the state of our design, we notice that there is now a lot of repeated code in our watch classes. It's time to think about refactoring our design to eliminate it – **DRY**!

Look at the `testWatch(…)` method in all three watch classes. In all cases, inside the `for` loop, it does what is necessary to display the current state of the watch, and then calls `tick()`. Let's refactor by moving the code responsible for displaying the watch state into a separate method called `display()` in all the watch classes. Using `Watch1` as an example, we now have:

```java
package edu.monash.fit2099.watches;
import edu.monash.fit2099.counters.Counter;

public class Watch1 implements Watch {

    Counter minutes;
    Counter hours;

    public Watch1() {

        hours = new Counter();
        minutes = new Counter();
    }

    public void tick() {
        minutes.increment();
        if (minutes.getValue() == 60) {
            minutes.reset();
            hours.increment();
            if (hours.getValue() == 24) {
                hours.reset();
            }
        }
    }

    public void display() {
        System.out.println(
                String.format("%02d", hours.getValue())
                + ":"
                + String.format("%02d", minutes.getValue())
        );
    }

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

}
```

If we now look at the `testWatch(…)` method in each of the watch classes, we notice that it is identical in *all* of them! This is definitely an opportunity to remove some duplication.

An abstract Watch class

We have already seen the notion of an *interface* in Java – a special kind of class that specifies the methods that all the classes that implement it must have, but which provides no method implementations. `Watch` is currently an interface. We have also seen examples of classes that inherit method implementations from their superclasses and override some of them, as in the `Counter` class hierarchy. Sometimes we have a situation where we want something in between these situations: we know methods that all the subclasses should have, and can provide implementations for some of them, but not all. In that case, we can use an *abstract* class.

An abstract class has at least one *abstract method*. An abstract method is just like a method in an interface – there is a declaration of the method signature, but no implementation. Unlike an interface, an abstract class can also contain concrete methods, which do have implementations.[14]

We currently have a `testWatch(…)` method that is identical in all the classes that implement `Watch`. By changing `Watch` to be an abstract class, we can *pull up* that abstract method into the superclass and thus eliminate the repeated code. We also need to change the declarations of `tick()` and `display()` in `Watch` to declare them as abstract methods. The result is this abstract `Watch` class:

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

public abstract class Watch {

    public abstract void tick();

    public abstract void display();

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

}
```

We can then delete the `testWatch(…)` method from all the subclasses – goodbye duplicated code!

We can still do better though. Consider the `display()` methods in the watch subclasses. They are identical in `Watch1` and `Watch2`. The version in `Watch3` is almost the same, but slightly different, because it has a different number of counters. How could redesign our system so that they are all the same, and could thus be moved up into the abstract `Watch` class?

What we need is a data structure to contain our counters. We have already seen `array`, but that has several drawbacks. For one, its size is fixed once the array is declared. We would like to have a data structure that can contain a variable number of counters, so we can build different kinds of watches.

[14] A class can *implement* multiple interfaces, but only *extend* one superclass (abstract or not). Can you think of reasons why this language design decision might have been made?

The Java platform provides libraries of classes to address many commonly encountered programming needs. The Java Collections Framework provides classes for many standard data structures. Here we will use the `ArrayList` class. An `ArrayList` provides an implementation of the Java `List` interface. For now, we can think of it as a resizable array.

We will give our abstract `Watch` class an attribute of type `ArrayList<Counter>` to store our counters. `ArrayList` is an example of a generic type (sometimes called a parameterized type). We specify what type of class this particular `ArrayList` can contain by placing the type in the <…> brackets after the basic type `ArrayList`. By specifying `Counter`, we indicate that this `ArrayList` can contain `Counter` objects (and thus all subtypes of `Counter` too).

We will also give `Watch` a constructor `Watch()` that creates the `ArrayList`, and a method `addCounter(…)` that adds a counter to the end of the list. Note that the visibility of both is `protected`. This means that they are visible in their package, *as well as the subclasses* of `Watch`, but not elsewhere. We will also rewrite `display()` to use this list of counters.[15]

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

import java.util.ArrayList;
import edu.monash.fit2099.counters.*;

public abstract class Watch {

    private ArrayList<Counter> counters; // counters MUST be added to the list in order
of most significant to least significant

    protected Watch() { // note that an abstract class can still have a constructor,
which will be called before any concrete subclass constructor.
        counters = new ArrayList<Counter>();
    }

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

    public abstract void tick();

    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();
        }
    }

}
```

[15] *Question: can you see how the "prefix trick" works is* `display()`*?*

The rewritten version of `display()` uses a kind of `for` statement that we have not seen before:

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

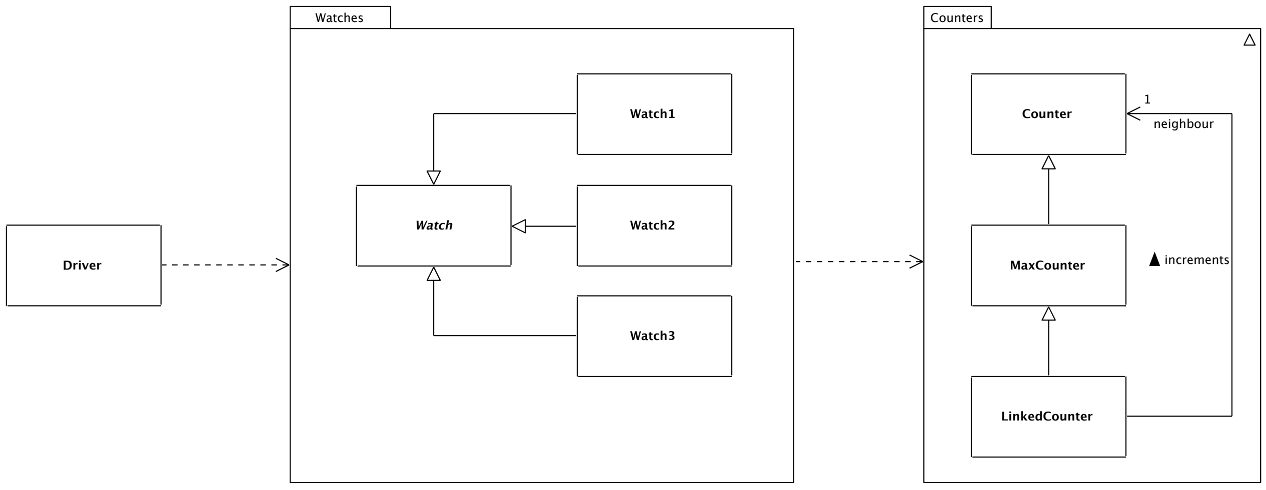
This is sometimes called an *enhanced for* statement. It is specifically designed for iterating over containers. It lets us specify a local variable (here `thisCounter`) of the type of the things in the container (here `Counter`) that will become a reference to each item in the container (here `counters`) in sequence.

Notice that our design requires that counters are added to the list in the right order (most significant first). This is done by the rewritten constructors in the watch subclasses, e.g. in `Watch3` we have:

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

public class Watch3 extends Watch {

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

    public Watch3() {
        hours = new MaxCounter(24);
        this.addCounter(hours);
        minutes = new LinkedCounter(60, hours);
        this.addCounter(minutes);
        seconds = new LinkedCounter(60, minutes);
        this.addCounter(seconds);
    }

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

}
```

We have succeeded in eliminating the duplicated code that was in the `display()` methods in all the various kinds of watch.

The UML diagram for our latest design is:



There is one new piece of UML notation here. `Watch` is now an abstract class. This is indicated by showing the class name in italics.[16]

Questions for next week

How could we continue to refactor our design to:

- remove the need for `Watch` subclasses altogether?
- fix the problem with the fixed width of counters that is still assumed by the format string in the `display()` method of `Watch`?

[16] When drawing UML by hand, I find it hard to write in italics in a way that I, or anyone else, can reliably recognize. I thus use the stereotype <<abstract>> in hand-drawn UML.