OBJECT ORIENTATION FUNDAMENTALS IN JAVA – PART 1

Introduction

In this document, we are going to use examples built on the basic idea of a *watch*. Fundamentally, a watch consists of something that ticks, and something that counts those ticks — a *counter*. Those counters are often connected to other counters that count how often a counter has reached its maximum value. For example a watch may count ticks of one second. When 60 seconds have been counted, a minutes counter is incremented, when minutes reaches 60, an hours counter is incremented, when hours reaches 24, a date counter is incremented, and so on.

We will use this example as a means of introducing various fundamental design principles, and the fundamental concepts of object-oriented design. We will show how those principles and concepts can be implemented in Java.

Classes

A class provides an implementation of an abstract data type. A class has a set of *members*. Members may be:

- **fields** (a.k.a. attributes)
  - These store the *state* of the objects that instantiate the class
- **methods** (a.k.a. operations)
  - These specify the *behaviour* of the objects – the messages it can receive. They can tell the object to do something (a command), or ask it to answer a question (a query).

In Java, a class is defined by the text in a file such as `Counter.java` or `Watch.java`. A class consists of:

- a class declaration, that specifies the name of the class, as well as:
  - its visibility: `public`, `private`, or `protected`
  - any class in inherits from: `extends`
  - any interfaces that it `implements`
- clauses defining the various members of the class, with their
  - type: `int`, `void`, `double`, `SomeClass`, etc.
  - visibility
  - optionally, initial value for fields
  - definition for methods
    - the code that specifies what the method does

A `Counter` class in Java

First we need a class the represent a counter. A counter needs to know its current value, and we need to be able to tell it to increment that value, reset the value to zero, and (for completeness) to decrement its value.

```
public class Counter {

    private int value = 0;

    public void reset() {
        value = 0;
    }

    public void decrement() {
        value--;
    }

    public void increment() {
        value++;
    }

    public int getValue() {
        return value;
    }
}
```

The members of `Counter`

The member `value` is simply declared as being of type `int`, with no associated algorithm: it is a *field*, and is initialised to the value of 0. All other members have a clause of the form

```
public type name() {
  // instructions go here
}
```

which defines some algorithm. This indicates that the feature is a *method*.

Type in Java

Every variable and method in Java has a *type*. There are three kinds of variable:

- local variables
  - declared within the body of a method
- instance variables (non-static fields)
  - i.e. non-static attributes, that can have different values for every object that is an instance of a class
- class variables (static fields)
  - i.e. static attributes, for which one value is shared by all instances of a class

The type of the variable must be specified when the variable is declared. This allows compile-time type checking. The type of a variable determines which operations can be applied to it, and what can be assigned to it. For example

```
int num = "Frog";
```

will not compile. You can't assign a string to an integer.

Benefit of compile-time type checking

In non-statically typed languages, such as Python, you can do things like this:

```
class Duck:
    def fly(self):
        print("Duck flying")

class Butterfly:
    def fly(self):
        print("Butterfly flying")

class Whale:
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

animals = [Duck(), Butterfly(), Whale()]

while 1:
    choice = input("Enter 1, 2, or 3: ")
    if (choice >= 1) and (choice <= 3):
        lift_off(animals[choice - 1])
    else:
        print("Invalid choice entered")
```

This Python program first specifies three classes, `Duck`, `Butterfly`, and `Whale`. `Duck` and `Butterfly` have a method `fly()`, and `Whale` has a method `swim()`. It then defines a method `lift_off(entity)`, which calls a method `fly()` on the argument it is passed, `entity`. Then it creates an array `animals` with a `Duck` object in position 0, a `Butterfly` object in position 1, and a `Whale` object in position 2.

Finally it enters an endless loop, in which the user is prompted to enter a number (1, 2, or 3). If a valid number is entered, the method `lift_off(…)` is called, with the element of the array `animals` at position `choice – 1` as its argument.

This program compiles and runs… *until it doesn't*.

Python happily compiles this program, and it runs without problem so long as the user keeps entering 1 or 2. That could go on indefinitely. One day, however, when the user enters a 3, it crashes:

```
Enter 1, 2, or 3: 3
Traceback (most recent call last):
  File "DuckTyping.py", line 21, in <module>
    lift_off(animals[choice - 1])
  File "DuckTyping.py", line 14, in lift_off
    entity.fly()
AttributeError: Whale instance has no attribute 'fly'
```

What has happened? The choice 3 has caused the element at position 2 in `animals` to be passed to `lift_off(…)` – the `Whale`. `Whale` has no `fly()` method. There is no type checking done to ensure that everything passed to `lift_off(…)` has a `fly()` method, so the result is a run-time failure.

In a small program like this, it is easy to see what is going on, and even to realise quickly that there is a problem and that the program will crash for some inputs. Imagine, however, a program with many thousands of lines of code spread over hundreds of files, developed by many different people. It can be very difficult to find such errors, and failure to do so means that one day, at an unpredictable time, a run-time error will crash the program after it has been deployed.

For example, one day some programmer adds a `Whale` element to the array `animals`. A `Whale` is an animal, right? They don't know that somewhere else in the program there is a method that one day will be passed an element of that array, and that it will expect that thing to have a `fly()` method. Perhaps the array should have been called `flying_animals`? That would certainly be better, but it still doesn't enforce anything. Also, the method `lift_off()` can't know where its argument comes from – it doesn't have to be an element of some array.

As we will see in the coming weeks, it is impossible to compile code with this sort of problem in Java. Java is strictly typed. Types are checked at compile-time, and you can never call a method on a variable of a type that doesn't have that method.

Catching errors early is good.

Java Primitive Types

Java has eight built-in *primitive data types*:

- `byte`
  - an 8-bit signed two's complement integer
- `short`
  - a 16-bit signed two's complement integer
- `int`
  - a 32-bit signed two's complement integer
- `long`
  - a 64-bit two's complement integer
- `float`
  - a single-precision 32-bit IEEE 754 floating point
- `double`
  - a double-precision 64-bit IEEE 754 floating point
- `Boolean`
  - a data type that has only two possible values: `true` and `false`
- `char`
  - a single 16-bit Unicode character

Variables of primitive types simply store values.[1]

Java Non-Primitive Types

Non-primitive types can be created by defining *classes* and *interfaces.* Java's built-in array type is also non-primitive. Variables of non-primitive types store a *reference* to an object of that type. They are thus also known as *reference types*. You will use many reference types defined by classes in the libraries that make up the Java platform.

[1] See https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html for further details.

A `Watch` class in Java

Now we need a class that represents a watch. Our first watch will make use of two `Counter` objects: one to count minutes, and another to count hours.

```
public class Watch {

    private Counter minutes;
    private Counter hours;

    public Watch() {

        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 testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            System.out.println(
                    String.format("%02d", hours.getValue())
                    + ":"
                    + String.format("%02d", minutes.getValue())
            );
            tick();
        }
    }
}
```

Watch has five members:

- Two fields: `hours`, `minutes`, both of type `Counter`
- Two methods: `tick()`, `testWatch(int numTicks)`
- One constructor: `Watch()`

The Constructor

A constructor lets us specify what initialization should be done when a new object of the given class is created. It has the same name as the class for which it is a constructor.

The constructor `Watch()` first (implicitly) creates a new `Watch` object. It then creates two new `Counter` objects, and assigns references to them to fields `minutes` and `hours` of the newly created `Watch` object. f this were not done, `minutes` and `hours` would still be void references, i.e. not attached to any object.

A constructor is the only sort of method that does not have its type explicitly specified. Its return type can be only one thing: an object of the class for which it is a constructor. `Watch()` returns a reference to the `Watch` object has created and initialized.

A class can have multiple constructors, with different signatures, i.e. different types and/or numbers of parameters.

Every object created during the execution of an OO system has a unique identity, independent of the object's state as defined by the values of its attributes. In particular, two objects with different identities may have identical attribute values. Conversely, the attribute values of a certain object may change during the execution of a system, but this does not affect the object's identity.

Creating Objects

The line

```
private Counter minutes;
```

declares a field `minutes` of type `Counter`. It does **not** create a `Counter` object. The line

```
minutes = new Counter();
```

in the constructor creates a new `Counter` object, and a stores a reference to it in the field `minutes`. The `Counter` object itself is initialized by calling the constructor method for `Counter`. The class `Counter` has no constructor defined, so the Java default constructor is used, which sets all attributes to zero or the equivalent for the type (i.e. `void` for references, `false` for Booleans).

These two things are often combined in a single line:

```
private Counter minutes = new Counter();
```

If we had done this, we would not need the constructor for `Watch` at this stage, and would have avoided the possibility of trying to use a void reference. In general, we should do this if we can.

The Client-Supplier Relationship

One way to use a class is to become a *client* of the class. The simplest and most common way to become a client of a class is to declare an entity (field, variable, etc.) of the corresponding type. `Watch` has two fields of type `Counter`. It is thus a client of `Counter`. `Counter` provides services to `Watch`. Each `Counter` is responsible for its own `value` field, and provides `reset()`, `increment()`, `decrement()`, and `getValue()` methods. The user, or client, of these objects need know nothing about how these methods are implemented. Indeed, in a well-designed system, the supplier should be able to change its implementation without affecting its clients.

Message-Sending

Objects communicate with each other by message-sending. We send a message to an object by invoking one of its methods. The general form of a method call

```
receiver.someMessage(arg1, arg2, ...);
```

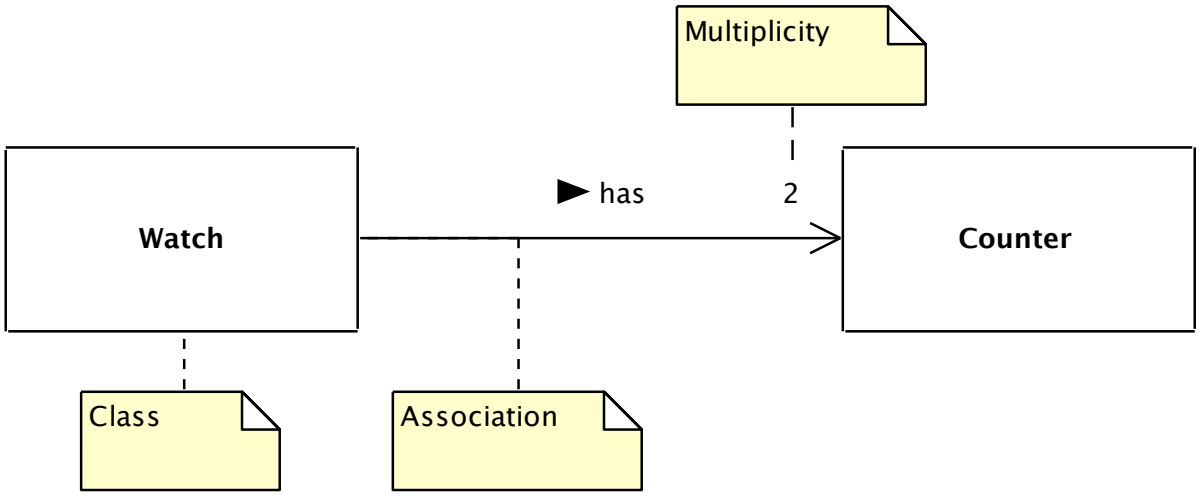
We can read this as

"Send message `someMessage` to object `receiver` with arguments `arg1`, `arg2`, …"

Watch sends `increment()`, `reset()`, and `getValue()` messages to its `Counters` at various times.

Representing the Design in UML

The design of this simple system can be represented in a UML class diagram. This allows a software engineer to see and understand the elements of the system and how they are related at a glance.



Code Smells

An experienced developer develops the ability to detect bad design and implementation practices almost automatically when viewing code. The signs of problems they are responding to are known as "code smells"[2]. Martin Fowler defines a code smell as "a surface indication that usually corresponds to a deeper problem in the system". You should aim to develop a nose that twitches when you read suspect code. Often only a page of code is enough for you to detect that smell.

The Watch code above smells. In the next few sections, we will address those smells, and show how we can refactor the code to improve the design (and thus remove the smells).

Embedded Constants

A classic code smell is seeing constants embed in the bodies of methods. This is bad for several reasons. First, if the value of the constant (a.k.a. *literal*) needs to change in future, you have to hunt for every place it occurs in the code and change it in all of them. There may also be subtle dependencies between pieces of code due to the value of the constant, even though the same value does not appear in both places. These also need to be found and fixed.

The related design principle is

Avoid excessive use of literals

Watch has literals embedded in several of its methods: the values `60` and `24` appear in `tick()`. The format strings in `testWatch()` contain the literal `"%02d"`.[3]

[2] https://martinfowler.com/bliki/CodeSmell.html

[3] The literals in `testWatch()` depend on those in `tick()`. Can you see how?

Duplicated Code

Another of the classic code smells is duplicated code. Indeed one of the fundamental principles of good design is

Don't Repeat Yourself

As software engineers, we want to say things once, and once only. This is often referred to as the DRY principle.

Why is Duplicated Code bad?

When the same code appears in multiple places, it must be maintained in multiple places. If a bug is discovered in that code, every piece of duplicate or very similar code must be checked and fixed.[4] If the requirements change and the code needs to be modified, it needs to be tracked down and changed everywhere. The same is true if you think of a better way of doing things.

`Watch` has identical or very similar code in several places. Consider the method `tick()`. It first increments the `minutes` counter, then it checks if it has reached its maximum value. If it has, it resets the `minutes` counter, then increments the `hours` counter, checks if it has reached its maximum value, and resets it if it has. The "increment-check-reset" part is the same for both counters.

We will now design new kinds of counter class that address the **DRY** code smell.

Designing a new kind of `Counter`

Think about duplicated code in the `tick()` method. Why is it there? It's because both counters need to be reset when they reach their maximum value. Is this something `Watch` should be responsible for? Wouldn't it be better if counters could automatically reset themselves? If a counter knew its maximum value, it could do so. We would need to change the way the `increment()` method works though. So we need a new kind of counter, that has an extra field, and a modified method. Everything else can stay the same though.

If we just created a new class by copying `Counter`, renaming it, and making the desired changes, we would end up with *even more* duplicated code (everything we didn't change). Fortunately, a fundamental construct of object-oriented languages lets us do what we need without doing this: inheritance.

Inheritance

Object-oriented languages allow us to create a class that inherits from another class. This new class is called a *subclass*, and the class it inherits from is its *superclass*. We also say that the subclass is a child of its superclass (parent).

The subclass has all the fields and methods that exist in the superclass. It is also possible to add new fields and methods to the subclass. Importantly, it is possible to *override* methods from the superclass – to give them a new definition. This means that the behaviour of an object of the subclass can be different from that of an object of its parent class even when the method with an identical signature is called.

[4] There *is* a bug in the duplicated code in `tick()`. Can you see it?

A MaxCounter class

We want a class that does everything a `Counter` does, but has a new field to store its maximum value, and modified `increment()` method that resets the counter when its maximum is reached. We also need a way of setting the maximum value – the best place to do this is in its constructor, since a `MaxCounter` needs to have its maximum value set correctly from the moment it is created.

```
public class MaxCounter extends Counter {

    private int max;

    public MaxCounter(int maxValue) {
        max = maxValue;
    }

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

}
```

The class `MaxCounter` inherits from the class `Counter`. The Java keyword for this is `extends`. This is an excellent choice, because a subclass should extend the behaviour of its superclass – it should be able to do everything the superclass can do, and more.[5]

`MaxCounter` has a new field `max`, declared in the line

```
    private int max;
```

This field will store the maximum value of the counter. It has a *constructor* that sets this value when a `MaxCounter` object is created.

```
   public MaxCounter(int maxValue) {
        max = maxValue;
    }
```

The constructor is the right place for this, since it doesn't make sense to have a `MaxCounter` without this value set, but we can't know it in advance, because it can be different for different `MaxCounters` (e.g. 60 for minutes and 24 for hours).

The increment() method is overridden, i.e. it is redefined.

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

[5] We will talk more about this later in the semester when we discuss *Design By Contract*.

The line `@Override` is a Java annotation. It is not necessary for the code to compile, but it makes it easier for humans to set what is going on, and also helps to prevent errors, because it allows the compiler to check that there really is a method with the same signature in a superclass that is being overridden.

The first thing the new version of the `increment()` method does is to call the version of `increment()` found in its superclass

```
super.increment();
```

The keyword `super` allows us to do this. This is a common thing for overriding methods to do: they want to do what the version in the superclass does, and then do something more. This code pattern allows us to avoid duplicating the code from the method in the superclass.

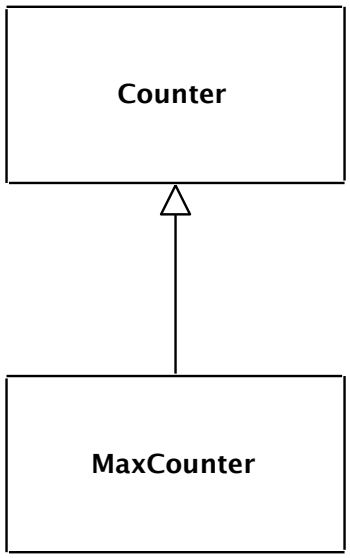
The next thing it does is to add the new desired behaviour: it checks to see if it has reached its maximum value, and resets itself if so:

```
if (getValue() == max) {
    reset();
}
```

The `reset()` message is being sent to the current object. We could write this as `this.reset()`, if it were necessary to make it explicit. The keyword `this` gives a reference to the current object (i.e. the one whose method is executing).

UML Notation for Inheritance

We show the relationship between a superclass and a subclass in UML like so:



I find it useful to remember that the arrow direction tells us where `MaxCounter` needs to look for the definitions of the fields and methods it doesn't define itself.

Watch2 – a watch that uses MaxCounter

```
public class Watch2 {

    private MaxCounter minutes;
    private MaxCounter hours;

    public Watch2() {

        hours = new MaxCounter(24);
        minutes = new MaxCounter(60);
    }

    public void tick() {
        minutes.increment();
        if (minutes.getValue() == 0) // this can only happen if the minutes counter has
just reset itself
            hours.increment();
    }

    public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            System.out.println(
                    String.format("%02d", hours.getValue())
                    + ":"
                    + String.format("%02d", minutes.getValue())
            );
            tick();
        }
    }

}
```

Watch2 has two attributes minutes and hours of type MaxCounter, instead of the original plain Counter (Counter still exists though, and our original Watch class still works). The maximum values of these counters (60 and 24) are set when the MaxCounters are created in the Watch() constructor. Much duplicated code has been removed from the tick() method.

Another design principle is at work in this improved version.

Classes should be responsible for their own properties

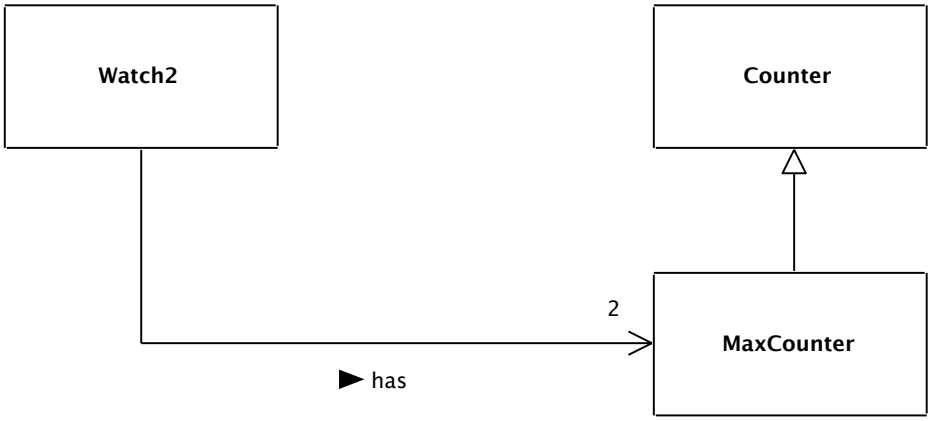
The maximum value of a counter is a property of that counter. As far as possible, classes should be in charge of themselves. Information should be stored as close as possible to where it is needed. This redesign makes that possible. Counters now know their own maximum values, and reset themselves.

Refactoring

What we have done here is called refactoring.[6] We have changed – and improved! – the design of the system without changing its behaviour. Refactoring is a good habit to get into.

[6] https://en.wikipedia.org/wiki/Code_refactoring

The `MaxCounter` System Design in UML



Linking Counters

At the moment, the `tick()` method of the watch is still responsible for checking if a counter has reset itself and incrementing the next counter if it has. If we wanted to add more counters (e.g. if we wanted to count seconds as our smallest unit of time, or even milliseconds), we would have to edit `tick()`, and it would become obvious that we still have some duplicated code there. In a "real" watch the counters are linked, so that when one reaches its maximum and resets to zero, its neighbour is prompted to increment.

A class `LinkedCounter` can be used to capture this abstraction. It will do everything a `MaxCounter` can do, and have an additional field, `neighbour`, that contains a reference to the counter that needs to increment when it is reset.

The `LinkedCounter` class

```
public class LinkedCounter extends MaxCounter {

    private Counter neighbour;

    public LinkedCounter(int maxValue, Counter linkedNeighbour) {
        super(maxValue);
        neighbour = linkedNeighbour;
    }

    @Override
    public void increment() {
        super.increment();
        if (getValue() == 0) { // This can only happen if increment()
                               has caused the counter to reset itself
            neighbour.increment();
        }
    }

}
```

`LinkedCounter` inherits from `MaxCounter`. It has a constructor which takes two arguments: an integer specifying the maximum value, and a `Counter` that is to be incremented whenever the counter has reset itself. Notice that this constructor makes use of the superclass constructor that takes only an integer argument.

`LinkedCounter` overrides the `increment()` method of `MaxCounter` so that it increments the neighbouring counter. Notice that the type of the new field `neighbour` is simply `Counter` – it doesn't need to know if its neighbour is a `LinkedCounter`, a `MaxCounter`, or just a regular `Counter`. They are all subclasses of `Counter` and have an `increment()` method. That is all that `LinkedCounter` needs to know.

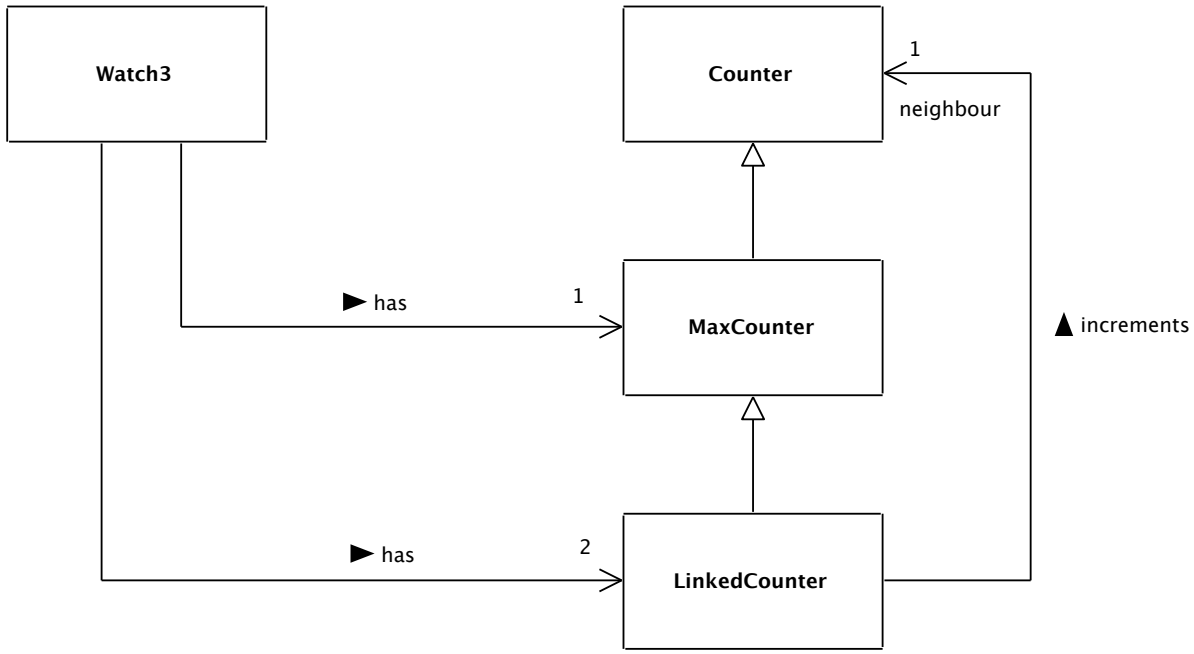
`Watch3` – a watch that uses `LinkedCounter`

```
public class Watch3 implements Watch {

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

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

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

    public void testWatch(int numTicks) {
        for (int i = 0; i < numTicks; i++) {
            System.out.println(
                    String.format("%02d", hours.getValue())
                    + ":"
                    + String.format("%02d", minutes.getValue())
                    + ":"
                    + String.format("%02d", seconds.getValue())
            );
            tick();
        }
    }

}
```

This refactoring has allowed us to eliminate the remaining duplicated code in the `tick()` method. The method `tick()` has become *very* simple. All it needs to do is increment the counter for the smallest interval of time, and everything follows from there. Counters are now responsible for incrementing their neighbours.

The constructor sets up everything necessary for the system of counters. Note that it is important that everything happens in the right order here: A counter object must have been created before a reference to it can be passed to the constructor of the `LinkedCounter` responsible for incrementing it.

The `LinkedCounter` System Design in UML



For Next Week

There is still much scope for improvement. Look at all the duplicated code in `testWatch()` involved in formatting the `Counter` values for display. Hopefully, for you, it now smells.

There is also a problem with responsibilities here. The code in `testWatch()` "knows" the width of each counter implicitly, because it uses that knowledge in the format strings (`"%02d"` is only suitable for counters with maximum values between 10 and 100). It would be easy to make a mistake here. What would happen if we added a milliseconds counter?

Also, while adding a seconds counter didn't make `tick()` more complicated, `testWatch()` has gotten worse – it now has more duplicated code. How could we redesign the system to avoid this?

If you really want to think ahead, how could we handle days, months, and years? What sorts of counters and relationships between them would be necessary?