10. Classes Part 1

Remember this from the Arrays and ArrayLists Chapter?

```
import java.util.Random;

public class BetterShape {
     public static void main(String[] args) {

          Random rnd = new Random();

          final String[] shapeNames = {"Triangle",
                    "Square", "Pentagon",
                    "Hexagon", "Heptagon",
                    "Octogon", "Nonagon", "Decagon"};
          final int[] shapeSides = {3,4,5,6,7,8,9,10};

          int sides = rnd.nextInt(shapeNames.length);

          System.out.print("The shape of the day is a ");
          System.out.println(shapeNames[sides]);
          System.out.println("It has " +
               shapeSides[sides] +" sides");
     }
}
```

There are *two* arrays containing information about shapes – but the trouble is that information about a single thing is split across two structures: the name is in one; the number of sides is in another. This is not ideal: we would prefer to group data about single 'things' together. Most programming languages have a way to do this – often called *records*, or *structures*. In Java and other object oriented languages we do this (and more) with *classes*. We're going to develop a *playing card* class in stages to show both the thinking behind classes, and what they can do in Java – but before we do that we're going to talk about the idea of a class.

Obviously we've seen classes before – the first line of every program has started:

```
public class SomeName {
```

But we've only ever had one – most 'serious' programs contain several (usually many).

Classes, Objects – and Shops

A *class* is a key concept in object-oriented programming – and so is an *object*. Briefly, a *class* is a 'model' or 'template' which contains code and a *description* of the data that code works with. An *object* is an instance of a class – it's a 'real' thing (as much as software can be 'real') and includes actual data – not just a description. This is probably not very clear – so here's an example:

(these are all public domain images BTW). This person is holding an actual real Galaxy S7 – it's an actual *object* – it has actual memory, and contains actual data and applications. But clearly it's not the only one – there are many S7s and there is a set of computer design files (obviously not public) that tell you how to make one, or as many as you want, if you have the right equipment. The design files are the *class* – the specification of the actual objects which are the phones themselves. Each phone is physically near enough identical – but they differ in terms of the actual data they contain, and the installed apps.

You can apply this analogy to more or less any mass-produced item – but we're going to apply it to Lidl.

If you've been to more than one Lidl, you know they are also more or less identical – even in different countries (OK, there *are* minor local differences, but they are all very close to the same template).

Once, again, the 'standard' layout of a Lidl – both the exterior design and the interior layout is the *class*. An *actual* Lidl you can go into and buy things from is an *object*.

Creating an actual Lidl 'object' from the 'class' description of a Lidl (or doing the same thing with a phone) is very complex – you have to actually build one. In software, using Java, we just have to use the keyword `new` – if you look back over the programs you've written, you will see you've done that a lot. Every time you've written '`new`' you've been making an object from a class. So in this line:

```
Scanner in = new Scanner(System.in);
```

You've made an object called `in` of class `Scanner`.

We said that a class is code and a *description* of data – not the *actual* data itself. When we first create a Lidl, it'll look a bit like this inside:

(this isn't a Lidl – I couldn't find a picture of a completely empty one…). Just like a new shop when it's built, we have to 'stock' objects with data. Slightly unlike a shop, it's possible to 'pre-load' an object when you make it. So when we write:

```
Scanner in = new Scanner(System.in);
```

We are 'pre-loading' the Scanner with the 'data' `System.in` – which represents the *keyboard*; so this says "read from the keyboard" – you can actually put other things here, though we don't in this module. However, consider this:

```
ArrayList<String> list = new ArrayList<>();
```

Here we've created an object called list that's an instance of the class `ArrayList` – this doesn't contain any data yet (although it 'knows' it will store strings). We have to add it in later, by calling methods:

```
list.add("lemon");
```

which adds a new data item – the string "lemon" to the `ArrayList` called `list`.

Encapsulation – and Shops

Encapsulation in Computer Science is 'hiding' things you don't need to 'know' (or at least don't want to think about at that point) – and we can explain this with shops too:

When you first go into a supermarket, you often need to spend time hunting around for the things you want. However, as you get used to the layout, you get quicker because you know where things are. But if they change the shop layout, you need to work out where things are again. (Many shops do this –

allegedly so you have to look for things you want, and you come across things you didn't know you wanted, and buy more things…).

However, there are shops where this doesn't happen – two kinds in fact.

This is a *trade counter*:

You don't browse around – you ask for what you want and they go and get it from some mysterious warehouse out the back. You have no idea how the stock is stored. ***And if they change the layout you neither know nor care!***

The other type is more obvious – it's an *online shop*:

Incidentally, the Swansea warehouse looks like this – again, you have no idea, and nor do you care, where things are unless you work there.

Traditional programming is like a traditional shop – the details of how the data is structured; how things work – is all completely open. You build your data structures, and you write code to directly deal with it. If anyone else needs to work with your data, they also deal with it directly. This has two disadvantages:

- Anyone working with your data structure needs to understand it.
- If you change it, you'll break their code.

You may think that only you are writing code for your data – *but our experience shows this is commonly not true.* Data and code are widely shared and you are not very likely to be the sole person writing or dealing with a set of data or a load of code.

Object oriented programming is much more like online shopping (or a trade counter). You *hide* the way your data is structured – but you let people access it by an *interface* that you design. This interface is made up of a set of *methods* that you write (as well as a couple of other things*). Provided you keep the methods the same, you can change the way you structure your data (for example, if you think of improvements) however you want.* And when we say 'hide' your data we don't mean "just don't tell people how it works" – we mean "build a fence around it and stop them getting in". (In fact it's more like a chain fence – you can *see though* because we don't mind people knowing how it works. *But you still can't get in*, and you still have to use the methods that are provided for you).

Advanced Aside: Changing the Methods/Interface

You may think that sometimes you will want to change the methods that interact with your system that can be accessed by others – and you'd be right. But there's a process for doing this. Normally it goes like this:

- Provide the new updated ones you want them to use, and encourage them in the documentation. But keep the old ones as well
- Mark the ones you don't want them to use as *deprecated* – you can do this in your code as we'll see later. This means then when they compile

their code, they get warned (and the documentation will tell them what they should be doing instead).
- Finally – usually after several years at least – take the old stuff out.

Applying These Ideas in Practice

Over the rest of this chapter (and the next) we're going to show how to apply these ideas in practice to code. We are going to do this in stages.

- We will start with showing how to group data together.
- How to write methods that interact with our data.
- How to prevent anyone directly accessing our data.
- How to customize how our objects, and data, are created in the first place.

The First Card Class

We're going to build a playing card class that includes a *value*, a *name* and a *suite*. Values will range from 1 to 10; names will be the traditional names of playing cards; and suites will be one of 'spades', 'clubs', 'hearts', 'diamonds'. Here's the code of the first version.

```
public class Card1 {
     String name;
     String suite;
     int value;
}
```

All we have done is create a class but instead of putting code into it we've just put variables. Each variable represents one piece of the data associated with a playing card. So the `Card1` *class* is the 'template' for a playing card, and we need to create card *objects* of this class, then put in actual data. How do we do this? Here's a main method:

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

          Card1 card = new Card1();

          card.name = "Ace";
          card.value = 1;
          card.suite = "Spades";

          System.out.println("Card: " + card.name + "
               of " + card.suite);
          System.out.println("Value: " + card.value);

          //Now make another one
          Card1 anotherCard = new Card1();
          anotherCard.name = "King";
          anotherCard.value = 10;
          anotherCard.suite = "Hearts";
          System.out.println("Another Card: " +
               anotherCard.name + " of " +
               anotherCard.suite);
     }
}
```

The first line:

```
Card1 card = new Card1();
```

creates a new `Card1` *object*.

KEY POINT: Creating Objects from Classes

In general, when we create a class we write something like this:
`ClassName objectName = new ClassName(…);`

where we may pass in some parameters (in place of the …) between the brackets after `ClassName`. When we do this we are calling a piece of code called a *constructor* – more about these later: for now, the `Card1` constructor doesn't have any parameters.
There are other ways of creating objects but this is the most common and basic. It should be clear that we've seen this many times already. For example:

```
Scanner in = new Scanner(System.in);

Random rnd = new Random();

ArrayList<String> list = new ArrayList<>();
```

creates new *objects* of the `Scanner, Random` and `ArrayList` *classes*. (If you're wondering about how `<String>` fits in, you can write your own

classes that do things like that, but we won't in this module – things in `<..>` are called *generics* and are a bit more advanced.)

Accessing Elements of a Class

If you look at the main method above you will see the lines:

```
card.name = "Ace";
card.value = 1;
card.suite = "Spades";
```

This is the way we access elements – or *members* – of a class. The name of the *object* (`card` in this case) goes before the '.' and the name of the *member* (`name`, `value`, `suite`) go after it. We can both assign to members:

```
card.suite = "Spades";
```

and read their values:

```
System.out.println("Card: " + card.name + " of
" + card.suite);
```

which in this case prints out 'Ace of Spades'.

KEY POINT: Fields

In this case, our class only contains *variables* (we'll put other things in later). Commonly, people just use the term *variables* for these but strictly speaking in object oriented programming variables in a class like this are called *fields.*

You can of course create as many objects of a class as you want – just as you can make as many models of a car as you want; or open as many branches of Lidl as the owners want. In the example above

```
Card1 anotherCard = new Card1();
```

creates a second car object, which is initialized in the following lines.

Printing out an Object

In the example above, the line:

```
System.out.println("Card: " + card.name + " of
" + card.suite);
```

prints out 'Ace of Spades'. But this seems a bit clumsy. What happens if we just do:

```
System.out.println(card);
```

If you try this, you get something like:

```
Card2@eb42cbf
```

which seems like gibberish. Actually, it's the name of the class (`Card`), followed by '`@`' and then a *hashcode* computed from the data stored in a particular object in *hexadecimal*. This isn't very helpful, so we need to change it (and will in a bit).

Classes and Methods

As well as variables, classes can contain *methods* – and they actually come with some 'for free' that we don't have to write. One of these is called `toString()` which is called whenever you do something with a class that 'expects' a value that's a string (like printing it). You can also call this method explicitly if you want – this:

```
System.out.println(card);
```

is the same as this:

```
System.out.println(card.toString());
```

KEY POINT: Calling Methods in Objects

By now, the notation '`objectName.method(…)`' should be obvious to you – it's exactly what we've been doing when we write things like:

```
in.nextLine()
```

We are calling a *method* called `nextLine()` belonging to an *object* `in` (which is an *instance* of the `Scanner` class).

Writing our Own toString

The default `toString()` method is not very useful – we don't really want to know what it tells us. However, we can write our own and replace the default one.

The Second Card Class

Here's a new version of Card with a `toString() method`

```
public class Card2 {

	String name;
	int value;
	String suite;

	//why 'public' – see later
	public String toString(){
		return (name + " of " + suite);
	}
}
```

And here's how we can use it:

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

		//As before...
		Card2 card = new Card2();

		//Same again
		card.name = "Ace";
		card.value = 1;
		card.suite = "Spades";

		//We'll leave this for now
		System.out.println("Card: " + card.name + "
			of " + card.suite);
		System.out.println("Value: " + card.value);

		//We'll skip the other card here

		//Here's our version of toString():
		System.out.println(card);
		//Suddenly not gibberish!
	}
```

Now printing out the value of `card` directly generates 'Ace of Spades' automatically – so we don't have to print it out in the old, clumsy, way. (We can, if we want, call this method explicitly – `System.out.println(card.toString());` - but Java 'knows' we want to do this if we leave off `toString()`);

Advanced Aside: Skip on First Reading if New to Programming: Inheritance and Overriding

So far we're written two classes apparently 'from scratch' – but we can actually write classes that build on, or *inherit from*, others. For example,

suppose we had a class `Car`, and wanted a new class `SportsCar` – however, most things about a `SportsCar` are just like a `Car` – we want to be able to say 'A `SportsCar` is like a `Car` with Less Seats and an Insecure Owner' (or something similar). We can do this in Java using inheritance. We're not going to do it much here but:

- As well as doing it *explicitly*, it's happening *automatically* whenever we create a class: any new class inherits from a 'base' class called `Object`. It's in this base class the original `toString()` method is defined.
- When we wrote the `toString()` method above we are actually replacing the original one – not all the methods we will write will do this, but it's something we do sometimes do. This is called *overriding*. Java has a way of telling the compiler you know you are doing this (or think you are…) by using the annotation `@Override`:

```
@Override
public String toString(){
     return (name + " of " + suite);
}
```

  You don't *have* to do this – but it has two advantages:
  - It tells the reader what you are doing and that you are *replacing* a method not *adding* one.
  - If you make a mistake and get the name wrong, so you're not actually overriding something, then the compiler will tell you.

The Third Card Class

The way we've been assigning values to our Card objects is like this:

```
Card2 card = new Card2();

card.name = "Five";
card.value = 5;
card.suite = "Spades";
```

But there are two problems with this:

- It's *redundant* – once we know the name the value follows *automatically* from that. We don't really want to have to set the name *and* the value.
- It's *error prone* – there's nothing to stop us writing:

```
card.name = "Five";
card.value = 10;
card.suite = "Lemon";
```

  or any other nonsense – because any strings (for `name` and `suite`) and integers (for `value`) are permitted by Java. We'll start with the first problem.

We're going to write a method to set the value from the name:

```
public class Card3 {

     String name;
     int value;
     String suite;

     // Here's our method
     public void setNameValue(String cardName){
          name = cardName; //sets the name
          switch(value) {
          case "Ace":
                    value = 1;
                    break;
          case "Two":
                    value = 2;
                    break;
          //others skipped to make it shorter
          case "King":
                    value = 10;
                    break;
          default:  value = -1; //An error value
                    name = "error";
                    break;
          }
     }

     // We'll leave in toString()
     public String toString(){
          return (name + " of " + suite);
     }
}
```

As an aside notice the error value of -1 and an error name of "error"– this is used whenever an illegal value for the name gets used. (This does raise a tricky question – is this a good way to handle errors? This isn't something we're going to deal with yet.)

Here's how you call it:

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

          //As before...
          Card3 card = new Card3();

          //Now we set name and value
          card.setNameValue("Ace");
          card.suite = "Spades";
```

```
        System.out.println(card.toString());
        System.out.println(card.value);
    }
```

This now means that provided we pick a legal value for the name, we *automatically* get the correct value. And if we pick an *illegal* name, we get an error value for both the name and value. Which leads on to the *suite* – which should only have one of four possible values – Spades, Diamonds, Hearts and Clubs. The next version will add a method to enforce that.

KEY POINT: Redundant Data and Being 'Reasonable'

In the example above, we are using a switch statement to set the value based on the name – we've left a lot of the switch out to shorten the code, but in reality it would be a bit long. Arguably, a bit too long – and suppose there weren't 13 possible cards but 130? It's nice when it's sensible to write code that minimizes the number of data items you need to specify, and which creates other data automatically whenever possible. But you can't always reasonably do this simply because the code would get too long – you have to use your judgement. (In fact, in this particular case, we can make the code a lot shorter – as we'll see later in the module – but this isn't always possible).

The Fourth Card Class

Instead of repeating the whole class, we'll just write the extra method we need.

```
    public void setSuite(String suite){

        if(suite.equalsIgnoreCase("Hearts") ||
            suite.equalsIgnoreCase("Diamonds") ||
            suite.equalsIgnoreCase("Clubs")  ||
            suite.equalsIgnoreCase("Spades")) {
            this.suite = suite;
        } else {
            this.suite = "error";
        }
    }
```

This sets the value of the suite to "error" if we try to call a suite anything other than Spades, Clubs, Diamonds or Hearts. Here's how we would call it:

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

        //As before...
        Card4 card = new Card4();

        //Now we set name and value
        card.setNameValue("Ace");
        card.setSuite("Spades");
```

```
        System.out.println(card);

        // Lets test the 'error handling'
        Card5 wrongCard = new Card5();

        wrongCard.setNameValue("Lemon");
        wrongCard.setSuite("Grapes");

        System.out.println(wrongCard.toString());
    }
```

The output of this will be – for the first card:

```
Ace of Spades
```

and for the second – wrong – one:

```
error of error
```

This isn't *great* error handling – but it's better than nothing and, later in the module (and in the next module, CS-115) you will learn better ways of doing it.

The Fifth Card Class

This is all great – we now have methods that enforce some rules on the data held by our class, making it harder to end up with incorrect or inconsistent data. However, there's a problem. Suppose we create a new class and call our methods like above:

```
        Card4 card = new Card4();

        card.setNameValue("Ace");
        card.setSuite("Spades");
```

There's nothing to stop us, after we've done this, from simply *overwriting* the values we used (and checked with our methods) by *directly* writing to the variables (or, remember, *fields*) in the class.

```
        card.name = "Lemon";
        card.value = 1066;
        card.suite = "Lemon";
```

This is like putting up a really secure front door but leaving all your windows open. To stop this happening, we need to *restrict access* to the *fields* `name`, `value` and `suite`. The way we do that is with the `private` keyword.

```
public class Card5 {

     private String name;
     private int value;
     private String suite;

     public void setNameValue(String cardName){
          name = cardName; //sets the name
          switch(value) {
          case “Ace”:    value = 1;
                    break;
          case “Two”:    value = 2;
                    break;
          //others skipped to make it shorter
          case “King”:   value = 10;
                    break;
          default:  value = -1; //An error value
                    name = “error”;
                    break;
          }
     }

     public void setSuite(String suite){
          if(suite.equalsIgnoreCase("Hearts") ||
               suite.equalsIgnoreCase("Diamonds") ||
               suite.equalsIgnoreCase("Clubs")  ||
               suite.equalsIgnoreCase("Spades")) {
               this.suite = suite;
          } else {
               this.suite = "error";
          }
     }

     public String toString(){
          return (name + " of " + suite);
     }
}
```

Notice we've put the keyword `private` (in bold, just to make it stand out), in front of the declarations of the variables we are using.

KEY POINT: Public, Private Etc.

By now the point of the keyword `public` should also be clearer:
Anything preceded by `public` is visible outside whatever it's in (in our case, outside the class `Card5`.
Anything preceded by `private` is not visible outside the (in this case) class – only inside.

There are actually four possible options we can have in Java

Option	Meaning	Comment
`public`	Visible outside the class of the item it is attached to – in our case, field or method	Use this whenever you want something to be accessed from outside a class
`private`	Not visible outside the class of the item it is attached to	Use this whenever you don't want something to accessed from outside a class
`protected`	Visible outside the class but only to other things in the package or to subclasses	Best not to use this since we haven't done subclasses and packages
nothing	Visible outside the class but only to other things in the package	See above

In practice, we should stick to `public` and `private` – they have meanings that make sense in terms of what we've done. Using `protected` (which basically means "a bit less public than `public`" or nothing at all (" bit less public than `protected`") doesn't really help us in this module.

Here's an example of using Card5:

```
public static void main(String[] args){
     Card6 card = new Card5();

     //Now we set name and value
     card.setNameValue("Ace");
     card.setSuite("Spades");

     System.out.println(card.toString());

     System.out.println(card.value);
     System.out.println(card.name);
     System.out.println(card.suite);
}
```

This seems straightforward enough – we have created our `Card5` class, initialized a card to be the Ace of Spades, and then printed it out using both `toString()` and by printing the `value`, `name` and `suite` directly

The trouble is it does not compile!

You get an error message something like this (exact text will depend on the name of the file you use to test your card class).

```
TestCard5.java:16: error: value has private access in
Card6
          System.out.println(card.value);
                                  ^
TestCard5.java:17: error: name has private access in
Card6
          System.out.println(card.name);
                                  ^
TestCard5.java:18: error: suite has private access in
Card6
          System.out.println(card.suite);
                                  ^
3 errors
```

This is because, of course, we made `value`, `name` and `suite` private! We can no longer (directly) access them – and that means that not only can we not *change* them, we also cannot *read* them. We can set them because `setSuite()` and `setNameValue()` are public; we can also print them as a string because `toString()` is also public. The public methods are visible from outside the class and they can access the private variables because they are all (methods and variables) in the same class. But the variables themselves are private and so we can't directly read them – the way to fix this is to add *accessor*, or *getter*, methods. These are just methods to access the values stored in private fields. In the case of `Card5` we just need to add these three:

```
public String getName() {
     return name;
}

public int getValue() {
     return value;
}

public String getSuite() {
     return suite;
}
```

These three methods are public so can be called from outside the class – so we can update the method that did not compile:

```
public static void main(String[] args){
     Card6 card = new Card5();

     //Now we set name and value
     card.setNameValue("Ace");
     card.setSuite("Spades");

     System.out.println(card.toString());

     System.out.println(card.getValue());
```

```
        System.out.println(card.getName());
        System.out.println(card.getSuite());
    }
```

Now we are calling our public getter methods we can access the values in the fields – because the methods return those values.

KEY POINT: Get/Set, Accessor/Mutator, Getter/Setter Methods

As stated above methods that return values from a class are called one of *get methods, accessor methods*, or *getter methods* – they have names that start with *get*. There is nothing special about them – they are just methods – or of using names starting with *get* – it just indicates to the reader what they do. Similarly, methods that set the value of things are called (one of) *set methods, mutator methods,* or *setter methods* – they have names starting with *set*. Again, this is not special – it's just a standard convention to make reading programs easier.

A special variation on using *get* with getter methods is when the value being returned is a `boolean`. Then the name usually starts with *is*. For example, the `ArrayList` method

```
isEmpty()
```

which returns true when an `ArrayList` is empty.

VITAL KEY POINT: Why Are We Bothering With This?

This all looks like a lot of trouble – we are creating a lot of extra methods and it seems like out code is getting longer. That's true but:

This is absolutely standard good practice in object oriented programming – you never, never, EVER let other code directly access the values of fields (variables) from outside the class they are in. Instead, you always, always, ALWAYS use methods to set/get values.

The reasons for this are:

- **Abstraction** – we are hiding details about how things work: by only letting the calling code know about the methods, it only has to know what the code does and not how it works.
- **Encapsulation** – the term for this practice is encapsulation: grouping together operations and data, and then hiding the details about how the data is actually stored or manipulated.
- **Error Checking/Prevention** – by doing this, we are putting a layer of 'protection' code around our data. In the card example, we use this code to prevent anyone using our class from setting the data in a card to illegal values – if we allowed direct access to the data, we could not stop this happening.

- **Software Reuse** – doing this helps software reuse. Provided we don't change the methods, we can update the way things work (and make it better) without affecting those using it.

Below I'm going to show you two versions of `Card5` – both do exactly the same thing; but they work in different ways. Because they have exactly the same public methods, we could change one for the other without causing anyone whose code uses it any problems at all.

Advanced Aside

If you look at other programming languages, they seems to have some other ways of doing this – e.g. *properties* in C#. They *look* different (and are often a bit shorter and more convenient), but they are really just 'disguised' methods.

Changing How Card Works

I said above that we could change how card works without affecting code that calls it. Here's an example of how. First I'm going to repeat all the code, with the public bits highlighted in bold.

```
public class Card5 {

     private String name;
     private int value;
     private String suite;

     public void setNameValue(String cardName){
          name = cardName; //sets the name
          switch(name) {
          case "Ace":    value = 1;
                         break;
          case "Two":    value = 2;
                         break;
          //others skipped to make it shorter
          case "King":   value = 10;
                         break;
          default:  value = -1; //An error value
                         name = "error";
                         break;
          }
     }

     public void setSuite(String suite){
          if(suite.equalsIgnoreCase("Hearts") ||
               suite.equalsIgnoreCase("Diamonds") ||
               suite.equalsIgnoreCase("Clubs")  ||
               suite.equalsIgnoreCase("Spades")) {
               this.suite = suite;
          } else {
               this.suite = "error";
          }
     }
```

```
    public String getName() {
        return name;
    }

    public int getValue() {
        return value;
    }

    public String getSuite() {
        return suite;
    }

    public String toString(){
        return (name + " of " + suite);
    }
}
```

We can change what we like, as long as we don't change those bits.

Next, we're going to change it so that we don't *store* the value – instead we *calculate* it from the name when `getValue()` is called.

```
public class Card5 {

    private String name;
    private String suite;

    public void setNameValue(String cardName){
        name = cardName;
    }

    public void setSuite(String suite){
        if(suite.equalsIgnoreCase("Hearts") ||
            suite.equalsIgnoreCase("Diamonds") ||
            suite.equalsIgnoreCase("Clubs")  ||
            suite.equalsIgnoreCase("Spades")) {
            this.suite = suite;
        } else {
            this.suite = "error";
        }
    }

    public String getName() {
        return name;
    }

    public int getValue() {
        int value;
        switch(name) {
        case “Ace”:    value = 1;
                  break;
        case “Two”:    value = 2;
```

```
                    break;
            //others skipped to make it shorter
            case "King":   value = 10;
                    break;
            default:  value = -1; //An error value
                    break;
            }
            return value;

      }

      public String getSuite() {
            return suite;
      }

      public String toString(){
            return (name + " of " + suite);
      }
}
```

The *public* interface is exactly the same – but now `setNameValue()` only actually sets the name; we *calculate* the value every time the `getValue()` is called (and we don't even have a private field for it anymore). Provided we call it with *legal* values, this code behaves *exactly* the same even though it works differently. (If we call it with *incorrect* values, it does something slightly different though – see if you can spot it.)

KEY POINT: Always Define a Public Interface, Keep How Your Code Works Private

This is a fundamental principle of object oriented programming and it's essential you stick to it ***always***.

public static void main(String[] args)

Hopefully now, you know a bit more about the line you've been typing at the start of every program. public static void main(String[] args) is a method which:

- is called `main`
- returns nothing (`void`)
- has an array of `Strings` as an argument
- is `public` – that is, it can be seen outside the class

Now we only need to explain `static` which we'll do in the next chapter.

Object Oriented Programming – How you need to think

It can be *very* difficult to start to think in an object oriented way – *especially* but not only if you've been writing 'normal' programs for some time. Actually some people who have less/no programming experience find it eaiser to start with. Here's three points.

- **It's about 'need to know'** – 'need to know' is normally a 'spy' term used in the context of keeping things secret. But here it's slightly different – you *can* know, it's *not* secret – and in fact the code for many classes is freely available. *But you really don't **need** to know to use it, and even if you do, you can't take advantage of knowing*. This is deliberate – we don't want programmers working out that some detail of how things work lets them do some 'clever' thing that will break in the future if that detail changes. It's like the case of a PC – provided it's locked shut, we don't care if there are windows in the side (and pretty lights) so you can see inside or not; nobody can actually mess about with the way it works.
- **Need to know applies to you** – if you're writing a program that consists of lots of classes, you need to apply the need to know principle to yourself. That is, each of your classes needs to only make public the things it needs to, even though you are writing all of them. Some branches of Lidl have a bakery – we are saying that the bakeries are run by specialized bakery staff; we don't require the 'normal' Lidl staff to run the ovens even though both Lidl and Lidl bakery 'objects' are 'owned' by Lidl. You have to 'pretend' that each class is written by a separate programmer, who needs to prevent access to data that should be private, *even if all these 'programmers' are really you.*
- **What belongs in a class, stays in a class** – if all the data related to some computation is inside a class, that computation should be too. As an example, I used to ask for a student record system as the second CS-110 coursework (a very simple one) – and it had to compute averages for students. Often code would look like this:

  ```
  //this is bad!
  int year1 = aStudent.getScoreYear(1);
  int year2 = aStudent. getScoreYear (2);
  int year3 = aStudent. getScoreYear (3);
  double average = (year1 + year2 + year3) / 3;
  ```

  But all the data related to the average is already inside a student object - there is no need to read it all out by calling `getScoreYear` () three times to then calculate. The average can be generated automatically inside the student record class – the code should look like:

  ```
  double average = aStudent.getAverage();
  ```

  Where the average is either calculated when `getAverage`() is called, based on the values of the year 1-3, of is based on some value stored inside the student record class – *and we neither need to know nor care which.*

Classes Continued

There are more important things to say about classes – and we'll continue this with a Classes Part 2 chapter.