11. Classes Part 2

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

At the end of the last chapter, we'd got to the point where we could create classes that (within limits) included error checking and did not allow access to the internal detail of how it worked. However, to create a card had to do things like this:

```
Card card = new Card();
card.setNameValue("Ace");
card.setSuite("Spades");
```

But if we look at how we created a `Scanner`, we just had to do this:

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

It looks like we should be able to create a new `Card` like this:

```
Card card = new Card("Ace", "Spades");
```

And we can – but we need to write a *constructor* first.

Constructors

Constructors are the code that gets called when we create a class – and we've already been calling them with code like this:

```
Card card = new Card();
```

because `Card()` is the constructor for the `Card` class – we did not write this constructor though: it's the *default* constructor we get if we don't provide our own.

Default Constructors

"If you cannot afford a lawyer, one will be provided for you."

All classes need a constructor, and if you don't *write* one a default no-argument constructor will be created *automatically* – this will set up the basic (internal) things needed for the object that is being created needs. So far all the classes we've written just use that one. But now we need to write our own. Constructors look like this:

```
public class SomeClass {

     public SomeClass(…) {
          …
     }
     …
}
```

At first sight the constructor looks like a method. But it does have differences:

- Constructors don't have a return type (not even void).
- Constructors have the same name as the class.

The Sixth Card Class

Here's a version of the card class with a constructor:

```
public class Card6 {

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

     // Constructor
     public Card6(String cardName, String cardSuite) {
          // Rather than repeating code in the methods to
          // set the name, value and suite we just call
          // them - but see important note below
          setNameValue(cardName);
          setSuite(cardSuite);
     }

     // setNameValue sets the name and value
     public void setNameValue(String cardName){

          … code left out to save space
     }

     // setSuite sets the suite.
     public void setSuite(String suite){

          … code left out to save space
     }

     // get methods
     public String getName() {
          return name;
     }

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

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

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

In our constructor, we've called the methods we defined already. *But notice the syntax was a bit different.* When we called them from *outside* the class, we had to put the *object name* in front (e.g. `card.setNameValue("Ace")`) but *inside* the class we didn't. This is simply because when we call a method from outside, we have to say which object we mean. For example, if we've created two `Card` objects - `card` and `lemon` - and we want to find out the suite of both of them, we need to *differentiate* between card and lemon - `card.getSuite()` and `lemon.getSuite()`. But *within* a class, we don't need to do that.

Advanced Aside – Private Constructors

Another point - we put the keyword `'public'` in front of the constructor. Given that it seems that the whole point of a constructor is that it's called from *outside* the class, then it seems there would be no point in it being `private`, so why to we have to say it's `public` - couldn't that just be assumed? In fact although constructors are very nearly always `public` *there is* a reason to make constructors `private` – and we won't go into details here (but it's useful if you want to create a fixed group of one or more *constant objects*).

Here's how we call this constructor:

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

        //Not quite as before...
        Card7 card = new Card6("Ace", "Spades");

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

        card.setSuite("Hearts");

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

Notice we can still call the method `setSuite()` which in this case will change the suite of the card from 'spades' when we created it, to 'hearts'.

Multiple Constructors

Suppose we wanted to create a card and, say, knew what name we wanted but didn't know what suite we wanted. To do this we would need a *different*

constructor – and, in fact, we can have *more than one* constructor for a class (and most classes in the standard Java library do) *provided* the arguments are different so that Java can tell them apart. (In fact, we can also have methods with the same name in a class – provided, again, that the arguments, or return type, are different so that Java can distinguish them.)

Seventh Card Class

Here's our next card class with multiple constructors:

```
public class Card7 {

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

     // Here's the constructor from Card6.
     public Card7(String cardName, String cardSuite) {
          setNameValue(cardName);
          setSuite(cardSuite);
     }

     /* This one only sets the name and value; we will
      * have to call setSuite() later to set the suite.
     */
     public Card7(String cardName) {
          setNameValue(cardName);
     }


     // Here's the 'no-argument' constructor
     public Card7 (){
     }

     // setNameValue sets the name and value
     public void setNameValue(String cardName){
          … code left out to save space }

     // setSuite sets the suite.
     public void setSuite(String suite){
          … code left out to save space }

     // get methods
     … code left out to save space
}
```

Now we have *three* constructors:

- One that creates a complete card – the same one we wrote in version 6.
- One that only creates a card with name and value – you have to set the suite later.

- One that has no arguments.

Wait… Didn't we get the one with no argument's 'for free' already?

Notice we've written a constructor in `Card7` that doesn't have any arguments.

KEY POINT: Write a Constructor, Lose the Free One

As soon as you write your *own* constructor, you no longer get the one that has no arguments automatically and have to write your own if you want one – and since it doesn't need to actually do anything that doesn't happen automatically, in this case it's empty.

```
public Card7 (){
}
```

We could though if we wanted to put some code in – for example, we could say that the 'default' card is the 'two of clubs' – for example:

```
public Card7 (){
     setNameValue("Two");
     setSuite("Clubs");
}
```

Here's an example of calling two of the constructors in Card7:

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

     //The 'normal' constructor
     Card8 card = new Card7("Ace", "Spades");

     System.out.println(card);

     //The empty one:
     Card8 emptyCard = new Card7();
     //prints 'null of null' unless we use 2nd
     //constructor above and then it
     //prints 'Two of Clubs'
     System.out.println(emptyCard);
}
```

Why Would We Ever Change a Card? Private Methods

When you think about it, playing cards are not something you would 'change' – having made one:

```
Card card = new Card("Five", "Diamonds");
```

it doesn't seem to make sense to then do this:

```
card.setSuite("Clubs");
```

because you would reasonably assume that changes like this would not be allowed. To do this, we can simply change the methods that set the suite, name and value to be private.

Eighth Card Class

Here's a new version that has private methods – marked in bold so they stand out.

```
public class Card8 {

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

     // Since we're don't allow the fields to be set
     // after an object has been created,
     // this is the only construtor that makes

     public Card8(String cardName, String cardSuite) {
          setNameValue(cardName);
          setSuite(cardSuite);
     }


     // setNameValue sets the name and value
     private void setNameValue(String cardName){
          … code left out to save space }

     // setSuite sets the suite.
     private void setSuite(String suite){
          … code left out to save space
     }

     // get methods
     … code left out to save space
}
```

We've skipped the extra constructors here and only have the one that sets all the properties of a card – in fact, because out set methods are now private, this is the *only* one that makes sense: because if we *don't* set them when we create the card, we can *never* set them because we can't write directly to the variables (fields), and we can't call the methods either because they're private – all we can do after we've created a card is *read* the values that are stored in it and not change them anymore.

Final Fields

Now that we can't change cards once we've made them, we are only actually setting the values of the fields *once* – so we should make them *final*. Telling

Java that all the fields in a class are final is a good idea (if that makes sense of course) because by doing so Java can work out that objects of that class are *immutable* – and that means it can make significant efficiency changes when it generates the executable code. This does not make sense for *all* classes – for example, we are going to look later at a `BankAccount` example and it would be unhelpful if the bank balance could not change. Also, consider `ArrayLists` – it makes no sense not to be able to change them. However, it's more common than you might think that once created, objects do not need to change.

Ninth Card Class

Here's the ninth example – we've changed the fields to be final.

```
public class Card9 {

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

     public Card9(String cardName, String cardSuite) {
          //Code to set suite, name, value left out
     }


     public int getValue() {
          return value;
     }

     public String getSuite() {
          return suite;
     }

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

Naming our Static Variables

Note that we have NOT used the 'all capitals' naming scheme for the final variables in `Card8`. That is, we have written:

```
private final String name;
```

not

```
private final String NAME:
```

That's because we normally only use the 'all capitals' naming scheme for final variables that represent 'constants' and although we can't change them, `name` etc. are not 'constants' in the traditional sense. How do we tell the difference?

Final variables that represent 'real constants' are *static* – which means it's time to explain…

Static

We've now nearly explained all the things we've been writing for weeks without explanation:

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

This is a `public` (visible everywhere) class called `SomeClass`, containing a method called `main`, also `public` (visible everywhere) which has an array of strings as an argument. However, we haven't explained the word `static`. And if you look back at the examples of methods in classes in this chapter and the previous one, the only time we've used the keyword static is for the main method. So what does it do?

Class Data vs Object Data

All the classes in this chapter and the last have included three data items, or properties – name, value and suite. And in each case, these values belong to *objects*: a particular card will have it's own values for these.
For example, in:

```
Card card  = new Card("Six", "Hearts");
```

has `value = 6`, `name = "Six"`, and `suite = "Hearts"`, while:

```
Card anotherCard = new Card("Queen", "Clubs");
```

has `value = 10`, `name = "Queen"`, and `suite = "Clubs"`.

However, in any particular pack of cards there will be, generally, a fixed number of cards – typically 52. It would be helpful (probably) to have this value available in the code – probably as a `final` variable. But this value does not 'belong' to any individual object: it's a property of the class itself and not of an object of that class.
It is for things like this that we use the static keyword – `static` means data (or methods) that don't depend on data stored in any particular object. In the Card example, we can create a static final variable that tells us how many cards are in a pack:

```
public class Card10 {

     //Here's our static field - the number of cards
     private static final int CARDS_IN_DECK = 52;
```

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

        … constructor left out to save space

        // static method to access the static field
        public static int getCardsInDeck() {
             return CARDS_IN_DECK;
        }
        … code left out to save space
}
```

How do we access the static method `getCardsInDeck()`? Because it doesn't depend on a particular object we would normally use the class name instead of the name of an object – however we can do either:

```
Card10 aceHearts = new Card10("Ace", "Hearts");

//We can do this
int numCards = Card10.getCardsInDeck();
```

or this:

```
int moreNumCards = aceHearts.getCardsInDeck();
```

Why 'Static'?

It seems an odd choice of word but it's used because static variables always exist while the program runs – they are 'static' as opposed to the 'dynamic' variables in objects that only exist once you've created that object (using `new`) and stop existing as soon as that object is finished with.

More About Static – Maths and `main`

In the past I've often asked questions about `static` in exams and understanding has not been great… So here's a bit more information to help with that.

Objects and classes make a *huge* amount of sense to lots of things we need to do – they combine data with operations on that data. *But there are some things that don't fit.* One example is mathematical functions – which are *pure* functions that only depend on the data passed directly to them. So in the Java class that contains mathematical functions – called `Math` – *all* the methods are static – one example is: `Math.sqrt()`.
The other case to consider is the `main` method that is the starting point of all Java programs – and which is also `static`. For a method not to be static, you have to have first created an actual object or you can't call it – if you look

back at all the non-static examples in this chapter and the last, you'll see that the way we called them was always:

```
objectName.methodName(…)
```

In the case of main, when we start running a program we haven't had a chance to create any objects, so it *has* to be static – otherwise there would be no way to call it because it "starts" the program, and until it *has* "started" there's no way to create an object.
For the same reason, all the example methods in the chapter on methods were static: *we had not created any objects of the relevant class* – simply because we hadn't got to the chapter on Classes and Objects. For example, in this case:

```
class SquareExample {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.print("Number? ");
          while(in.hasNextInt()) {
               int squareValue = in.nextInt();
               int result = square(squareValue);
               System.out.println("The square of "
                    + squareValue +
                    " is " + result);
               System.out.print("Another number " +
                    (anything else to quit)? ");
          }
     }

     static int square(int num) {
          return num * num;
     }
}
```

for `square()` to be non-static, there would have to be a line in the main method like this:

```
SquareExample squareObject = new Square();
```

Since there isn't the only way to be able to access the `square()` method is to make it static.

Classes and Methods – Nouns and Verbs

One way to think about classes and objects is that they represent things – cards, `ArrayLists`, `Scanners`, bank accounts, details about people. Some of these things are software models of real things (cards); and some are only software things (`ArrayList`) – but they are all in at least some sense *things*.

This means that the names of classes (and objects) tend to be **nouns** – words used to name things.
Conversely, the methods in a class represent actions – you call them to do something. This means their names tend to be **'verbs'** – words used to represent actions. I've put 'verbs' in '' because most method names are made up of more than one word so, strictly, they can't in linguistic terms be actual verbs – but they do represent *actions*. For example, consider this example from the last chapter:

```
Card card = new Card();
card.setSuite("Hearts");
card.setNameValue("Ace");

String cardName = card.getName();
int cardValue = card.getValue();
```

The names of the class (`Card`) and object (`card`) represent things so they have names, which are nouns. On the other hand, all the methods describe actions (they are 'verb-like') because they all represent something that is being done.

KEY POINT: Naming Classes, Objects and Methods

- When choosing the names of classes and objects use **nouns** – words that represent names of things.
- When choosing the names of methods either use **verbs** for single-word names; or sequences of words that describe **actions** – also see the two points below.
- When choosing names to set or get the values of properties (variables or things that could be variables),start them with **set** and **get** respectively unless they return values which are boolean.
- When choosing the names of methods that return boolean values, begin them with **is**.