2. Programming – Numbers and Arithmetic: Integers

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

In Chapter 1, we saw a couple of example programs and covered some key concepts. But a very important part of programming is dealing with numeric data. We will look at numbers, operators and programs using them in this chapter.

What Kind of Numbers?

Generally, mathematicians, scientists and engineers distinguish between *whole numbers* and *decimal numbers* - or more formally *integers* and *real numbers* (mathematicians will also separate out fractions, but we won't consider these).

Integers

Integers are represented in Java by the *data type* `int`. We've seen data types before – for example `String` from the first chapter. An `int` in Java is a number in the range -2,147,483,648 to 2,147,483,647 (inclusive). Why these apparently strange numbers? Because these are the *signed decimal numbers* that can be represented by 32 binary bits – which is the number of bits Java uses – with the *Two's Complement System* (which you will see in CS-150 and/or CS-155).

Advanced Aside

You might be thinking: what if I want bigger numbers? If so, you can use the data type `long` which uses 64 bits and goes from -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807. And if you want to go bigger *still*, there are ways to do it – but for us, `int` is fine. (You might also think: what if I don't need numbers that big? There is also a `short` type – which uses 16 bits, and `byte`, which is 8 bits. We won't be using these here – and you probably wouldn't unless there's a real reason you need numbers that size to, say, talk to other software. There's no need to try to save memory by using shorter values – we have plenty of memory these days.)

You create an `int` in Java like this:

```
int someNumber = 4;
```

This creates an integer variable called `someNumber` with a value of 4. As we saw in the last chapter, you don't have to initialize integer variables when you create them. So this is equivalent:

```
int someNumber;
someNumber = 4;
```

But generally, if you know the value of a variable, the first way is better.

KEY POINT – Naming Variables

One important point to make here is the way we've named the variable `someNumber` – unlike the example variables we had in the last chapter, this one is made up of *two* words. Notice the first word is all lower case, and the second starts with an upper case letter, but is otherwise all lower case again. This is the naming convention used for variables in Java – it's not required by the language, but *it is generally expected by all Java programmers (and me) that you'll stick to it.* It makes it easier to understand programs when you read them (because you can immediately see if an identifier is a variable or something else). Remember in the last chapter we also used identifiers to name programs, and we called them `HelloWorld` and `HelloYou` – these are not variables, but program (or class) names, and these *should* start with a capital letter. We'll see some more conventions (and actual rules) about naming later.

KEY POINT – Style

You can actually declare more than one variable of the same type at the same time:

```
int valOne, valTwo, valThree;
```

or

```
int valOne=1, valTwo=2, valThree=3;
```

However, it's considered bad style and not very readable – don't do it.

Advanced Aside

You may be used to programming languages where the underscore '_' character is used to separate words in variable names – for example `some_value`. We do **not** do this in Java for variables (though we do for 'constants' as we'll see later). Also, some languages use the convention that many variables start with 'm_' (e.g. `m_aVariable`) where 'm' stands for 'member'. We **don't** do this in Java either. **Neither of these things are standard conventions** in Java *so don't use them.* (Unhelpfully, when you write Android programs in Java you *do* use this way of naming – but we are *not* writing Android programs.)

Arithmetic

Once we've created a variable we can do some arithmetic on it. For example

```
int someNumber = 4;

someNumber = someNumber + 4;
```

After we've executed these two statements, the value stored in `someNumber` is 8. Unlike variables in Mathematics, variables in programming can change within a statement.

Speaking of Mathematicians…

If you are one, or know any, you might be unhappy about our use of the '=' sign:

```
someNumber = someNumber + 4;
```

clearly `someNumber` does NOT equal `someNumber + 4` in the mathematical sense! The key is not to think of '=' as a conventional equals sign, but to 'read it' as the word *becomes*. That is, the statement:

```
someNumber = someNumber + 4;
```

should be read as:

'someNumber becomes someNumber + 4'

It's a bit unfortunate in some ways that many programming languages do use the '=' sign this way: some don't (for example, some use '<-', or ':=') but most use '=' and it's too late to do anything about it now.

Let's do some more arithmetic:

```
int someNumber = 10;

int someOtherNumber = 5;

int gettingBoredWithTheseNames = 9;

int yetAnotherOne =
     someNumber * someOtherNumber; // '*' is multiply

gettingBoredWithTheseNames =
     yetAnotherNumber / someOtherNumber;
```

(Notice the last two statements are too long to fit on a line – so we've split them across two – we're allowed to do that and it's better if we *choose* where to split lines than just let them 'wrap' around the width of the page or screen. And notice we've *indented* the second line to make it clearer it's a continuation line.)

KEY POINT – Style

The point about breaking lines is important – and a common problem. One thing I see a lot in labs is people immediately making their editor window full-screen. This means they have a ***huge*** window to use – and one that is very, very wide. They merrily type very long lines of text in that look fine on their huge screen, but look awful in a smaller text window (for example, one in an *integrated development environment*) or if you print the code out, because the lines wrap in bad places and the wrapped lines mess up the indenting. My advice:

- Don't use massively wide windows to enter code.
- If you do, limit your lines to at most 100 characters (most editors let you set this) – if you do this, you will see that your huge window is mostly blank, and maybe (hopefully….) you'll stop doing it!
- Make sure *you* decide where to break lines to make you code readable – don't let your editor/printer do it for you (it will look awful).

The usual mathematical operators, are available to us, but sometimes look slightly different because the correct mathematical symbols are not available on the keyboard.

+	Addition
-	Subtraction
*	Multiplication
/	Division

But division here is *integer division* – it gives a whole number result. So, for example:

```
int x = 10/3;    //Has value 3 – NOT 3.3333….
```

This quite naturally leads to the idea that there should be a remainder operator – and there is:

%	Remainder

So

```
int x = 10%3;    //Has value 1 – 10/3 = 3 remainder 1
```

Advanced Aside

If you are familiar with some other languages, you might have come across a *power operator* - ^ - where, for example, `2^3` would be 2^3 or 2*2*2 = 8. Java does have a ^ operator but it does *not* represent the power function. In practice, we don't miss it.

Operator Precedence

You can of course combine operations together:

```
aValue = 9 * 3 + 4; //if we've already declared aValue
```

which evaluates to 31 – (9 * 3) + 4 – because Java obeys the usual rules of operator precedence (commonly called BIDMAS in UK schools). If you want to change this so that the terms you want to be computed first are in brackets:

```
aValue = 9 * (3 + 4); //This time evaluates to 63
```

I'm going to basically assume that you know this stuff because it's in the Mathematics syllabus of every educational system in the world, and you must have done it before you got here – but in case you've forgotten.

Brackets	Evaluate terms in (…) first
Indices	Things like x^y – Java does not have this for integers (but there is one in the Math library – next chapter)
Division	Next do divide: /
Multiplication	Then do multiply: *
Addition	And then addition: +
Subtraction	Finally the subtractions: -

Unary Minus

Another operator we need of course is unary minus – to change the sign of a value:

```
aValue = 4;
anotherValue = -aValue; //anotherValue is now -4
```

Advanced Aside

It's called *unary* minus because it only applies to *one* argument – hence unary (meaning one). The other arithmetic operators - +, - etc. – are *binary* operators, because they need two arguments.

Extra Shortcut Operators

These operators are of course all we need to do arithmetic – but language designers noticed long ago (the late 1960s) that expressions like this were common:

```
aValue = aValue * 3;
aValue = aValue + anotherValue;

aValue = aValue - 1;
aValue = aValue + 1;
```

The *unary increment and decrement operators* ++ and -- add and subtract one:

```
aValue--; //the same as aValue = aValue - 1;
aValue++;  //the same as aValue = aValue + 1;
```

The binary operators `+=, -=, *=, /=:`

```
aValue *= 3; //the same as aValue = aValue * 3

//next one is the same as aValue = aValue + anotherValue
aValue += anotherValue;
```

In practice, these are pretty handy: ++ is especially commonly used because we often use variables to *count* things.

Operator Summary

To summarise all the operators so far:

`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a % b`	Remainder
`-a`	Unary minus – change sign of `a`
`a++`	Add one to `a`
`a--`	Subtract one from `a`
`a += b`	Add `b` to a
`a -= b`	Subtract `b` from `a`
`a *= b`	Set `a` to `a * b`
`a /= b`	Set `a` to `a/b`

The + Sign – Using it Twice…

In the last chapter we used + to join strings together. Now we're using it again to add numbers. How is this possible? The answer is that provided Java can understand what *type* of data you're applying it to, it can work out which operation you are doing. So if the things on either side of the + sign are Strings, it joins them; if they are numbers, it adds them. And if one is a number and one is a String, it also joins them. Using an operator symbol like this in more than one way is called *overloading.*

Case Study: The Swimming Pool Cost Estimator

Now we're going to work on a bigger example – the swimming pool cost estimator. The idea is to work out how much it costs to build a swimming pool. To build a pool you need to:

- **Excavate the space for the pool** – the cost of this depends on the volume of the pool
- **Line the pool with concrete and tile it**– the cost of this depends on the area of the walls and base of the pool
- **Add in the pumps and machinery** – for normal-sized pools, this is pretty-much the same: that is, it's a constant value (as this is an estimate).

Our first task is to work out an *algorithm* for the calculation:

1. Work out the *volume* of the pool – length * width * depth - and multiply the cost per unit volume (we'll use m^3) to get the cost of excavation
2. Work out the *area* of the walls and base to get the cost of lining and tiling. This breaks down into smaller steps:
    a. Work out the area of a long side (length * depth) – but there are two so multiply by two.
    b. Work out the area of the short side (width * depth) – again multiply by two
    c. Work out the area of the base (width * length)
    d. Add the areas together and multiply by the cost per unit of area (we'll use m^2)
3. Finally, add up the excavation and lining/tiling costs, and the machinery cost, to get the final estimate.

Strategy

Of course, there is also a step before these – read in the width, depth and length of the pool; and one after too – printout the estimate. But we're not going to do it in that order – we're going to build it in stages starting with the calculation code. ***The first thing you need to do to solve programming problems is to break them down into smaller parts***.

KEY POINT – Build Programs in Stages and Not from the Beginning

Usually, it's *not* best to just start a program at the start – instead, break it down into parts, and build the parts you can test. Then put them together to build the whole program. It's always easy to test a small part of a program in comparison to a whole, large one.

In this case, we're going to work on the calculations first. Obviously, in the final program we need to have read in the values before we can do any calculations – but for now we'll just use temporary values. Here's our first program:

Pool Volume

```
/* Program to compute the volume of a pool
   This is the first stage of building the pool cost
   estimator program
*/
public class PoolVolume {
     public static void main(String[] args) {

          int depth = 2; //Values in metres
          int width = 4;
          int length = 7;

          //Now calculate the volume
          int volume = depth * width * length;

          //Print it out - volume should be 56
          System.out.println("Depth: " + depth +
               " Width: " + width +
               " Length: " + length);
          System.out.println("Volume: " + volume);
     }
}
```

Notice we've split the first `System.out.println` over three lines, and lined up the components – we don't have to do this, but it makes it more readable.

Enter this program and run it – it should print out the values of depth, width and length, and the volume 56.

Exercise

Change the values of depth, width, length – make sure you update the comment! – and run it to see if it still works. That is, test the program with multiple data values – it might seem excessive here, but later on you will need multiple tests to be at least fairly sure your progam is correct. (The program will still work if you don't update the comment - *but it's bad practice. Start as you mean to go on:* ***do it properly – when you change the values for the depth, width and length, also change the comment***).

Aside – Testing

It's easy to think that once your program compiles and runs, then it's correct. But in most cases, programs do not initially do what they are supposed to do, and contain logic errors. For example, if we had put this line in:

```
int volume = depth * width; //left out length!!
```

then we are going to get the wrong answer even though the program 'works'. You should always run test cases on your program – the number (and way) you do this will depend on how sophisticated it is. *But minimally you should always know what the right answer is before you start.*

Excavation Cost

Computing the volume is only part of the process – we need to compute the actual cost. To do this, we need to multiply by the cost per m^3. Let's just assume that this is £20/m^3.

KEY POINT – Data Values in Java have no Units

Notice we've talked about values in £ and in m^3 – but Java does not know anything about these units. We must keep track of these ourselves. In this case, if we assume that depth, length, width are in metres, and we multiply them together; and then multiply by £20/m^3 the final value will be in £ - but as far as Java is concerned, it's just an integer.

Pool Excavation Costs

Now we extend our program to calculate the excavation costs.

```
/* Program to compute the cost of excavating a pool
   This is the second stage of building the pool cost
   estimator program
*/
public class PoolExcavationCosts {
     public static void main(String[] args) {

          int depth = 2; //Set to specific values for now
          int width = 4;
          int length = 7;

          //Now calculate the volume
          int volume = depth * width * length;
```

```
                //Print it out - volume should be 56
                System.out.println("Depth: " + depth +
                     " Width: " + width +
                     " Length: " + length);

                System.out.println("Volume: " + volume);

                //Now do the costs
                int cost = volume * 20;

                //Should be 1120
                System.out.println("Cost: " + cost);
        }
}
```

Notice we've updated the comments and changed the name to reflect the new, slightly changed, function – i.e. **we've done it properly!**

Problem – a Magic Number

There's one thing I don't like about this program:

```
int cost = volume * 20;
```

The reason I don't like it is that the meaning of 20 is not clear – what is it exactly? Someone reading the program would not immediately know.

We call this a *Magic Number* – because it just appears 'by magic' with no explanation of what it means.

We could just put in a comment:

```
int cost = volume * 20; //20 is £ per cubic m excavation
```

But there's another problem – though it doesn't actually apply here. Very commonly, when numbers appear in programs, ***they appear more than once.*** Suppose you want to *change* them at some point in the future?

- You have to search through your program (and things you need to change could be anywhere) to find them all and change them.
- Because they are just numbers, there may be *other* values that represent *different* things which just happen to have the *same* numeric value, and you must not change them.

To illustrate the problem, soon we're going to write the code that calculates the lining/tiling costs – and suppose the cost of that is £20/m^2 (note m^2 **not** m^3). Then suppose the excavation costs go up to £25/m^3 – *it would be very easy to change them both **or** change the wrong one.*

KEY POINT – Don’t Repeat Yourself (DRY)

A hugely important concept in computing and software engineering is DRY – Don’t Repeat Yourself. **Data items should be defined once and only once.**

It’s all very well me saying that – but how? One way would be define a variable that stored the value:

```
int excCostPerCubicMetre = 20;
```

This will work but what is we accidently change it? Perhaps by accidentally putting in a line like:

```
excCostPerCubicMetre++; //now =21 and wrong!
```

The program would still ‘work’ but the cost per cubic metre would be wrong. This value should not be changed by the program. You may think that this would not be possible – but accidents happen.

KEY POINT – Many Programming Features Protect Against Mistakes

- If programmers were perfect, programming languages could be a lot simpler – and there would be no bugs.
- It’s much better if the *programming language* can find mistakes than leaving it to the programmer.
- This is why we have to *declare* our variables and say what *type* they are – this *massively* helps the compiler catch mistakes we make. For example, it can work out if you’ve typed the name of a variable wrongly, or incorrectly mixed numbers and Strings.

Advanced Aside

Even more than this, it’s much better for the *compiler* to find problems than for it to have to wait until the program is *run*. This is because the compiler looks at the *whole* program every time you compile the code – when you run a complex program, only some parts may be run. So problems may not be found for ages.

The Solution – Final Variables (Constants)

We need a way to declare and set variables so we cannot later change them. We do this in Java with *final* variables:

```
final int EXC_COST_PER_CUBIC_METRE = 20;
```

There are two things we need to explain about this.

- The keyword `final` – if you put final in front of a variable declaration, you can only assign to the variable once and cannot change it later.

- The name `EXC_COST_PER_CUBIC_METRE` – final variables are named differently from other variables: they are all in capitals with '_' separating the words. The reason we do this is so it's immediately obvious to someone reading the program that the value cannot change.

(You might think that is a bit too much to type

```
EXC_COST_PER_CUBIC_METRE
```

and it *might* be a bit long. But the key thing is that it's *obvious* what it means to someone reading it.)

Note that we do not have to set the value of a final variable straight away. So this is ok:

```
final int SET_LATER;

SET_LATER = 20;
```

You can use this for example to read in the value you want stored in the final variable. But what you *cannot* do is this:

```
final int SET_LATER;

SET_LATER = 20;

SET_LATER = 30; //NO – will not compile
```

Because `final` variables cannot be changed once set.

We have introduced final variables here because we need them in the pool example – but there's a bit more about them at the end of this chapter.

Advanced Aside

You may be familiar with programming languages that have constants, and you may think that final variables are constants. They more or less are – but the fact that we don't have to set them to start with makes them *subtly* different. For example, you could set one to a value read in from a user – you can't do that with 'true' constants. Java does actually reserve the keyword `const` – but for now it doesn't use it.

Pool Excavation Costs – New Version

Here's our updated code with a final value for the excavation costs.

```
/* Program to compute the cost of excavating a pool
   This is the second version using a final value for
   the excavation costs per cubic metre
*/
public class PoolExcavationCosts2 {
     public static void main(String[] args) {

          final int EXC_COST_PER_CUBIC_METRE = 20;

          int depth = 2;
          int width = 4;
          int length = 7;

          //Now calculate the volume
          int volume = depth * width * length;

          //Print it out - volume should be 56
          System.out.println("Depth: " + depth +
                             " Width: " + width +
                             " Length: " + length);

          System.out.println("Volume: " + volume);

          int cost = volume * EXC_COST_PER_CUBIC_METRE;

          //Should be 1120
          System.out.println("Cost: " + cost);
     }
}
```

Notice that the value of 20 for the excavation costs is now very clearly named, it appears only once, and *if we want to change it we only have to look at the top of the program.*

Pool Lining/Tiling Costs

Now we're ready to move on to the lining and tiling costs. This is very similar – we have to compute the area of the sides/base of the pool, and multiply by a cost per m^2. The actual calculations are a bit more complex, but not much. Here's our example program – I've skipped printing out the area of the pool and gone straight to the cost.

```
/* Program to compute the cost of lining and tiling a
   pool
*/
public class PoolLiningTilingCosts {
     public static void main(String[] args) {

          final int LINE_TILE_COST_PER_CUBIC_METRE = 25;

          int depth = 2;
          int width = 4;
          int length = 7;

          //Area of the base
          int baseArea = width * length;

          //Area of ends
          int endArea = width * depth * 2; //Two ends

          //Area of sides
          int sideArea = length * depth * 2;

          //Cost
          int lineTileCost = (baseArea + endArea
                + sideArea)
               * LINE_TILE_COST_PER_CUBIC_METRE;

          //Print it out - cost should be 1800
          System.out.println("Depth: " + depth +
                         " Width: " + width +
                         " Length: " + length);

          System.out.println("Cost: " + lineTileCost);
     }
}
```

Notice a few things about this.

- We have used (…) in the expression to work out the costs because we need to add *before* we multiply.
- We have added the comment `//Two ends` to remind readers that there are two ends to a pool – this is an example of using a comment to illustrate a potentially tricky bit. (You might think it's obvious that a pool has two ends – but when I used to set this example as a lab class, about 75% of people forgot this the first time they did it.)
- Note we've also used *blank lines* to separate blocks of code – this helps with readability.
- We have also used a comment before each 'logical block' to say what it's doing. Actually, in this case, I've put extra ones in to make the point clearer – in practice, I would not put this many in (nearly one per line) and would write it like this:

```
//Compute area and cost
int baseArea = width * length;
int endArea = width * depth * 2; //Two ends
int sideArea = length * depth * 2;
int lineTileCost = (baseArea + endArea
      + sideArea)
     * LINE_TILE_COST_PER_CUBIC_METRE;


//Print it out - cost should be 1800
System.out.println("Depth: " + depth +
               " Width: " + width +
               " Length: " + length);
System.out.println("Cost: " + lineTileCost);
```

Lots of Variables….

You may think, particularly if you've programmed before, that I'm overdoing the variables and that I'm 'wasting' memory. I could just compute the area like this:

```
int poolArea = width * length + width * depth * 2
     + length * depth * 2;
```

and skip a lot of lines. And I could – and you might think this would 'save' the memory used by the variables I don't declare anymore. I could go further, and just do this:

```
System.out.println("Cost: " + (width * length
          + width * depth * 2
          + length * depth * 2)
          * LINE_TILE_COST_PER_CUBIC_METRE);
```

(I've highlighted the first + in red for a reason – see below.) Yes we could and it would work. But…

- The first shorter version with only one variable is fine.
- But in the second one, it's getting a bit confusing to work out what's going on.
- We are using + for two different things – the first (red) one is joining strings; the others are actual additions. Again, potentially confusing.
- Although it looks like we might be 'saving' some memory by not having the extra variables:
  - Compilers are very good these days – just because you declare a variable does not mean the compiler will actually create it if it works out it doesn't need to.
  - Humans are actually very bad at working out what is 'efficient' or not in computer programs.

    - So what if you *are* wasting memory – an integer variable takes up 32 bits; computer memory today has *billions.*

KEY POINT – These Three Things, in This Order

Programs need to do these things in this order:

- They need to work – obviously the most important.
- They need to be easy to read, understand and change – this is a *very close second t*o working
- They need to be efficient – this is a *very distant third* to the other two. In this module, there's no need to worry about it at all.

In fact in most programs there is no need to *ever* worry about efficiency – interaction with users is by far the slowest part. Of course there are programs where this is an issue – but not in this module, and not for very many programmers even in their working career.

Advanced Aside

The kind of 'efficiency' I'm talking about here is the programmer tweaking the program – trying to shave a bit of time here and there. This is nearly always a waste of time – *but* there *are* changes that do have a huge impact. Picking the right *algorithm* in the first place before you start writing code is *massively* significant. It won't impact us here, but there's a module that includes this in the second year.

Back to the Program – Data Input

We can now compute the volume, area and associated costs. The machinery costs are a constant value, so it's pretty obvious we can compute the final costs by just adding everything up and printing it out. The one bit we haven't done is getting user input.

We DID do this of course in the last chapter – we read in the user's name. But that was a different type of data – a string of characters, not a number.

Data Types, Numbers and Representation

At this point you might be thinking 'but if I type a number it is a string of characters' – well, yes, but that's the *representation* of the number, not the number itself. Here's a table showing different representations for the *same* number

42	Western Decimal
٤٢	Arabic Decimal
101010	Binary
2A	Hexadecimal
52	Octal
XXXXII (or XLII)	Roman

These are all the *same* number – just represented differently. What we want is not the representation, but the actual number itself – so we can do arithmetic on it.

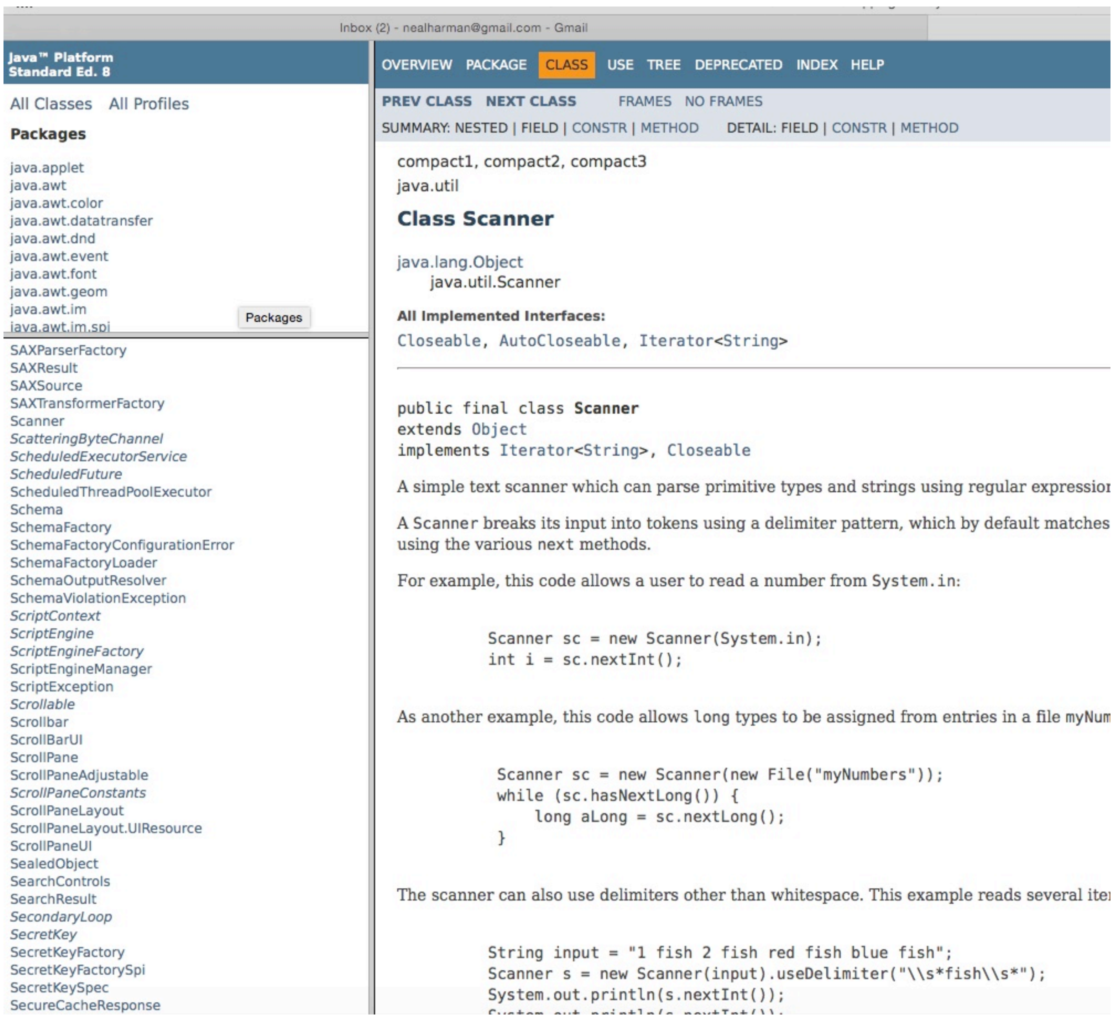
*Strings and integers are different **data types** – and we can't just interchange them*. We'll do more on data types later.

The Scanner Library

The solution to this problem is in the Scanner library we used to read in input in the last chapter. It turns out there is more than one way to read in input. The first step is to find the documentation for the Scanner. Go to this URI:

http://docs.oracle.com/javase/8/docs/api/

and you'll see something like this:



compact1, compact2, compact3
java.util

Class Scanner

java.lang.Object
    java.util.Scanner

All Implemented Interfaces:
`Closeable, AutoCloseable, Iterator<String>`

```
public final class Scanner
extends Object
implements Iterator<String>, Closeable
```

A simple text scanner which can parse primitive types and strings using regular expressio

A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches using the various `next` methods.

For example, this code allows a user to read a number from `System.in`:

```
     Scanner sc = new Scanner(System.in);
     int i = sc.nextInt();
```

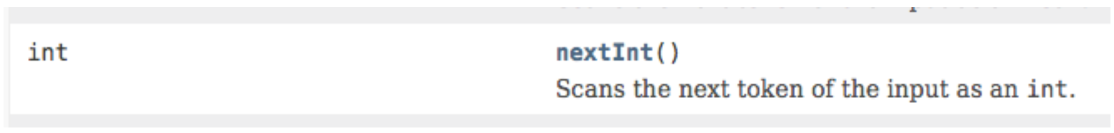
As another example, this code allows `long` types to be assigned from entries in a file `myNum`

```
      Scanner sc = new Scanner(new File("myNumbers"));
      while (sc.hasNextLong()) {
          long aLong = sc.nextLong();
      }
```

The scanner can also use delimiters other than whitespace. This example reads several ite

```
     String input = "1 fish 2 fish red fish blue fish";
     Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
     System.out.println(s.nextInt());
```

This is a screenshot of the documentation library for Java. The tall window in the bottom left is the list of library items available (there is a huge number). I've scrolled down to Scanner and clicked it – so details are in the big window on the right. At the top are some examples of how to use a Scanner – in fact the very top one tells you what we need to do here. But more generally, if you scroll down, you'll get to a section called *Methods*. This is the list of things that a scanner can *do* – and the one we're interested in is:



int	nextInt() Scans the next token of the input as an int.

This says that there is a method called `nextInt` that returns an integer that it reads from the input. If you click on `nextInt` it will tell you some more – a bit more than we need to know right now.

Using this information, the example at the top of the Scanner page, and our last program, we can have a go at reading in input data.

Reading in One Integer

Here's a program to read in and print out one integer number:

```
/*Program that reads and prints a single integer
 */
import java.util.Scanner;

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

          System.out.print("Enter an integer: ");
          int value = in.nextInt();
          System.out.println("You typed: " + value);
     }
}
```

Notice that instead of `System.out.println` we've typed `System.out.print` – this is the same except that it does not print a newline character, but instead leaves the input cursor on the same line as the text it just printed out.

Instead of reading in a string of characters and storing them in a `String` variable, we are now reading in a string of characters, converting them to an integer, and storing then in an `int` variable (which means we can do arithmetic). What if the characters we type are not an integer? See exercise below.

Exercise

- Enter this program and run it, typing in a whole number – you'll find it works

- Now try entering something that isn't an integer. You should see something like this:

```
Exception in thread "main"
java.util.InputMismatchException
	at java.util.Scanner.throwFor(Scanner.java:864)
	at java.util.Scanner.next(Scanner.java:1485)
	at java.util.Scanner.nextInt(Scanner.java:2117)
	at java.util.Scanner.nextInt(Scanner.java:2076)
	at ReadOneInt.main(ReadOneInt.java:10)
```

This is an example of a program going wrong when we run it – previously we've only seen errors when we compile programs.
Notice that on the very last line it says where in our program the error occurred – and on the second line it has something that does a reasonable job of explaining the problem `InputMismatchException` – meaning that the data we entered wasn't the right type.

It's obviously not acceptable for programs to crash just because we type the wrong kind of data – and later in the module we'll see how to fix this. But for now, we just have to be careful about what we type.

Reading Depth, Length, Width

Now we know how to read in an integer, reading three is easy:

```
/*Program that reads and prints three integers
 */
import java.util.Scanner;

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

		System.out.print("Enter the depth: ");
		int depth = in.nextInt();
		System.out.print("Enter the length: ");
		int length = in.nextInt();
		System.out.print("Enter the width: ");
		int width = in.nextInt();

		System.out.println("Depth: " + depth +
			" Length: " + length +
			" Width: " + width);
	}
}
```

Exercise

Before you read the rest of this, try putting the parts of the programs above together to create the whole thing. It's not as simple as copying the whole programs though!

The Complete Swimming Pool Estimation Program

Here is the complete program – made up of parts of the programs we wrote above (having written it, the various costs seem ridiculously cheap for a real pool – but because we've used final variables we could easily change that).

```
/*Program that estimates the cost of a swimming pool
 */
import java.util.Scanner;

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

          //These value represent the costs of each phase
          final int EXC_COST_PER_CUBIC_METRE = 20;
          final int LINE_TILE_COST_PER_CUBIC_METRE = 25;
          final int MACHINERY_COST = 1000;

          Scanner in = new Scanner(System.in);

          //Read in the pool size
          System.out.print("Enter the depth: ");
          int depth = in.nextInt();
          System.out.print("Enter the length: ");
          int length = in.nextInt();
          System.out.print("Enter the width: ");
          int width = in.nextInt();


          //Now calculate the excavation costs
          int volume = depth * width * length;
          int excCost = volume *
               EXC_COST_PER_CUBIC_METRE;

          //Calculate the lining/tiling costs
          int baseArea = width * length;
          int endArea = width * depth * 2; //Two ends
          int sideArea = length * depth * 2;
          int lineTileCost = (baseArea +
               endArea + sideArea) *
               LINE_TILE_COST_PER_CUBIC_METRE;

          //Calculate the total cost
          int totalCost = excCost +
               lineTileCost + MACHINERY_COST;

          System.out.println("Estimated Cost: " +
               totalCost);
     }
}
```

To create this program, we've taken the key parts of the ones we wrote above and combined them.

- We have removed all declarations of variables depth, width and length except for the first ones – *don't declare the same variables more than once* (though see the definition of *Scope* later in the module).
- We've also renamed a couple and reformatted the code a bit to tidy it up
- As always, we've updated the name and comments.

Final Variables in Practice

Using final variables in Java is a constant source of confusion in coursework. There are two basic reasons for using final variables in Java.

- The first is if someone is going to look at your code and say "what is that!!??" because it's some meaningless value (e.g. 6.62607004e-34 means nothing but `PLANKS_CONSTANT` does – at least if you're a Physicist).
- The other reason for constants is ***because you're anticipating the possibility of something changing in the future.*** This is particularly true for values that are going to appear more than once in your code, but it's even true if they don't. There's a tendency when doing coursework to not think like that, because it's one-use only: it's just for the assessment. *But you need to get into the way of thinking because that's what you're going to have to do commercially.* Code has a long life and often you will not be the person who initially wrote it, and neither will you be the last person who worked on it – you are 'looking after it' for a while. In that case, you want to be anticipating possible changes in the future and building you code to make them easier – and constants are key here. One simple example is localising a program for another language. Having all the text strings the program will print in one place, represented as constants, makes it massively easier to change them into another language.

KEY POINT – Final Variables are for Thinking Ahead

When you are writing a program even if it's only for, say, a lab or coursework – try to imagine that it's something that will have a life beyond the time you are working on it, and think about how you can make it easier to change in the future.

Meaningless Final Variable Names

One thing that you see occasionally in coursework is names like this:

```
final int BAG_ONE = 1;
```
Or, worse:
```
final int ONE = 1;
```
(or similar). In general, if you find yourself writing names like that then either (a) a constant is not very helpful; or (b) you should think of a name that

represents what the value *means*, not what it's value *is*. One 'test' of the 'sensibleness' of a constant name is: if it's not a physical constant (like π) would it ever make sense to change its value? And if it did make sense, what does that do for the name. So in the example above, it's clearly absurd:

```
final int ONE = 2;
```

So in that case either drop the final variable or change the name.