3. Programming – Data Types and Names

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

This chapter is about two subjects we've already looked at quite a bit already – *types* of variables and what to *call* them. Why a whole (though short) chapter? Mostly so we can put all the information and rules in one place.

Data Types

A key part of programming is dealing with different types of data – and we use the term *data type* to refer to this. So far we've seen two – `String` and `int` representing sequences of characters and integers. There are others – some of which we'll deal with in more detail later. Here's a basic list.

Name	Examples	Notes
`int`	`int a = 5;` `int b = a * 3;`	Represents integers, dealt with at length in Chapter 2
`String`	`String value = "hello";` `String val2 = value + " friend";`	Represents sequences of characters, dealt with at length in Chapter 1
`double`	`double pi = 3.1415;` `double area = pi * radius * radius;`	Represents non-integer data
`boolean`	`boolean value = true;` `boolean value = false;`	Represents truth values, used in logic to make decisions
`char`	`char val = 'x';`	Represents single characters (as opposed to sequences)

We've already done `String` and `int` – we won't say much about char but we need to talk a bit about the others.

Double

The data type `double` represents decimal data. Like `int` it also has a range of (very similar) arithmetic operators – +, -, *, / etc. (though there is no % as this would not make sense). An important point about `double` is that they are often *approximations* because it (potentially) takes an *infinite* amount of memory to represent decimals. As an example, consider the fraction 1/3, which is represented by the recurring decimal 3.333333…. – for ever. This needs an infinite number of bits, which is clearly not possible. Similarly, the constant π is an infinite sequence. The consequence of this is that errors can *accumulate* over time as you do arithmetic.

Combining Number Types

If you are just using `double`, or just using `int`, then there are no real problems – the same operators are available for both `double` and `int` (except there is no % for double), and the results are mainly what you would expect. There are just a few things to worry about.

Making Sure a Double is a Double…

Consider the following program:

```
class DoubleDivideFail {
     public static void main(String[] args) {
          double mystery = 10/3;
          System.out.println(mystery);
     }
}
```

We have declared the variable mystery to be a `double`, so you might this this will print 2.5 – but it actually prints 2.0. This is because Java recognizes both 10 and 3 as integers, so it does integer division 10/4 = 2 remainder 2. This not what we want to happen, and there are two ways to fix it. The first is to explicitly force one of either 10 or 4 (or both) to be doubles by writing them as decimals – here's an example where we have re-written 4 as 4.0:

```
class DoubleDivide {
     public static void main(String[] args) {
          double works = 10/4.0;
          System.out.println(works);
     }
}
```

The other is to force Java to change an `int` to a `double`:

```
class DoubleDivide2 {
     public static void main(String[] args) {
          double works = 10/(double)4;
          System.out.println(works);
     }
}
```

We have written `(double)4` which forces Java to turn the `int` 4 into the `double` 4.0. This is called *casting* and is an example of forcing one type to turn into another. This is only legal when it makes sense, which it does here. Since in most cases, we are not dividing by 'simple' numbers, but by variables, then casting is usually what we have to do. For example:

```
double works = y/(double)x;
```

where `x` and `y` are integers – we cannot just write '`x.0`' – it doesn't make sense and won't compile; but `/(double)x` does work.

Ints can be Doubles; Doubles can't be Ints

Notice in the first divide example above the answer printed is 2.0, not 2. Even though 10/4 involves only integers, and integer division, because we've said the variable mystery is a `double`, Java converts it for us. This makes sense – we can always represent an `int` as a `double` because, mathematically, the Integers are a subset of the Real numbers.
But the same is not true the other way round. We're going to be using a version of the same program above in this section, and to keep things shorter I'll only show the single important line that does the arithmetic.
If we change the important line in our example to

```
int mystery = 10/4;
```

We get 2 – which is fine since 10 and 4 are integers, we are doing integer division, and so we get an int as a result. But if we change it to:

```
int mystery = 10/4.0;
```

(or use the casting version) we get:

```
XXX.java:3: error: incompatible types: possible lossy
conversion from double to int
          int mystery = 10/4.0;
```

because, quite naturally, since 10/4.0 is no longer an integer division (because 4.0 is not an integer) we cannot store the result of this calculation in an `int`.
What if we change it so that the result *is* an `int` – by changing 4.0 to 5.0 (because 10/5=2)? ***It still doesn't work*** *– because in general, Java cannot know when it compiles a program that the result will always safely be an integer.*

What if we really want to force this to happen? We can by using casting:

```
int mystery = (int)(10/4.0);
```

We've used brackets to make sure the cast to the `int` type applies to the result of the operation – in this case, it prints 2. Casting a double to an `int` always throws away the non-integer part.

KEY POINT – Careful Mixing Integers and Doubles

Though you can store the result of an integer operation in a double, you cannot do the opposite (and be careful with integer division and doubles – force one of the operands to be a double to be safe).

Representing Doubles

So far we've written down doubles simply: for example, 4.0. But this is not going to be convenient for very large or very small numbers. You should be familiar with *scientific notation:* $a \times 10^b$ – for example 1×10^2 (which is 100.0), or 5.4×10^{-4}, which is 0.00054). In Java just replace `x10` with `e`: `1e2` is 100; `5.4e-4` is 0.00054. The '`e`' stands for *exponent* – in `1e2` 2 is the exponent, and 1 is the *mantissa*.

The Math Library

If you are using doubles you might well want to use more advanced mathematical operations – we won't (much…) but if you do, you can find the documentation for the library at the same URL we found Scanner. So if you want, for example, to raise a number to a power – there is no direct operator for it, but you can do it with the Math library.

Advanced Aside

Why the odd name, `double`? The mathematical name for decimals is the *Real* numbers – but because they are only approximate, it's not very accurate to use the term here (though some languages do). Instead, they are normally called *floating-point* numbers (because the decimal point can move back and forward – 'float' – to make the representation of possible numbers more flexible). The problem is that most floating-point numbers (called `float` in Java), like integers, use only 32 bits of data. This does not allow a very wide range of numbers, or a very high precision. So most languages, including Java, have a 64 bit version – or *double length* – of floating-point to allow more precision (and a wider range of numbers). These days, because memory is plentiful, most people don't bother with the 'old' `float` and just use `double` all the time.

Boolean

Named after the Irish mathematician George Boole (1815 – 1864). Unlike most other data types, Boolean has only two possible values – true and false. We use the Boolean type when we need to do logic, which is mainly when we need to make *decisions*, as we'll see when we talk about if statements and loops. The Boolean data types comes with a range of operators representing logical operators:

Operation	**Name**	**Meaning**
`!a`	Not	If `a` is `true`, then `!a` is `false`; if `a` is `false` then `!a` is `true`
`a && b`	And	`true` if *both* `a` and `b` are `true; false` otherwise
`a || b`	Or	true if *either one or both* `a` and `b` are `true; false` otherwise
`a ^ b`	Exclusive Or	`true` if *either* `a` or `b` are `true` *but not both*; `false` otherwise

Just like ordinary arithmetic operators, there are precedence rules: first you do the operations in brackets; then any `!` operations; then `&&;` and finally `||` and `^` (order of these last two does not matter).
Here are some examples:

```
boolean a = true;
boolean b = true;
boolean c = false;

boolean d = a && b; //This is true
boolean e = a && c; //This is false
boolean f = a || c; //This is true
boolean g = a ^ b; //This is false
boolean h = a ^ c; //This is true
```

On its own `boolean` is not very useful – what we need is a way to create them by comparing data of different types. For example, by comparing numbers. There is a range of operators that are either true or false, depending on the relationship between a pair of numbers:

Operation	Name	Meaning
`a == b`	Equality	`true` if `a` equals `b`; `false` otherwise
`a != b`	Inequality	`true` if `a` not equal to `b`; `false` otherwise
`a > b`	Greater than	`true` if `a` greater than `b`; `false` otherwise
`a >= b`	Greater than or equal	`true` if `a` greater than or equal to `b`; `false` otherwise
`a < b`	Less than	`true` if `a` less than `b`; `false` otherwise
`a <= b`	Less than or equal	`true` if `a` less than or equal to `b`; `false` otherwise

Here are some examples where `x`, `y` and `z` are of data type `int`.

```
boolean a = (x == y); //true if x equal to y
boolean b = (y > 0);  //true if y greater than zero
boolean c = !((z > 0) || (z < 0)); //see below
```

The last one is tricky and you would not really write this because it can be simplified (see below) – but it's included as a more complex example. The term `z > 0` will be true if `z` is greater than zero; and the term `z < 0` is true if `z` is less than zero – then the term `(z > 0) || (z < 0)` is true if either z is greater or less than zero – i.e. not equal to zero. However, the ! is applied to the whole expression, so the overall meaning is that c will be true if (and only if) z is equal to zero. In this case of course we would write:

```
boolean c = (z == 0);
```

KEY POINT - = vs ==

A very common mistake to make, which is very natural when you've spent many years using '=' for equality, is to write '=' instead of '=='. There is nothing you can really do about this, except practice!

We are going to see a lot more of `boolean` later on, when we need to deal with decisions – we won't often actually declare `boolean` variables, but we will use `boolean` expressions all the time.

KEY POINT – Comparing Doubles

Above we saw the == operator for checking if two numbers are equal. This works fine for integers, but for doubles it does not; at least not always. Try entering and running this program:

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

          double r = Math.sqrt(2.0);

          //You would expect r*r==2
          System.out.println(r*r);

          //And this would be true
          System.out.println(r*r == 2);

          //But neither is true

          //And this is the right way
          final double EPSILON = 1E-14;

          //And this *is* true
          System.out.println(Math.abs(r*r - 2.0)
               < EPSILON);
     }
}
```

We are using the `Math` library here – `Math.sqrt` is the square root operator; and `Math.abs` returns the absolute value of a number. Obviously, mathematically $\sqrt{2}^2 = 2$ but that is not happening here, because Java cannot represent the true value of `r` exactly – so $\sqrt{2}^2$ is *nearly* equal to 2 but not quite. The way to solve this problem is *not* to test if doubles are equal to some value, but instead are *close enough* – that's what the part of the program after `//And this is the right way` does – it checks to see if the two values are within `1E-14` (10^{-14}, or 0.00000000000001) of each other.

Char

The char type represents single characters – we won't be dealing with these much, but they can be useful. The char type uses single ' characters instead of the double " used with Strings. It's important to know that:

```
char a = 'a';   //The single character 'a'
String b = "a"; //String containing the one character "a"
```

are ***not*** the same.

KEY POINT – Mixing Types

Pretty obviously, the different data types represent different data, and you cannot just mix them – you can't for example, assign a `String` to an `int`; or a `char` to a `boolean`. There are some exceptions to this – for example, as we saw above, in some cases you can mix integers and doubles. Also, you *can* in *some* circumstances force Java to mix types; and you can also *sometimes* convert one to another. But for the most part the rules make sense and if Java is not letting you compile a program because it says the types of variables don't match, then chances are it's correct and you've made a mistake.

KEY POINT – Type Names

One thing you might have noticed is that the names of most types we've seen so far are in lower case – e.g. `int, double`. But the exception is `String`. The reason for this is that `int, double, char` and `boolean` are *built in primitive* types; and `String` is a much more *complex* type. It's actually an example of something called a *Class*, just like `Scanner`. We will get back to this later on and deal with classes in much more detail. But for now, it's just something you have to remember.

Advanced Aside

Java could make the types `int, double` etc. classes too – and some languages do this. But the disadvantage is that it can be very slow. So the designers of Java made the decision to make these more efficient, primitive items. There are times though when we *do* need to treat things like `int` as complex objects – and when we need to do that, there are Class versions (for example, `Integer` for the `int` type). We will sometimes see these Class versions – for example, when we convert between types.

Identifier Names

Just as we've seen some data types before, we have also seen some examples of the 'rules' for naming identifiers in Java. There are two kinds of 'rules':

- **Those imposed by the language itself**. These are *real* rules, you *must* obey for your programs to compile.
- **Those defined by programmers**. These are *conventions*, *not* rules *but* all Java programmers (pretty much) think they are a good idea and obey them, ***and so should you.***

In the rest of this chapter, we'll define and explain these rules.

Java Language Rules

I'm going to try to keep this section short – because these are not the important rules; the next ones are. *If you stick to the* ***programmer-defined rules*** *for names in Java explained in the next section, you will automatically keep to the actual language rules.* Identifier names in Java must begin with an alphabetic character (i.e. letter) of either case; the rest of the name can be made up of alphanumeric characters (numbers and letters) of either case as well as the '_' character. *All* other characters (including space) *are not allowed*. That's it.

Programmer Defined Rules

THESE ARE THE IMPORTANT RULES – STICK TO THESE.

The rules you use to name things depend on *what* you are naming. Here are the things we need to name in this module

Thing	Rule	Examples

Non-final Variables and methods – we haven't written our own methods yet but we will	Starts with a lower case letter; all 'inner' words start with an upper case letter; all other letters are lower case; can contain numbers (but not start with one); **never** contains '_'	`lemon – OK`
		`lemonSorbet – OK`
		`lemon_sorbet – not OK`
		`lemon2 – OK`
		`lemonIceCream - OK`

Program names (or, strictly, **class names**)	Same as variables but also starts with an upper case letter.	`Lemon - OK`
		`LemonSorbet - OK`
		`lemonSorbet – not OK`
		`Lemon_Sorbet – not OK`

Final Variables	All upper case – with a '_' between each word.	`LEMON - OK`
		`LEMON_SORBET - OK`
		`LEMONSORBET – not OK`

Note that final variables are the only time you should use '_' in Java names.
Notice we said that variables and methods follow the same rules. We haven't yet written any methods (or even really said what they are) – but we will in a while. We can use the same rule for both variables and methods because methods will always have brackets after the name, so we can easily tell them apart.
The reason we have these rules is that it makes it easier for a programmer reading the code to understand what a name represents – and this is why it's critical to stick to them

THIS IS ANOTHER VERY IMPORTANT RULE – STICK TO THIS
The names you choose should be readable and reflect what the data the refer to means. For example, consider this:

```
int x = 90; //what is x???
```

as compared to this:

```
int rightAngle = 90; //much clearer now, doesn't actually
need a comment
```

Avoid the temptation to use names based on the value of a variable rather than it's meaning. For example:

```
int two = 2; //Well of course 'two' is 2!!
```

(this is actually an example from a real past coursework submission).
Compare that with:

```
int smallestEven = 2; //So obvious, does not need this
comment
```

Remember – names should be **readable**; they should be based on what they **represent** (not the actual value stored in them).

Advanced Aside

There is one other group of names in Java that we won't write ourselves, but will use – *package names.* A package is 'group' of fragments of Java that go together, and packages are used a lot in libraries. So, if we recall the Scanner example, we had to include this line:

```
import java.util.Scanner;
```

`Scanner` is the name of the class; but `java.util` is the package it's in. Fairly obviously package names are in all lower case.