2020 CI401
Introduction to programming

Week 1.08
Working with numbers

Dr Roger Evans

Module leader

24th November 2020

Lecture recording and attendance

- This lecture will be recorded and published in the module area
- The focus of recording is on the lecturer, not the audience
- If you are particularly concerned not to be part of the recording, turn off your microphone and camera.

- In addition, lecture attendance is now being routinely recorded (in all modules) to help the School Office monitor engagement

- (This slide is really a reminder to me to start recording and record attendance!)

Module structure (version 3)

Semester 1

Week	Topic	Theme
1.01	Introduction / Hello World	Coding
1.02	Variables, loops and choices	Coding
1.03	Input, more loops and choices	Coding
1.04	Variables and expressions	Coding
1.05	Types, assignment and arrays	Data
1.06	Let’s play Top Trumps!	Data
1.07	Objects and methods	OO
1.08	Working with numbers	Data
1.09	Simple Algorithms	Dvp
1.10	Introduction to JavaFX	Dvp
1.11	Simple Animation	Dvp
	Xmas vacation 21 Dec – 8 Jan	
1.12	GUIs using MVC	OO
1.13		

Semester 2

Week	Topic	Theme	Project
2.01	Project topics and assessment	Project	Set
2.02	Simple Inheritance	OO	Lab
2.03	Scope, Visibility and Encapsulation	OO	Lab
2.04	Testing - JUnit	Testing	Lab
2.05	Documentation - Javadoc	Doc	Study
2.06	Collections and generic types	Data	Study
2.07	IO: files and streams	Dvp	Study
	Easter Vacation 29 Mar – 16 Apr		
2.08	Numbers – the computer’s view	Data	Study
2.09	Java vs Python		Submit?
2.10	More algorithms – search and sort	Dvp	
2.11	How fast is my code?	Dvp	
2.12	Java ‘under the hood’		
2.13	Revision week		Exam ↓

Numbers

Different sorts of numbers

- Positive whole numbers (also called natural numbers)
  1,2,3,4 ….
- Roman numerals
  I II III IV V … X … L … C … D … M
- Zero
  0 (and the ‘Arabic’ number system, 1, 10, 100 etc)
- Negative whole numbers
  -1, -2, -3 ….

More sorts of numbers

- Fractions
  1/2, 3/4, 5/6 (and improper fractions 3/2, 24/5, 355/113)
- Decimals
  2.5, 0.667, -273.16 (and infinite decimals, such as 0.3333.... and PI)
- Scientific notation
  2.5e3
  This represents $2.5 * 10^3 = 2500$ – the green bit is called the significand and is always between 1 and 10 and, and the red bit is called the exponent and is a whole number.

2.5e3	2500
2.5e2	250
2.5e1	25
2.5e0	2.5
2.5e-1	0.25
2.5e-2	0.025
2.5e-3	0.0025

Numbers as patterns

- Numbers as patterns
  PIN
  Phone number

- Are these numbers? It depends on how you define ‘number’.
  - If a number is just a sequence of digits and symbols, then they count
  - If a number is something you do arithmetic with, then maybe they don’t
  - HOW you use a pattern to do arithmetic can also vary – to an accountant (1000) means -1000

Number bases

Bases and base 10 arithmetic

- In ordinary life we use the decimal system
- There are 10 digits (0,1,2,3,4,5,6,7,8,9), and you build numbers by keeping track separately of how many units, tens, hundreds etc. there are
- So 247 is shorthand for 2 hundreds + 4 tens + 7 units
- Notice having zero available is important, to keep the units, tens and hundreds lined up – 207 is not the same as 27 or 270
- We call the number of digits the base, and so the decimal system is also called base 10

Other bases – base 8 (Octal)

- In maths we can do the same thing but with a different number of digits, and get different counting systems
- For example, let’s change to base 8 (also known as octal).
- Here we have only 8 digits (0,1,2,3,4,5,6,7), and we count in units, eights, sixty fours (8*8), five hundred and twelves (8*8*8) etc …
- So 247 in octal is 2 sixty fours + 4 eights + 7 units which is one hundred and twenty eight + thirty two + seven which is one hundred and sixty seven (using words to say numbers the normal way - base 10)
- Counting in octal goes like this:
  0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20 ….
  … 67, 70, 71, 72, 73, 74, 75, 76, 77, 100 (sixty four)

Bases in Computing

- You can do this with any base (> 1) you like, though you may have to invent new digits.
- Bases used in Computing include
  - Base 10 – decimal
  - Base 8 – octal (old fashioned, but still seen for example in linux)
  - Base 2 – binary (more on this in a minute)
  - Base 16 – hexadecimal – we need extra digits, and we use letters A-F (as well as 0-9) – used for things like colours - C0FFC0 (very light blue)
  - Base 36 – used occasionally in encoding schemes using all the digits and all the letters

Computing and binary numbers

- Computers famously use binary numbers
- This is base 2, and has only two digits, 0 and 1
- In binary, we have units, twos, fours, eights, sixteens, thirty twos … and we always only have one or none of each one
- So a binary number 1011 is 1 eight + 0 fours + 1 two + 1 unit which is eight + two + one which is eleven
- And counting in binary goes like this:
  0, 1, 10, 11, 100, 101, 110, 111, 1000 …
- This works well for computers because computer hardware is based on circuits which can be in one of two states (which we call 0 an 1)

Counting in bases – decimal, octal, hex, binary

0	0	0	0
1	1	1	1
2	2	2	10
3	3	3	11
4	4	4	100
5	5	5	101
6	6	6	110
7	7	7	111
8	10	8	1000
9	11	9	1001
10	12	A	1010
11	13	B	1011
12	14	C	1100
13	15	D	1101
14	16	E	1110
15	17	F	1111
16	20	10	10000
17	21	11	10001
18	22	12	10010
19	23	13	10011
20	24	14	10100
21	25	15	10101
22	26	16	10110
23	27	17	10111
24	30	18	11000
25	31	19	11001
26	32	1A	11010
27	33	1B	11011
28	34	1C	11100
29	35	1D	11101
30	36	1E	11110
31	37	1F	11111
32	40	20	100000

Numbers in Java

Numbers in computers

The way numbers are stored and used in computer programs depends on two things:

- The sort of number that is stored (eg whole number, fraction, decimal etc)
- How much memory space is allocate to store the number

Sorts of numbers

Most programming languages have at least two sorts of numbers

- Integers – whole numbers, positive, zero or negative (signed integers)
- Floating point numbers – decimal numbers, stored using the scientific notation (significand and exponent)

Some languages have additional sorts such as

- Unsigned integers – whole numbers, which are zero or positive only
- Fractions

Java has signed integers, floating point numbers, and some special unsigned integers

Memory space for numbers

- When we write numbers we just use as many digits as we need for the particular number, short or long:
  - 3, 250, 156002
- When a computer stores numbers it uses a fixed amount of memory for each one, using extra zeros to fill up the space
  - 000003, 000250, 156203
- This is because it often needs to make space for a number without knowing exactly what the number is (or to allow for number values that change – eg variables)
- We sometimes need to tell our program how many digits (or actually bits) to allow – we call this the width of the number

Number widths

- When you specify a width for a number, you limit the range of values that can be stored
- For example, if you say the number width is 3 digits, then you can store positive numbers from 000 to 999
- If you want to store positive and negative numbers, you either need extra space for the sign (+ or -) or you allow half the range for each, and so you could only store -500 up to 499
- Usually, we store numbers in binary form, so we count widths in bits (binary digits, 0 or 1), rather than decimal
- A typical example is a number that is 8 bits wide and can store numbers from 0 to 255, or -128 to 127, and this is called a byte

Number types in Java

- Remember the primitive types in Java we talked about a few weeks ago:
  - byte
  - char
  - double
  - float
  - int
  - long
  - short
  - boolean

- All of these apart from boolean are numbers
- But why are there so many? Because they correspond to different combinations of sort and width of number

Java number types

- byte – 8 bits, signed integer (-128 ☾ 127)
- short – 16 bits, signed integer (-32768 ☾ 32767)
- int – 32 bits, signed integer (-2147483648 ☾ 2147483647)
- long – 64 bit, signed integer (-9223372036854775808 ☾ 9223372036854775807)

- float – 32 bit single precision float – (1 sign bit, 24 bit significand, 8 bit exponent)
- double – 64 bit double precision float – (1 sign bit, 53 bit significand, 11 bit exponent)

- char – 16 bit unsigned integer

Why does this matter?

- Integers can be stored exactly, but floating point numbers are only stored approximately, so you have to be careful
- Integer arithmetic may be faster than floating point arithmetic
- The width sets the range of numbers that can be stored and also the amount of memory they need – important with big arrays of numbers
- 8 bits is one byte in common memory and disk space sizes, so 64 bits is 8 bytes
- It also matters to the processor – modern processors can calculate with 64 bit numbers in parallel – older ones could only read 32, or even 16 or 8 bits at a time.

But don't worry too much!

- When you type a whole number in Java, Java will assume it is an int (unless you tell it otherwise)
- When you type a decimal number, Java will assume it is a double

- For most things that we will do this year, ints and doubles are all you need to worry about (when declaring variables etc.)

Using numbers

Basic number operators

- Arithmetic
  - `+` (addition)
  - `-` (subtraction)
  - `*` (multiplication)
  - `/` (division)
  - `%` (remainder)
- Comparison
  - `<` (less than)
  - `<=` (less than or equal)
  - `>` (greater than)
  - `>=` (greater than or equal)
  - `==` (equals)
  - `!=` (does not equal)
- We call these binary operators (a bit confusingly – nothing to do with binary numbers!) because they combine two values to make a new value: `x * y`, `a == 3`
- The arithmetic operators return a numeric value
- The comparison operators return a boolean value so you can use them in tests

Integer division and remainders

- If you divide one number by another, we are used to the idea that the answer might be a decimal, even if the original numbers are integers
- For example: 13/5 = 2.6
- In Java (and other languages), if you divide one integer by another, you get an integer result – the fractional bit is thrown away:

    13/5 = 2

- Java thinks the answer to 13/5 is "2 remainder 3", and / just returns 2
- The operator % lets you get the remainder

Using different number types with operators

- If the arguments of an arithmetic operator are the same type, then the result will be that type
- If the arguments are different types, they will be converted automatically as follows:
  - If one is a double, the other is converted to a double (and may lose accuracy)
  - If one is a float, the other is converted to a float (and may lose accuracy)
  - If one is a long the other is converted to a long
  - Otherwise, both are converted to int if necessary (and that may result in integer division etc)

Some more operators

- Unary operators (work on just one value)
  - `+` (plus - does nothing)
  - `-` (minus – make negative)
  - `++` (increment – add 1 to variable)
  - `--` (decrement – subtract 1 from variable)
  - `!` (not operator – for boolean expressions)

- `+26 (26)` `+ -26 (-26)`
- `-26 (-26)` `- -26 (26)`
- `++a` add 1 to `a` and then return `a`
  `a++` return `a` and then add 1 to `a`
- `--a` subtract 1 from `a` and return `a`
  `a--` return `a` and subtract 1 from `a`
- `!(2 > 3) (true)`

And a few more

- Compound assignment operators
  - `x += y;` (x = x+y;) – works with Strings as well as numbers!
  - `x -= y;` (x = x-y;)
  - `x *= y;` (x = x*y;)
  - `x /= y;` (x = x/y;)
  - `x %= y;` (x = x%y;)

Conditional expressions

- All the ways of making decisions we have seen have involved new statement types (if, switch etc.)
- But remember that expressions are just another way of giving instructions.
- So is it possible to put if statements inside an expression?
- YES, by using a conditional expression:

  boolean-expression?expression1:expression2

- If *boolean-expression* is true, the result is *expression1* otherwise it is *expression2*.
- For example:

```
String title = gender.equals("male") ? "Mr" : "Ms";
```

Printing numbers nicely

Formatted output in Java

- The only method for printing we have used so far is System.out.println
- This lets us include numbers in strings that we print out, but it doesn't give us much control over layout
- When we are working with numbers a lot, laying them out neatly, for example in columns, is often very important
- In Java, we can achieve this using a new method System.out.format

- Note: you may also see System.out.printf in examples – the printf method does the same as format, but format is more modern and used in places other than printing.

`format(formatString, arg1, arg2, …)`

- `System.out.format` prints `formatString,` substituting any format patterns with successive `args` provided
- `formatString` is just a String, with any content you like, but it can also include special format patterns, which start with %
- Where the % appears in `formatString`, format prints one of the args, following instructions provided after the %, and then the rest of the formatString
- There can be multiple % in `formatString` – each one uses the next arg provided (you can provide any number of args, but you need to provide enough for all the % in `formatString`)

Format example

- `%s` in a `formatString` means "print the next arg as a string"
- So if we write

  ```
  System.out.format("Hello %s%n", "Mike");
  ```

  it will print

  ```
  Hello Mike
  ```

- Notice the %n at the end of the `formatString` – this is a 'newline' character, to make printer finish the line here (like println does)
- If we write

  ```
  System.out.format("Hello %s and %s%n", "Mike", "Mary");
  ```

  it will print

  ```
  Hello Mike and Mary
  ```

Formatting numbers

- Format is most useful for printing out tables
- The format element %d is used to print out a decimal (ie base 10) integer, and %f is used for a floating point number
- Both of them allow information between the % and the letter to specify how the number should be laid out
- %4d says “print the next arg as an integer, adding spaces to make it at least 4 characters wide”
- %8.2f says “print the next arg as a floating point number, 8 characters wide in total, and with 2 characters after the decimal point”

Format example 2

- So if we write

```
System.out.format("%4d %s %8.2f%n", 1, "Mike", 15.49);
System.out.format("%4d %s %8.2f%n", 16, "Mary", 234);
System.out.format("%4d %s %8.2f%n", 133, "Joe", 5.40);
```

  it will print

```
   1 Mike    15.49
  16 Mary   234.00
 133 Joe     5.40
```

- Notice in the first two lines how the number columns line up neatly, but in the third line, the shorter name messes up the alignment of the last number

Format example 2

- We can fix this by adding a width to the %s element, and also a '-' which tells it to line up its content on the left hand side of the column, making it %-4s:

```
System.out.format("%4d %-4s %8.2f%n", 1, "Mike", 15.49);
System.out.format("%4d %-4s %8.2f%n", 16, "Mary", 234);
System.out.format("%4d %-4s %8.2f%n", 133, "Joe", 5.40);
```

so that it will print

```
   1 Mike    15.49
  16 Mary   234.00
 133 Joe      5.40
```

Formatting elements quick guide

- There are lots of formatting elements. The basic structure is:
  - Starts with % (always)
  - If there's a – (minus sign) next, it means 'left-justify' in the column (line up on left-hand side)
  - If there's a number next, it specifies the total width of the column
  - If there's a . (period) and number next, it specifies the precision (number of digits after decimal point etc)
  - It (always) ends with a conversion type character – common ones are:
    - s – string, d – integer, f – float, b – Boolean, % – actual % sign, n – newline
- Check out documentation for Java formatter for more details

Creating formatted Strings

- Another version of the format method is available to create a new String, instead of printing out the formatted text
- It is called String.format, and is used exactly like System.out.format, except that it returns a String value
- So if we write

```
String s = String.format("Hello %s\n", "Mike");
```

then `s` will contain the string `"Hello Mike"`

Additional reading on formatting

- https://docs.oracle.com/javase/tutorial/java/data/numberformat.html
  - a short tutorial introduction to the format and printf methods

- https://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html
  - the official definition – see the section ‘Format String Syntax’ to see EVERYTHING you can do (it’s a lot!) --

Extras

Writing integers in Java

- Decimal integers
  - Digits 0-9
  - Mustn't start with zero
  - Eg 35472, 22
- Binary integers
  - Digits 0-1
  - Start with 0b or 0B
  - Eg 0b1011, 0B11011110101010
- Hexadecimal integers
  - Digits 0-9, A-F (upper or lower case)
  - Start with 0x or 0X
  - Eg 0xFFFF, 0x4c507FFBC00
- Octal integers
  - Digits 0-7
  - Start with 0
  - Eg 0664, 07
- General
  - All integers default to int type (32 bit, signed).
  - Suffix with l or L to make them long (64 bits)
  - Eg 1453673L, 0xFFFF0000L

Writing floating point numbers in Java

- Decimal floating point
  - Digits 0-9
  - General form: 12.3456e12
  - At least one digit and either . or e/E
- Hexadecimal floating point
  - Start with 0x or 0X
  - Use p/P for exponent
  - Eg 0x13F2p20
- General
  - Suffix with f/F or d/D to force float (32 bit) or double (64 bit)
  - Default is double
- All numbers
  - Underscores allowed between digits (eg 123_000_000)
  - Minus sign is not part of the number – it's an operator

Chars

- The primitive type char is used to represent character codes
- A char is a 16 bits wide, unsigned integer
- Characters on screen, in documents etc. are each represented by a numerical code (remember – everything is a number/pattern)
- Unicode provides a standard for thousands of character codes in different languages, typographies etc
- The ‘ordinary’ English ones (more properly known as Latin characters) have numbers between 32 and 127
- In Java, you can write chars using single quotes:

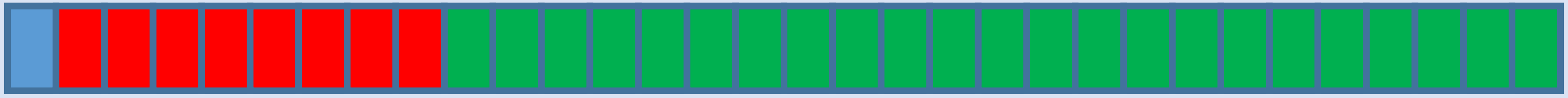
```
char c = 'a'; // character a = code 97
char C = (char) 'a' - 32;  // maps to uppercase (A)
```

`==` - 3 things to watch out for

- == is not the same as =
  - == compares two values to see if they are equal
  - = assigns the result of an expression (a value) to a variable
- == between decimal numbers is not reliable
  - `1.2 + 1.2 + 1.2 == 3.6` ?
- == between strings is not reliable
  - We use .equals instead, like this:
    `place.equals("London")`
    if place is a variable containing a string

Floats in memory

Float – 32 bits



Significand – 23 bits, top bit is always 1 so not represented

Exponent – 8 bits, biased by 127

Sign – 1 bit

Example

2.5e3 = 2500 = 100111000100 = 1.00111000100b1101

0	1	0	0	0	1	0	1	0	0	0	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

+ 11+127 = 138
  = 10001010

1.00111000100 -1 = 00111000100000000000000

Big numbers

When approximate answers are not enough

- Java programs can give ‘wrong’ answers to arithmetic calculations using number primitives for several reasons
  - Floating point numbers not being stored exactly
  - Approximation when converting from decimal floating point in code to binary form (Eg 0.2 decimal is a recurring binary number 0.001100110011 … and so cannot be exactly represented)
  - Loss of precision when converting between ints and doubles
- Sometimes it is really important to be accurate – for example when dealing with money
- Java provides special number classes BigInteger and BigDecimal to allow for this

BigInteger

- BigInteger allows you to create and do calculations with very large integer values – theoretically no limit to how big they are:

```
import java.math.BigInteger;
…
// make a BigInteger from a String
BigInteger i = new BigInteger("1234567890");
// multiply it by 3
i = i.multiply(new BigInteger("3"));
// print the result
System.out.println(i);
3703703670
```

BigDecimal

- BigDecimal allows you to create and do calculations with high precision decimal numbers, with accurate results

```
import java.math.BigDecimal;
…
// make a BigDecimal from a String
BigDecimal d1 = new BigDecimal("1.234567890E-5");
// or make a BigDecimal from a double value
BigDecimal d2 = new BigDecimal(1.5E9);

// divide one by the other
BigDecimal d3 = d1.divide(d2);
// print the result
System.out.println(d3);
8.2304526E-15
```

Using BigInteger and BigDecimal

- BigInteger and BigDecimal allow you to have as much precision(number of digits, not including trailing zeros) as you like
- They give accurate results, or an error if they can’t (for example a recurring decimal number)
- They have all the standard operations as methods (not operators) – add, subtract, multiply, divide
- They are not as fast as primitive number calculations

Lab exercises
Week 1.08

Lab exercises

- Lab1 – some little exercises with different kinds of numbers and operators
- Lab2 – A short format exercise
- Lab3 – A challenge exercise using the Big Numbers section
- Lab4 – an advanced exercise to create a new type of number!!