

Introduction to Programming

Michael Charleston

michael.charleston@sydney.edu.au

Semester 2, 2011

Contents

1	Introductions	3
1.1	Administration	3
1.2	People involved	5
2	Assessments	6
2.1	Tasks	7
2.2	Assignments	8
2.3	Quizzes	9
3	Good Practice	10
4	First Programs	12
4.1	What is a program?	12
4.2	Learning programming is a challenge	13
4.3	Pseudocode	16
4.3.1	Conventions in Pseudocode	17
5	Variables	19
5.1	Declaring Variables	19
5.2	Initialising Variables	20
5.3	Boolean	20
5.4	Assignment	22
5.5	Casting	24
5.6	Scope	26
6	Operators	28
6.1	Definitions	28
6.2	Increment	29
6.2.1	Increment operator ++	29
6.2.2	Comparing primitives	30
6.2.3	Comparing Strings	31
6.3	First mathematical operators	33
6.4	Boolean Logic	34
7	Arrays and Iteration	35
7.1	Arrays	35
7.2	Iteration through an array	39
7.3	Multi-dimensional arrays	41

8 Control Flow	48
8.1 if	49
8.2 if ... else	50
8.3 switch	52
9 Loops and iteration	55
9.1 while	55
9.2 for	56
9.2.1 initialisation	58
9.3 do...while	58
9.4 break	59
9.5 continue	59
9.6 foreach	59
A Lab work	59
A.1 The OneOff Template	59
Lab 1	60
Lab 2	62
Lab 3	73
Lab 4	75
Lab 5	76
Lab 6	77
Lab 7	78
Lab 8	80
B Tasks	81
Task 1	82
Task 2	83
Task 3	84
Task 4	85
Task 5	86
C Assignments	87
C.1 For both assignments	87
D Assignment 1	87
D.1 The main task	87
D.2 Stages	87
D.3 Requirements	89
D.4 Assessment	89
Index	91

1 Introductions

1.1 Administration

INFO1103

- This unit is *Introduction to Programming*.
- You will, I hope, by the end of this Semester, be able to write simple programs in Java and understand the basics of programming in general.
- This is a standard unit (i.e., not Advanced) but there will be extension material to keep everyone challenged.
- You will be expected to come to lectures and take notes, ask questions and engage with the subject!
- I will do my best to teach you all — but *you* have to take responsibility for your own learning.

The unit is taught using Java, one of the most commonly used languages today. Java is very well supported on most of the platforms you can think of, and is largely platform independent: you can write your code on whichever machine you like and it should work without fuss on a different machine, just so long as it has a compatible version of Java on it.

At the time of writing this, the current version is Java 7, but my computer is running Java 1.6, otherwise known (for some reason) as Java 6, as I can find out by typing `java -version` on my terminal:

```
> java -version
java version "1.6.0_24"
Java(TM) SE Runtime Environment (build 1.6.0_24-b07-334)
Java HotSpot(TM) 64-Bit Server VM (build 19.1-b02-334, mixed mode)
>
```

Where am I?

If you're wondering what you're doing in this unit, maybe something here will help:

- This unit is the lead-in to ALL the technical units in the School of IT
- This unit *does* require you to program!
- This unit has *no prerequisites*¹
- INFO1103 is not a simple version of INFO1903 – these two are *completely independent*.
- (or possibly...) Actually yes, you've walked in to the wrong room.

This unit

The unit is delivered in 13 weeks of lectures, with labs that you should attend. A register will be taken, and your attendance will be monitored. Many of the *assessments* will be held during labs: missing them will not be a good idea as you'll miss out on the marks!

If you engage with the material, come to lectures, and *practice*, you will probably pass. ☺

If you think you can learn it all by just reading the book and not programming, you will probably fail. ☹

Plagiarism — submitting someone else's work as your own — will not be tolerated. You MUST read and understand the University's policy documents on *plagiarism*. We use electronic means to identify potentially plagiarised work. You have been warned.

It is important to know *now* what is in store for you this semester, so you can plan your studies.

You should also get stuck in as soon as possible to the assignments and tasks, as well as trying out exercises from the textbook, so you can get as much practice in programming as possible.

¹Except you know, having finished High School reasonably well and such. But note: we don't assume you can already program.



I'm writing this while in the period of investigating students who've been under suspicion of plagiarism for their assignments in INFO1x05 in 2010, S2.

It seems that even though we say things like *you must read the policy*, many of you don't. This is kind of dumb. I mean, really: if someone tells you what you need to remember to avoid getting into some serious trouble, then you'd want to know that, right? For many, apparently not.



Learn the policy The policy document will enable you to know what the boundaries are of collaboration, and sharing knowledge. You are *welcome* and *encouraged* to talk with your friends, your tutors and your lecturers to get help. **But what you submit must be your own work!**



We aren't stupid We have met this practice before. We know what plagiarism looks like. We are pretty smart people and we've been doing this for a while. We have *very* clever programs that we use to detect plagiarism, and we do look at code. The chances of you getting away with it are minimal, and it's just **not worth the risk.**

Yes, it matters It really does matter. If we catch you submitting work that is so similar to someone else's that we cannot account for the similarity, then you will be penalised. There are two kinds of plagiarism in the Policy. Negligent, and Dishonest.

Negligent Plagiarism is when you really didn't know that what you were doing was wrong. You might have loaned your USB stick, with your code, to someone, just to look at, and even stood over them to make sure they didn't copy your files². You might have left yourself logged in to a lab computer while you left the room to get some lunch³ You need to take responsibility for your own learning and *not put yourself at risk*. That's right: you can be found guilty of negligent plagiarism *just by sharing your work with others.*

If you are found to be guilty of *negligent plagiarism* then a number of things happen:

1. You will be given a copy of the policy and be required to sign a letter saying you've read and understood that policy. You cannot be found to be guilty of *negligent* plagiarism after this, because now you'll have signed this document.
2. You would typically lose the marks for the assessed work that was found to be plagiarised.
3. In the School of IT, your name would also go on an internal register of names of people who've been found to be guilty of negligent plagiarism. This register is kept inside the school and the finding would not appear on any official University documentation. You can think of this as the "One Warning" option. This does *not* imply that if it's your first time you automatically get the "negligent plagiarism" finding: you can go straight to dishonest plagiarism, as below.



Dishonest Plagiarism is when you *did* know what you were doing was wrong, and ***did it anyway***. If you are found by our investigation to be deliberately plagiarising someone else's work and submitting it as your own, in full knowledge of the University Policy, then this is *dishonest plagiarism*. Then your name could be reported to your Faculty, and possibly to the University. If you are found guilty of this more serious form of plagiarism then this *does* appear on your University Transcript.

²Oops

³Wow, that was really stupid.

It is important to know *now* what is in store for you this semester, so you can plan your studies. Planning is a Very Good Idea™.

Assessment

Assignments and Tasks are *individual* work.

You must get $\geq 40\%$ of each major component to be permitted a pass: the major components are the *progressive mark* (Tasks, Quizzes, and Assignments combined), and the *exam mark*.

You must also get a combined mark of at least 50% in total, of course⁴.

Some Symbols

- I will put in the occasional mathematical symbol too: \leq , $>$, \leftarrow , \nearrow , which I'll expect you to know or learn.
- Emphasized things *look like this*.
- **This is very dangerous**. If you persist in this kind of practice your programs will probably fail horribly.
- **Caution!** This is a warning!
- You can easily go wrong here. This is something you do need to take substantial care with. It's not *wrong*, but it's *risky*.
- **Don't Ever Do This!**



1.2 People involved

About me

- I am your lecturer for the whole unit.
- my e-mail: [michael \[dot\] charleston \[at\] sydney \[dot\] edu \[dot\] au](mailto:michael.charleston@sydney.edu.au)
- phone: (02) 9351 4459 (direct line); x14459 (extension)
I hate voice-mail
- office: Room 412 in School of IT



Contacting me

You should definitely contact me if you're having problems, or if you'd like more challenging things to do.

You should *not*

- write me an e-mail that goes "hi im in ur class can i get sum help": incomprehensible or merely slobbish mail will be *ignored*.
- expect an immediate reply. I get lots of e-mail and it's not feasible to . If you don't hear back within a day, don't panic. If it's urgent, phone. If a week goes by then try again. I'm not trying to ignore you, I just have LOTS to do.
- write to me at the end of the semester and say "I don't understand that thing you did at the beginning": I shall be unable to help you because of laughing too much.
- expect to pass just by showing up.
- beg for more marks at the end of semester after doing really badly in the exam.

⁴If you are having trouble working this out, you are already in trouble.

About your tutors

- Your tutors have been hand-picked and quality selected to give you the best possible help in learning to program.
- You *must* take advantage of having them around.
- Make sure you learn your tutor's name and get his or her e-mail address.

About you

- You're enrolled in INFO1103 (I hope)
- or you're just interested in programming
- You have probably not done (much) programming before
- You *may* have ... **ahem** just failed a programming unit...
- You are highly motivated to pass — maybe even to learn!

When you want help

1. Look at the notes, slides and [course text](#)
2. Ask your friends (don't copy their code! you won't LEARN!)
3. Ask your tutor
4. Ask me! I have a free hour every week for students to come and ask me anything: make use of it. The time is **Tuesday 10am**. If you can't make that time then e-mail me and we can arrange an appointment.

There's one other character we should get to meet. This is Kurt.



Actually the real name of this beautifully redrawn character is Lehrer Lämpel ("Teacher Lämpel"). He's a teacher from a German children's story called Max und Moritz, and he's been redrawn by user Stefan-vonHalenbach and uploaded to [openclipart.org](http://www.openclipart.org). The full URL is <http://www.openclipart.org/user-detail/StefanvonHalenbach>. The original sketch, by Wilhelm Busch (1832-1908), is in the Wikimedia Commons and can be found at http://en.wikipedia.org/wiki/File:Max_und_Moritz_%28Busch%29_040.png. Kurt (I'm pretending his first name is Kurt but it might not be) will give the occasional comment throughout these notes. Pay attention!

2 Assessments

Assessments

It's worth noting that when you aim to get through this unit it's extremely unhelpful to think of it independently from the other units in your degree. This is a foundational unit that will contribute to your ability in all the other IT units: so even if you intend merely to *pass* you should aim to do *well* in this unit.

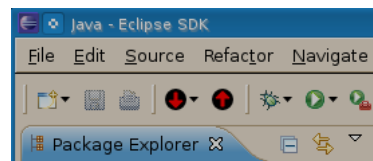
Another thing worth noting is that if you're aiming for a particular grade, along the lines of "well I've got 60% of the 50% so far, so I only need to get 40% in the exam," you're very likely to, well, *miss*. Don't aim at a grade! Aim to get all the marks you can.

Oh and one more thing. Ask yourself: why are you doing this unit? Are you at University to get a degree, or get more actual ability? Are you out to learn, or just to pass?

Date	Week	Assessment	Value	Notes
07.25	1	—	—	Labs begin
08.01	2	—	—	Complete A1.1 (simple iteration)
08.08	3	Task 1	2	SimpleCalc (very easy)
08.15	4	Quiz 1	5	Complete A1.2 (more iteration & logic)
08.22	5	Task 2	2	Sorter (straightforward)
08.29	6	Assignment 1	10	Lines (straightforward)
08.31		Census Date: last day to discontinue without \$\$ penalty		
09.05	7	Task 3	2	GeneScan (tricky in parts)
09.09		Last day to discontinue without fail		
09.12	8	Practical Test	10	Live coding in the lab
09.19	9	Task 4	2	MediaLibrary (moderate)
09.26		Semester 2 break		
10.03	10	Quiz 2	5	10.03 Public Holiday
10.10	11	Assignment 2	10	EcoSim (moderate)
10.13		Last day to discontinue		
10.17	12	Task 5	2	TBC (challenging)
10.24	13	Revision	—	
exam period		Final Exam	50	closed book



Table 1: Assessments and important dates this Semester

Eclipse Support with *katipo*

(a) *Lactrodectus katipo*

(b) plug-in for Eclipse

↘ Do not confuse these! ↗

- Nicholas Jefferson (PhD candidate, SIT) has written a plug-in for Eclipse that you will use to get (“pull”; ) all assignment templates.
- Further, Tasks 1-5 and Assignment 1 will have testing run on them every time you save (“push”; ) your work.
- Assignment 2 won’t have much in the way of unit testing as it will be marked in a different way.

2.1 Tasks

Tasks

5 × 2% — throughout the semester

Tasks are mini-assignments that should take only a few hours to do, and contain no more than a (few) hundred lines of code each.

They are partly marked by the tutor in the lab, but you won’t get your mark then: there will also be a component of your mark from an automatic tester, and the total marks will be compared across labs to ensure the same standards are used throughout, before the final marks are released.

For these Tasks 1 mark out of 2 is a straight *pass*: you have to do excellent work to get 1.5 or 2.

You will have to explain your code to your tutor, in the lab, to get full marks: until you do so you will get 0.

The automatic marking will be based on tests run automatically, similar to the ones you'll see whenever you push your Task work. If automatic marking is used, it will contribute half the mark. All Tasks and the first Assignment will be auto-marked. However if your code does not even compile, you will get 0 for that assessment, no matter how pretty the code is...

Task 1**2% — Week 3**

Write a Java program to read in a sequence of integers from the command-line and print out the following quantities:

1. the number of integers read in
2. the maximum value of the integers
3. the average value — which need not be an integer!
4. the maximum difference between any of the integers

Task 2**2% — Week 5**

Write a Java program to read in a sequence of integers, as before, but to do more with them:

1. store the integers in an array with `int []`, *not* using the `ArrayList` class;
2. sort the integers from highest to lowest value;
 - you'll have to implement a sort method based on pseudocode
3. print them out in order.

Task 3**2% — Week 7**

Write a Java program to read in data from a file and collate it into a report:

- read in the data from a text file in a simple format;
- store the data items in individual objects of a class you created;
- print a report on the data to a new file.

Task 4**2% — Week 9**

Write a Java program with multiple Classes, that maintains a collection of Media objects in alphabetical order of their title (which you may assume is unique):

1. Store CD, DVD, Book
2. Perform operations on selections of the library
3. Export the library as XML and then recover it

Task 5**2% — Week 12**

TBC: but something challenging involving trees, quite possibly, or solving logic problems.

2.2 Assignments

There are two individual assignments this Semester, which you should start as soon as you're able. These will be much more substantial pieces of work, and are worth 10% each. Thus you'll have about 30% of your mark from this unit based on programs you write.

As a rough guide these should not require more than say 1000 lines of code. This might seem like a lot, but really, they do mount up quickly. They will definitely take up quite a bit of time.

Don't ask your private tutor to do it for you.



Assignment 1**10% — Week 6**

Assignment 1 has three main stages and you are strongly encouraged to use the dates for these Stages in Table 1 as *deadlines*. The actual deadline for Assignment 1 (also, “A1”) is in your lab in Week 6.

You must get your assignment skeleton code using the “pull” button in Eclipse.

Assignment 1 requires you to complete a program that plays a simple game: the object is to fill up a 4x4x4 “board” with counters in your own colour to get a line of 4.

- Stage one: Your program must test whether a given input is *valid*.
- Stage two: Your program must check to see if one player has won.
- Stage three (final): Your program must be able to propose a valid move, given a board.

Assignment 2**10% — Week 11**

This will be a simulation.

You will have to create a program that simulates a toy system with random behaviour that models two populations of creatures: predators and prey. (Predators eat prey.)

Your goal will be to keep the whole ecosystem alive as long as possible!

Assignments: A little competition

Your assignment marks will be *standards based*: in order to get a passing mark for each you must fulfill the same requirements as everyone else. These will be very straightforward.

However, to get the very best marks you must also do better than your classmates. For Assignment 1 this means you must create the best game player possible. For Assignment 2 you must keep your ecosystem alive the longest. Not much of the mark will be from this, but bear in mind in order to get the best marks you’ll have to think a little differently and compete!

2.3 Quizzes**Quiz 1****5% — Week 4**

This will be an online quiz held during one hour of the lab. There will also be usual lab exercises.

This will be closed book.

Topics assessed: material covered in the first weeks lectures, up to Week 3 — including:

- Variables
- Assignments
- Arrays
- Expressions
- Pseudocode
- Control flow

Quiz 2**5% — Week 10**

The second online quiz, covering the material up to that taught in Week 9, including:

- loops
- methods, arguments and return
- writing a class
- constructors
- access modifiers
- generalization
- inheritance and polymorphism...

Practical Quiz**10% — Week 8 — barrier**

There is one *practical test* in Week 8.

- This will be a 50 minute quiz, during which you must write code in Java.
- Your work will be submitted electronically using Blackboard and marked afterwards.
- During the practical you *must not use any reference material at all* (except that which is provided within Eclipse): this includes, but is not limited to, web browsing, instant messaging, phoning or sending or receiving text messages.
- **Any student using any means to look up the answers will instantly get 0 for this assessment and may face further penalties under the University of Sydney's policy on Academic Dishonesty.**

**If you fail the Practical (again, and again)**

- You will get a practice run at the Practical the week before.
- If you fail you will get at most two more tries (with different tasks of course).
- These will be in the following two weeks (times to be arranged)

If you cannot pass the Practical after three tries the highest grade you will be able to get in this unit is a pass (P).

If you fail this test you will be given the opportunity to try again in each of the two following weeks. If you cannot pass this test after three attempts, your grade will be capped.

As an example or practice, you might implement a sorting algorithm which was given to you in pseudocode.

If you've never heard of the ACM Programming Challenge, check it out now. The problem archive hosted at Baylor University is [here](#).

The maximum grade you can get for INFO1103, if you cannot pass the practical quiz, is a pass (P). You can't be considered to be a Credit student if you cannot learn this!

The practical test will be relatively straightforward and will test your understanding and application of the material covered in the weeks prior to the week in which the test takes place.

If you don't pass the test in the first week then you'll have a chance to try it again the following week, and then if you fail *again* you'll have one last try in the week after that. Please, please, pass on the first try! You *won't have time* to spend trying tests over and over.

Final exam**50% — exam period**

- The exam will be 2 hours long with 10 minutes reading time
- It will contribute 50% to your final grade.
- You will be permitted 1 A4-size sheet of paper with notes on it (yes, both sides)⁵.

3 Good Practice

Good practice

You should *definitely*

⁵unless you have Möbius paper but that's your own fault if you do

- use references, e.g., the API
- back up your code regularly, e.g., on the School of IT servers
- don't delete code you don't like: commented it out and add a note to explain why it's wrong
- don't copy your code to or from someone else for assessed work
- practice practice practice writing code
- test everything
- comment anything non-trivial

You are strongly encouraged to use on-line resources to research concepts and examples for this unit (and for just about anything else you can think of). Here's a quick list of places to also get to know, in no particular order:

- [the Java Tutorials](#)
- [Google](#) and its friend [Let Me Google That For You](#)
- [xkcd](#) for some excellent geek humour
- [StackOverflow](#)
- [Coding Bat](#) for many simple examples to get you programming in the small

Note that research is *not* the same as plagiarism. We like you to share ideas but not code. We like you to look things up, but not to download it and submit it as your own work.

If you're unsure of what you can legitimately do when helping each other, ask your tutor or me.

Backing Up

Backing up is not the same as archiving.

When you are working on something that's important, e.g., an assignment, you should definitely keep a spare copy, in case something bad happens to the one you're working on.

Bad things include:

- hard disc failure
- losing the USB stick
- theft
- deleting the whole thing in frustration
- incompetence

WHEN, not IF

It's **WHEN** you lose your work. Not *if*.

When you lose your code it must be possible to recover it somehow. The School of IT servers are backed up, so as a *start* you should keep your code there.

Make sure you keep versions of your code too. One (very laborious) way is to regularly save a version of your work and give it a date label.

Get a **RAID** at home, install **Subversion** or **Git**, make copies on floppies, mail each version to yourself, print it out and stick it on your ceiling, but *back it up*.⁶

I had a colleague — not in the School of IT I must add! — whose laptop died. Completely died. He had no backup, had never backed up. He was pretty upset.



Testing Code

We will spend more time on this later, but for now, just try to remember that you *cannot* guarantee your code will work!

- Not the first time

⁶Some of these methods are not actually very good. But you should see my point.

- Not for every case
- Even if it compiles beautifully

Syntax and Logic and Compiler

When you write programs you have to write them in such a way that they make sense to the compiler – there's a *syntax*, that is, a set of rules, that you must follow in order for this to be true.

We have the same thing in English: we can't just make up a sentence of random words and expect it to make sense.

randomizing it's of enough words bad order the just the.

Tguhoh aaltpeprny if you lvaee the fsrit and lsat ltetres in pacle it's ok

Exercise: write a program to convert the input arguments (Strings) into muddled words as above: leave the first and last characters in place for each word and randomize the rest.

Syntax is not Logic

However even if we have the correct *syntax* we may still not make sense. This is a very famous syntactically correct yet meaningless sentence:

Colorless green ideas sleep furiously

Noam Chomsky

In most cases your compiler will complain because of *syntax* errors, but it will also sometimes give you an error message if a variable is unused, or used before it's been initialised, or if there is part of the code that is unreachable. These are *logic* errors. The compiler won't be able to pick up on every logic error (else there would be no need for debugging!) but it will help you find simple logical errors. Pay attention to those compiler messages!

4 First Programs

Finally, we can begin!

You may have just been wondering things like “well, when are we going to get started with things I don't know?” or “what *is* a program anyway?”. There is a big range in the backgrounds of students coming in to this unit so bear with us while we cover some ground that might be old news to you.

4.1 What is a program?

It's a set of instructions that were written by a programmer (or very occasionally by some other program) that tells a computer what to do. Computer programs come in all shapes and sizes, from the very simplest ones we'll look at below (the “Hello, World!” program is the most well-known) to extremely large, integrated systems with multiple functionality controlling factories, big business accounts, banks, world markets, scientific research, etc. etc.

All programs are written in some kind of programming *language*, and we're going to use the Java language here because it's pretty well established, works across platforms, and has some nice features. It's not designed to be a teaching language (or more importantly a *learning* language – one that's the best first language for students to learn) but then again, there are advantages in its extremely large collection of online resources, excellent on-line tutorials originally hosted by Sun and now by Oracle, and masses of physical and e-books.

Java is a *compiled* language, which means that the code you'll learn to write (and write well, I hope) must be converted into a language that the computer can understand. In Java this is called *byte code*.

You'll hear other terms like "machine code" and "assembler" or "assembly language", which are not the same thing as byte code but all have the characteristic that they're not meant to be human-readable, they're for the computer to understand.

We'll learn a lot about the various features of Java throughout this unit. It's not a unit that is *about* Java though: it's a unit about *programming*, and we're using Java as a good language for you to learn.

4.2 Learning programming is a challenge

Why is this? Because you basically have to learn everything at once.

You can't write a program that does anything without knowing something at least about a lot of things, any more than you should drive a car without knowing about the accelerator, brake, clutch, gears, road rules, indicators... You can find a nice place to practice driving where you have to worry about as few of these things as possible, but you can't get going without sitting in the driver's seat and trying it — so let's get started!

Your First Program

The *classic* program of all time is "HelloWorld". In Java it looks like this:

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

How does it work? We'll look at the ingredients in detail.

First, let's look at some different ways of writing the same program:

C

In old (1978) C it would be more like this:

```
1 int main() {
2     printf("Hello, World!\n");
3 }
```

but in current ANSI-C would be like this,

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     puts("Hello World!");
7     return EXIT_SUCCESS;
8 }
```

C++

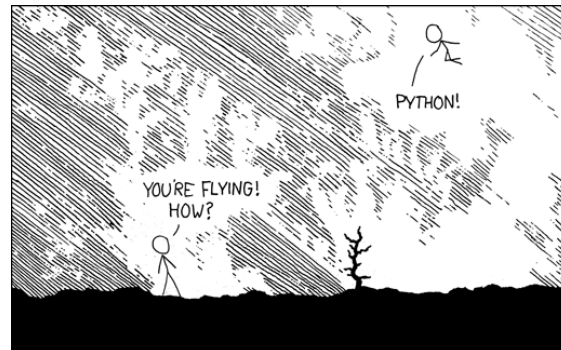
in ISO-C++ it would be like this,

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello World!" << std::endl;
6 }
```

Perl

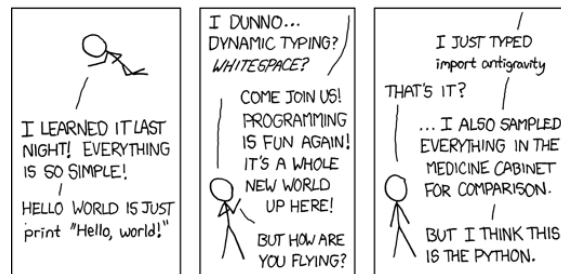
in Perl like this:

```
1 print "Hello World!\n";
```

Python

and, last but not least, in Python, like this:

```
1 print "Hello World"
```



Sources: [The Hello World Collection](#) and [xkcd](#)

In general...

What we have for all of these programs is something like the following:

```
1 <load things we need> // "optional"
2 <begin a method>      // explicit or implicit, "main"
3 <print the string>    // definitely needed
4 <end the printed line somehow>
5 <end the method>     // as in line #2
```

Java syntax

In order for the compiler to turn your code into a working program, the code has to obey a lot of syntax rules. You will pick up many of these as you go along, but let's just list a few now too:

- Some words are *reserved* — e.g., **public**, **static**, **void**, **int**, **float**. You can't use these for variable names.
- Variable names can't begin with numbers.
- Blocks of code are delineated with braces { }
- Expressions are delineated with parentheses ()
- Array items are accessed with brackets []
- Statements should end with semi-colons ;
- Strings are limited by double quotes " " (not single ones or curly ones)

It's very easy to get things wrong. Don't worry. Most things are fixable. [DIVE IN!](#)

Hello, World! in Java

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

public: this is an *access modifier*: here it means anything can call the “main” *method*.

static: this indicates that the main method isn’t tied to any particular object of its class. Don’t worry about this for the moment.

void: this indicates that the main method doesn’t return any value.

main: this is the name of the method. In order for a Java program to work there has to be a main method.

String [] args: this is saying that the main method takes an array of String objects as its *argument* (we’ll learn about what arguments are later).

System.out.println: this is a method (yes, a method *within* a method) that prints the “Hello, World!” string of characters.

So many ingredients

There are many many ingredients to even this simple program, and you’re still not done. This unit will give you the tools you need to write and run programs from the very simple ones like HelloWorld, right up to much more complex ones to play games and run simulations.

There is a *lot* to remember but persevere and you will do fine.

For the moment we will not be thinking much about classes and objects, but they will be there in the background so don’t worry about them, just focus on the rest of the coding.

In the second part of the course we will put programming in to an Object-Oriented Framework.

The important bits in Hello, World

The main part that you have to worry about and understand is the line

```
3     System.out.println("Hello, World!");
```

which does the real work. System.out is a special variable, an *object* of a *class*, that has a *method* defined for it called println. It’s the println method that actually prints a string of characters to the console.

println also puts a newline at the end of whatever it prints, so you don’t have to add a newline character (‘\n’, or, within a formatted string, the %n character), all the time.

Note: the %n character is for use within a formatted String, not for within a normal String. Forget I mentioned %n. Using the %n character in a String object directly fails miserably: just use the \n.

The \n is called an “escape character”: it’s a combination of symbols that combine to form a single character with a special meaning. There are several commonly used ones, as follows:

Character	Meaning
\n	new line / line feed
\r	carriage return
\t	tab
\a	bell
\b	backspace

The carriage return `\r` is used to end lines in old Mac OS; Unix and Linux and new Mac OS use a `\n`; DOS and Windows use a combination of both. This isn't *intended* to be a pain, it just *is* a pain.

You will probably use the tab and newline characters very frequently. If you're using Java to program for different machines it's probably a better idea to use the built-in special character "`%n`" in a formatted String, which is device-independent. You won't be tested on this.

Making a Program Go

In order to make a Java (or C, C++, but not Python) program actually do anything you have to *compile* it. This is a process done by a *compiler* that converts your human-readable (Java) code into machine-readable *byte code* or *machine code*.

To *compile* a single Java file, e.g., `HelloWorld.java`, you type:

```
> javac HelloWorld.java
```

(followed by a Return) and that should produce no output, which means there aren't any errors detected by the compiler.

This makes a *class file* for your program, which is where the bytecode is (all being well).

Running it

To *run* the program you type

```
> java HelloWorld
```

where it's assumed that `HelloWorld.java` has a proper `main` method in it.

Later on you'll be compiling multiple files at once to make more complex programs: for that you will type something like

```
> javac mydirectory/*.java
```

4.3 Pseudocode

Pseudocode

Pseudocode is a convenient way of describing how a program or algorithm works, without being in any particular programming language.

The main goal of pseudocode is that you should be able to convert pseudocode into a program.

You will do exercises on this, and you will be expected to be able to write out an algorithm or part of a program in pseudocode.

Pseudocode

Algorithm 1: HelloWorld

```
1 print "Hello, World!"
```

Java code

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         System.out.println("Hello, World!");
4     }
5 }
```

(Well yes, this *is* a good deal uglier than the pseudocode.)

Going from pseudocode to actual program code will become easier as you practice.

As an exercise, try to work out what the very simple algorithm below is doing:

Algorithm 2: Mystery (A, B) : C

```

1  given a set  $A$ 
2  given a set  $B$ 
3   $C \leftarrow A$ 
4  for each ( $x \in A$ ) do {
5  ·   if ( $c \in B$ ) then {
6  ·   ·   remove  $c$  from  $C$ 
7  ·   }
8  }
```

Pop Quiz!

Question 1: At line 3, what kind of variable is C ?

Question 2: At line 3, what is the value of C ?

Question 3: What does the \in symbol mean?

Question 4: At the end, what is special about the value of C ? Write it out in set notation (this should give you a hint...)

Pseudocode guidelines

- Pseudocode is not code! Don't put language-specific commands in it.
- Make things as clear as possible
- You can be brief with obvious procedures
- Use mathematical symbols to make things clear

4.3.1 Conventions in Pseudocode

For the purposes of this unit (and perhaps beyond) it will be a good idea to stick to some conventions when writing pseudocode. So here they are!

- Each procedure should have a useful name. Don't call your algorithm "algorithm 1" or "foo", nor "myReallyLongAlgorithmForAnsweringThatQuizQuestion".⁷
- Use indentation to indicate blocks of instructions: if you don't use *block delimiters* then indentation is the only way you can indicate a block.
- Use the "gets" assignment operator, " \leftarrow ", not " $::=$ " or (which would be bad), " $=$ ". Remember, pseudocode is not code!
- To access properties of a variable use a dot/full stop/period ".". E.g., the length of a list L might be written $L.length$, and given a pair p of elements (x, y) , we can write $p.x$ and $p.y$.
- To indicate the i -th element in an array, use square brackets "[,]". E.g., $A[4]$ is the 4th element in array A .
- End loops and if statements clearly, either with a block delimiter like { and }, or "end-if", "end-for", "end-while" etc.
- If you can use a simple mathematical formula clearly, go ahead: e.g., rather than $x \leftarrow \text{sqrt}(y)$, you can write $x \leftarrow \sqrt{y}$.
- It is fine to use comments! Indicate these in a clear and consistent way. As we're using Java in this course it makes sense to use something like $/**$ which, in Java and C++, means "the rest of this line is a comment", or $/* ... */$ which in both these languages means "everything within these symbols is a comment". Alternatively you can use a \triangleright symbol as an alternative to the $/**$.

Pseudocode Examples

Here are some simple examples of pseudocode to give you the idea.

Algorithm 3: InsertionSort (A)

```

1  for ( $j = 2$  up to  $A.length$ ) do {
```

⁷As with all we teach you, don't assume we've made a mistake when we don't practice what we have told you! There may well be a *deep pedagogical reason* for it.

```

2   ·  key ← Aj                                // insert Aj into the sorted sequence {A1, ... Aj-1}
3   ·  i ← j - 1
4   ·  while (i > 0 and Ai > key) do {
5   ·  ·  Ai+1 ← Ai
6   ·  ·  i ← i - 1
7   ·  }
8   ·  Ai+1 ← key
9   }

```

What does this do? It's a sorting algorithm⁸ but how does it work?

An alternative way of representing the same algorithm is here:

Algorithm 4: InsertionSort (*A*)

```

1   for (j = 2 up to length(A)) do {
2   ·  key ← A[j]
3   ·  i ← j - 1
4   ·  while (i > 0 and A[i] > key) do {
5   ·  ·  A[i + 1] ← A[i]
6   ·  ·  i ← i - 1
7   ·  }
8   ·  A[i + 1] ← key
9   }

```

Another algorithm in pseudocode

Algorithm 5: odds (*n*)

```

1   if (n ≤ 1) then {
2   ·  if (n < 0) then {
3   ·  ·  return some kind of error message
4   ·  }
5   ·  print 1
6   }
7   for (i = 1, ..., n) do {
8   ·  print (2i - 1)
9   }

```

Notes

You might not *program* it like this, but you should be able to convert what's above into a program that works without too much difficulty, once you get a bit better with Java.

Recall that this algorithm isn't in any particular programming language: it's your job to convert it.

naming tHiNgS

By convention,

- Class And Interface Names Begin With An Uppercase Letter
- variable names begin with lowercase letters
- CONSTANTS TEND TO BE ALL IN CAPITALS! LIKE YELLING!

⁸adapted from Cormen *et al.*'s wonderful book, Introduction to Algorithms

5 Variables

Variables

Programs would be pretty simple and useless if they couldn't handle data that could take different values. What if we wanted to print something other than "Hello World!" or "I'm sorry Dave, I can't let you do that"?

To do this we need to have some *variables*. The simplest ones are stored as *primitive types* in Java (and other OO languages): they correspond basically to numbers and characters.

In Java⁹ there are more complex types of variable, called *objects*. Objects can not only have their own data, but also their own functionality. You might think of them as "smart variables".

5.1 Declaring Variables

Declaring Variables

In Java and many other languages we must *declare* variables in advance, before we start using them. This makes the job of compiling code much easier, and it's a reasonably good way of thinking about your code too: you have to think about what you'll use before using it.

It's the difference between this:

Algorithm 6: HammerItIn (*nail*)

- 1 **let** *H* **be** a hammer
- 2 Apply *H* to the *nail*

and this:

Algorithm 7: HammerItIn (*nail*)

- 1 hammer *nail* in with *H* // which requires you to work out what *H* is

So you have to *declare* a variable before you use it: you say what kind of thing it is, like an integer, character or string, before you start using it. Code that misses the initialisation out or gets it in the wrong order...

```

1 public class OutOfOrder {
2     public foo() {
3         x = 5;
4         int x;
5     }
6 }
```

```

1 public class Undeclared {
2     public int foo() {
3         x = 3;
4         return x;
5     }
6 }
```

simply won't compile:

```

> javac Undeclared.java
Undeclared.java:3: cannot find symbol
symbol  : variable x
location: class Undeclared
    x = 3;
    ~
Undeclared.java:4: cannot find symbol
symbol  : variable x
location: class Undeclared
    return x;
    ~
2 errors
```

⁹and other OO languages

5.2 Initialising Variables

Initialising Variables

Once they're declared, variables have to be initialised before they're used.

In many languages (including C, C++) primitive variables automatically get a default value on declaration. Usually it's 0 or something equivalent.

In Java this won't help you because the Java compiler will insist that all variables be initialised before they're used.

This is very good practice anyway! 😊

Let's see what happens if we try to make some shortcut and not bother to initialise a variable:

```
1 public class Uninitialised {
2     public static void main(String [] args) {
3         int x;
4         System.out.println(x);
5     }
6 }
```

```
> javac Uninitialised.java
Uninitialised.java:4: variable x might not have been initialized
    System.out.println(x);
                       ~
1 error
```

Right. Get the idea? The Java compiler will actively prevent you from taking the shortcut of relying on default values.

In this next listing, the compiler also alerts us about the `str` variable, but the `msg` variable, which is explicitly initialised, is fine.

```
1 public class Uninitialised {
2     public static void main(String [] args) {
3         int x;
4         System.out.println(x); // x not yet initialised; not permitted
5         String str;
6         System.out.println(str); // str not yet initialised; not permitted
7         String msg = null;
8         System.out.println(msg); // msg initialised: all fine
9     }
10 }
```

5.3 Boolean

Simple variables: boolean

A Boolean variable, in Java written as “boolean” and in C/C++ as “bool”, is one that can take a value *true* or *false*.

You will come across Boolean variables all over the place, and you will mainly use them to test whether certain things are true or not, like this:

In pseudocode:

Algorithm 8: BoolTest

```

1   Boolean : t           // declare the variable
2   t ← TRUE             // initialize it
3   if (t) then {
4     · print "Yes! It's true!"
5   }

```

and in Java...

```

public void BoolTest() {
    boolean t = true; // we declare and initialize in one line here
    if (t) {
        System.out.println("Yes! It's true!");
    }
}

```

Simple variables: Numbers

There are several ways of storing numbers, but they divide into two broad categories: *integers* and *floating-point numbers*. Real numbers are stored as floating point numbers; whole numbers are integers.

We'll talk about binary numbers, integers, and floating-point numbers on the next few slides.

Binary Numbers

It helps to know about binary numbers in your programming, because everything in a computer is stored in binary digits (bits).

You can think of Boolean¹⁰ values as *binary* numbers (in a sense)

$$TRUE = 1, FALSE = 0$$

Integers are stored as binary

All integers can be thought of as binary numbers: 0, 1, 10, 11, 100, 101, ...:

base 10	base 2 / binary	expansion
0	0	0
1	1	1×2^0
2	10	$1 \times 2^1 + 1 \times 2^0$
3	11	$2^1 + 2^0$
5	101	$2^2 + 2^0$

In binary you can count to 32 on one hand.

¹⁰Why have I written "Boolean" here and not boolean? Because a Boolean variable is named after George Boole, who developed the concept of Boolean algebra in 1854. The variable type in Java uses the lower-case 'b' because it's a *primitive type*, and the naming convention for primitives is to use a lower case.

Variables have different types

When you write mathematical expressions like this,

$$d = 3x^2/7$$

and you set the value of x to 2, 1.1, -17, π , whatever, you expect to get the same kind of answer. You don't care what kind of number x or d is.

In a computer though, numbers can be stored in different ways: if x is stored as an integer with `int` or `short`, then saying " $x = 1.1$ " is just crazy.

In this example, x can only take values like 10, 71273917, 2, -17, 4 etc.; not 1.1 or π .

What happens when you run this code (in no particular language)?

```
1  int x = 1.1; // this is an assignment
2  print(x);
```

It will probably print "1", not 1.1, because x is an integer.

Printing ints

What if you put this:?

```
1  int x = -0.5;
2  print(x);
```

In fact in Java, the compiler will complain that you're setting an `int` to have a double value: let's work around it with the following hack:

```
1  public class Ints {
2      public static void main(String [] args) {
3          int x = -1/2;
4          System.out.println(x);
5      }
6  }
```

It prints zero!

```
> javac Ints.java
> java Ints
0
```

Integer division

What's going on?

This turns out to be a little subtle: because x has to be an `int`, the Java program at runtime has to ensure that the value on the right is an `int`, in this case by *truncating* the value.

But it's worse than that: because both "1" and "2" are most naturally interpreted as `ints`, the expression on the right is evaluated as though they are both integers.

In integer division, the remainder is *discarded*.

5.4 Assignment

Assignment

I snuck in something new there: nothing too terrifying, just an *assignment*¹¹.

In Java and C and its varieties and many many languages, if we want to set the value of something we use *assignment* that looks like this:

¹¹This is one assignment that's incredibly simple and you don't have to worry about marks for it.

```
3 int x = -1/2;
```

Just remember the value on the *left* gets the value of the expression on the *right*.

In pseudocode we'd write $x \leftarrow \frac{-1}{2}$: the left-arrow is often called “gets” to mean the variable on the left *gets* the value on the right.

Assignment is not equality

Saying $x = 3$ in mathematics means “ x is a variable whose value is currently 3”.

In that sense the equivalent statement is $3 = x$, but writing that in a Java (or C or Python) program doesn't make sense: you would be attempting to change the value of 3!

If you want to test whether x has the value 3, you would evaluate the following expression:

```
1 (x == 3);
```

which has the Boolean value *true* if x really does equal 3 and the value *false* otherwise.

Don't confuse '=' (assignment) with '==' (equality comparison)!

In pseudocode we often use the original mathematical meaning of “=”, not as assignment but as a statement of equality, and we use the left-arrow “gets” symbol (\leftarrow) to mean assignment.



Comparison — warning!

A trap for the unwary is in comparing floating point numbers for equality: this is a BAD IDEA.

In your computer floating point numbers are stored to finite precision, so for instance $1/3$ is not stored exactly but must be rounded to the nearest decimal. Let's look at an example:

```
1 public class Rounding {
2     public static void main(String [] args) {
3         float f = 1.0f / 3.0f;
4         System.out.println("f = " + f);
5     }
6 }
```

prints out

```
f = 0.33333334
```

which may not seem like a big error, but it does mean that it's not *exactly* one third.

Ok, back to more about variables.

Converting floating-point to integer types

Let's pursue this a bit further to see what's going on, with the following code:

```
1 public class Ints {
2     public static void main(String [] args) {
3         int x = -99/100;
4         System.out.println(x);
5     }
6 }
```

Again, the result is 0, because if you divide -99 by 100 the answer is -0.99 — which gets truncated to 0, *not* rounded to -1 as you might have hoped.



```
> javac Ints.java
> java Ints
0
```

Now let's look at what happens to a floating point number under these conditions: we'll divide an int by an int and assign the result to a float. That should be fine, right?

```
1 public class Ints {
2     public static void main(String [] args) {
3         float x = 1/2;
4         System.out.println(x);
5     }
6 }
```

```
> javac Ints.java
> java Ints
0.0
```

Um...

if all you have are ints...

...then all you'll get are ints.

What's going on is that if you only have integer values on the right hand expression, that's all that the result will be too.

In our `float x = 1/2;` assignment, the first thing that happens is the integer division, with the result of 0 (discarding the remainder, remember?).

The next thing is that this integer value (now 0) is converted to a float, which is too late for us to get the answer we wanted.

5.5 Casting

Casting

Of course there are good ways to get around this.

One is to be very clear about what kind of variables you have at the outset, doing something like this:

```
1 public class Ints {
2     public static void main(String [] args) {
3         float x = 1.0/2.0; // both values are now double
4         System.out.println(x);
5     }
6 }
```

which asserts that the values 1 and 2 are to be treated as floating-point numbers of the type double.

We can also directly *cast* one variable as another, which means “treat this as a *blah*”:

```
1 float x = (float) 1 / (float) 2;
```

Here both 1 and 2 are to be treated as though they're floats — which means at runtime an attempt will be made to convert them into float values.

In fact you only need one of the values to be a float (or double) for it to work as you'd like:

```
1 float x = (float) 1 / 2;
```

would also be fine, because the `(float) 1 / 2` expression means treat the “1” as a float, and then as at least one of the values is a float, both of them will be treated as floats.

Note: `(float) (1 / 2)` does not mean the same thing as `(float) 1 / 2!`

Casting quiz

Pop Quiz!

Question 1: What would be the value of the variable `x` after executing these code fragments?

```
1 float x = (float) 3 / 4;
```

Question 2:

```
1 float x = (float) (3 / 4);
```

Question 3:

```
1 int x = (int) 3 / (int) 4;
```

Question 4:

```
1 float x = (float) (int) (3.0 / 4);
```

Question 5:

```
1 float x = (int) (3.0 / 4);
```

Answers

Answer 1: 0.75

Answer 2: 0.0

Answer 3: 0

Answer 4: 0.0

Answer 5: 0.0

Casting among different types

In general if you have different types of values, you can cast one type as a different type (that is, treat it as a different type), if it makes sense to do so.

- Could you treat a cat as a dog? Not well.
- Could you treat a cat as a pet? Sure, if it's one of the tame ones.
- Could you treat a dog as a pet? Yep; ditto.
- Could you treat a pet as a dog? Well, only if the pet *were* a dog. Not if it were, say, a parrot or an aardvark.

- Does it make sense to treat a float as an int? Only if the float represents an integer value, as we've seen above.
- Can you treat an int as a float? Sure! All integer values are also floating point numbers.

Syntax of casting

Here is the syntax you should use when casting one thing as another: If you have two types say A and B then you can do this:

```
1 A a = <somevalue>;  
2 B b = (B) a;
```

which tries to treat the expression immediately following "(B)" as a B-type thing. For example,

```

1 Pet fido = <myDoberman>; // I'm initialising fido as myDoberman
2 Dog odif = (Dog) fido;

```

which makes sense because I'm saying "treat fido as a Dog and call it odif". I know it makes sense because I happen to know fido is a Dog. This is quite risky though, because while *I* know it's going to work in this case, not all Pets are Dogs, and I may find later that I try to treat my cat as a Dog.



Casting examples

Let's look at another case:

```

1 Cat petal = <myTabby>; // I'm initialising petal as myTabby
2 Dog odif = (Dog) petal;

```

doesn't make sense because petal isn't a dog. petal is a Cat, which is a kind of Pet, so I can definitely treat it as a Pet, but I can't treat it as a Dog.

And,

```

1 Pet fido = <myTabby>;
2 Dog odif = (Dog) fido;

```

won't work either: saying myTabby is a Pet is fine but not all Pets are Dogs.

5.6 Scope

Variables have a lifetime

In the code you've seen so far the variables have been short-lived but this hasn't mattered one bit.

Next we'll see this is a very important concept for you to remember (yes, another one).

Blocks of code

I have alluded to blocks already, but to make sure you have the right idea about them, let's be a bit more explicit about what they are, because we'll need this next.

A *block* is a collection of statements that is delimited by a pair of curly braces { }.

```

1 if (x == 3) {
2     if (y > x) {
3         if (x * x < y) {
4             // nested blocks
5         }
6     } else {
7         // do stuff
8     }
9 }

```

In the previous code a block begins on line 1 and extends to line 9; another block begins within it on line 2 and extends to line 6, where it finishes and another block begins. The innermost block begins on line 3 and ends on line 5.

The blocks are indented for readability — which is probably much more important that you realise so far, because as has been famously said,

Any code of your own that you haven't looked at for six or more months, might as well have been written by someone else.

Eagleson

You might be worried about over-writing your variables in an inner block that you were already using in an outer block, but have no fear: the Java compiler won't let you do that:

```

1 public class Scope {
2     public static void main(String [] args) {
3         int x = 3;
4         if (x < 5) {
5             int x = 6;
6             System.out.println(x);
7         }
8     }
9 }

```

```

> javac Scope.java
Scope.java:5: x is already defined in main(java.lang.String[])
        int x = 6;
            ~
1 error

```

But you *can* perpetrate other evil:

```

1 public class ScopeEvils {
2     public static void main(String [] args) {
3         for (int i = 0; i < 10; ++i) {
4             // do stuff
5         }
6         for (int i = 3; 3 < 8; i += 2) {
7             // do different stuff
8         }
9     }
10 }

```

This *can* go horribly wrong, but it's mostly fine. Just be careful, ok?



Scope

Once it's been declared, a variable will only persist within its *scope*.

The scope of a local variable is basically the block in which it is declared:

```

1 if (x == 3) {
2     int y = 5; // y is declared here, but z doesn't exist yet
3     int z = y; // y is still fine, and z is now declared too
4     x = y;    // we'll copy the value of y into x
5 }
6 // y is no longer defined: it is out of scope
7 // x now has the value 5

```

In the listing above the variables *y* and *z* aren't yet defined at line 1: only *x* is. *x* will still be defined inside the block whose code is on lines 2 - 4 inclusive, and beyond.

Within the block beginning on line 1, *y* is declared and initialised (line 2). It's now *in scope*, so other things can use it, and it's properly initialised so it has a well defined value (in this case, 5).

On line 3, *z* is also declared, and initialised to the value of *y*. Now both *y* and *z* are in scope, along with *x*. Line 4 copies the value of *y* into the variable *x*, whose scope extends outside the block.

When we end the block on line 5, *y* and *z* are no longer defined. We can't access them and their values are gone. (In this case, we saved the value of *y* by copying it into variable *x*, which is still in scope.)

Summary:

- Variables come in different varieties or *types*;
 - the logical type is `boolean`;
 - integer types are `int`, `short`, `long`;
 - floating point types are `float` and `double`;
 - the letter type is `char`
- treating floating point variables as integer variables truncates remainders
- variables can be cast as different types
- casting to a more general type should always work
- casting to a more specific type will *not* always work

6 Operators

6.1 Definitions

Operator terminology

An *operator* is a symbol or group of symbols used as a short-cut for some more complex operation.

Operators work on *operands*. They may or may not modify the operand.

An operator can be

unary operating on one operand;

binary operating on two operands

ternary operating on three operands

Assignment operator =

There are several operators in Java to make your life easier. The most common you'll probably use is the *assignment* operator:

=

We've seen this before: use = to assign the value on the left to take the value on the right:

lvalue ← *rvalue*

lvalue = *rvalue*

And remember the equality operator is `==`: it is used to compare whether two *primitive types* are equal.

Increment operator ++

Another very commonly occurring operator is the *increment*:

++

which is short-hand for “add 1 to the operand”.

Here's how it works in code:

```
1  int x = 4;
2  x++; // add 1 to x; now x is 5
```

The `++` *increment* operator can appear either before or after its *operand*¹². Its meaning is slightly different in the two cases and you should know about this subtlety.

¹²the thing on which an operator, well, operates

6.2 Increment

6.2.1 Increment operator ++

++before and after++

If you write the ++ after the operand (in Java — and you see the same effect in C++ and others)

```
1  int x = 4;
2  int y = x++; // increment after the operand
3  System.out.println("y = " + y);
4  System.out.println("x = " + x);
```

what do you think will be printed?

Let's see:

```
> javac PlusPlus.java
> java PlusPlus
y = 4
x = 5
```

inside ++

If you put the ++ *after* the operand it means “get the value of the operand and *then* increment it” — though what really happens is more like “store the current value of the operand, increment the operand, and then return the stored value”

So our code above makes perfect sense: y gets the value 4, because that's the value of x before it increments.

If we put the ++ operator *before* the operand then the increment happens *first*, *before* the value is returned:

```
1  int x = 4;
2  int y = ++x; // increment before the operand
3  System.out.println("y = " + y);
4  System.out.println("x = " + x);
```

```
> javac PlusPlus.java
> java PlusPlus
y = 5
x = 5
```

The next operator we'll look at is the obvious companion to the ++ *increment*: it's the - *decrement* operator.

Decrement operator --

The decrement operator works the obvious way: instead of adding 1, we subtract 1.

As with ++, when the operator is *after* the operand, the value returned is that *before* the change, so

```
1  int x = 5;
2  System.out.println("x = " + x--);
3  System.out.println("x = " + x);
```

produces

```
x = 5
x = 6
```

So the basic message is, avoid cases where the *side effects* of the operator make things awkward.

Don't embed increment or decrement

This is a little confusing, and for that reason some programmers avoid the increment and decrement operands entirely as it's too easy to get things horribly wrong.

What do you think happens in the following?

```
1  int x = 4;
2  if (x-- == 3) {
3      System.out.println(x);
4  }
5  int a[] = new int[5];
6  a[--x] = x++; // argh!
```

The problems occur because the increment and decrement operators are *within expressions*. Avoid this!



Personally I think the advantage to brevity and clarity of having the increment and decrement operators available outweighs the risk of using them embedded within expressions

In general, it's highly advisable therefore to only use them in their own explicit expressions.

```
1  if (x < 0) {
2      x++;
3  }
```

is fine, but

```
1  if (x++ < 1) {
2      // have a headache
3  }
```

is not.

6.2.2 Comparing primitives

Equality operator ==

This binary operator should be very familiar by now: use == to return the value *true* when the two operands are equal:

$$lvalue == rvalue$$

is true if and only if the two values are the same.

Comparing ints and booleans

```
1  int x = 4;
2  int y = 4;
3  int z = 2;
4  boolean xySame = (x == y);
5  boolean xzSame = (x == z);
6  System.out.println("xySame = " + xySame);
```

```

7 System.out.println("xzSame = " + xzSame);
8 System.out.println("(xySame == xzSame) = " + (xySame == xzSame));

```

```

> emacs Equality.java
> javac Equality.java
> java Equality
xySame = true
xzSame = false
(xySame == xzSame) = false

```

Comparing floats and doubles

You'll recall that comparing floating point numbers for equality is a bad idea — but you do it like this:

```

1 float f = 1.0f/3.0f;
2 float g = 1.0f - 2.0f/3.0f;
3 boolean same = (f==g);
4 System.out.println("same = " + same);

```

```

> emacs FloatCompare.java
> javac FloatCompare.java
> java FloatCompare
same = false

```

Note that this code is “checking” to see whether

$$\frac{1}{3} = 1 - \frac{2}{3}$$

and apparently it isn't. See what a bad idea this is?

6.2.3 Comparing Strings

Comparing Strings: `str2 == str2` ?

When you're comparing Strings, as I'm sure you are already (or will be soon) you mustn't use the `==` operator. That operator compares the *addresses* of the String objects, not their *value*.

Use the equals method that is defined on the String class:

```

1 String s = "hello";
2 String s2 = "hello";
3 if (s.equals(s2)) {
4     // ...
5 }

```

More on comparing Strings

Let's see what happens in practice:

```

1 public class Compare {
2     public static void main(String [] args) {
3         String s = "hello";
4         String s2 = "hello";
5         System.out.println("s = \" + s + "\"");

```



```

6     System.out.println("s2 = \"" + s2 + "\"");
7     if (s == s2) {
8         System.out.println("s and s2 are equal");
9     } else {
10        System.out.println("s and s2 are not equal");
11    }
12 }
13 }

```

Oh, did you see the use of the escape character there? I needed to put a double-quote " inside a string to print out, but the double-quote is used to *delimit* a string in the first place.

So we have to *escape* the normal usage of the " to actually *print* a double quote.

```

5     System.out.println("s = \"" + s + "\"");

```

Ok back to comparing Strings...

Running the Compare.java code we get

```

> java Compare
s = "hello"
s2 = "hello"
s and s2 are equal

```

Oh, um, what's going on here? It works!

In fact it doesn't always work. What's going on is that the Java Virtual Machine (JVM) where the program is run, recognizes that the two Strings are *static*: they're not changing, they're *constant*.

Because they're constant Strings, they're stored as the *same object*. So in *some* cases, comparing Strings with the == operator *does* work.

What if they are not constant Strings?

Let's put this into practice:

```

1 public class CompareAgain {
2     public static void main(String [] args) {
3         String s = "hello";
4         String s3 = "hello world";
5         String s4 = s3.substring(0,5);
6         System.out.println("s = \"" + s + "\"");
7         System.out.println("s3 = \"" + s3 + "\"");
8         System.out.println("s4 = \"" + s4 + "\"");
9         if (s == s4) {
10            System.out.println("s and s4 are equal with ==");
11        } else {
12            System.out.println("s and s4 are not equal with ==");
13        }
14    }
15 }

```

```
> javac CompareAgain.java
> java CompareAgain
s = "hello"
s3 = "hello world"
s4 = "hello"
s and s4 are not equal with ==
```

Ahah!



equalsIgnoreCase

When you don't compare about the case (upper or lower) of letters in the string, use `equalsIgnoreCase`.

```
1 String s = "hello";
2 String s2 = "HeLlO";
3 if (s.equalsIgnoreCase(s2)) {
4     // ...
5 }
```

6.3 First mathematical operators

Operators +, -, *, /

The next set of operators are very straightforward: they are the standard *binary operators* of mathematics:

+	-	×	÷
+	-	*	/

They do just what you expect them to do. Nothing to see here. Move along.

Operators +=, -=, *= and /=

These operators are a very nice shorthand. They all operate in the same way, by modifying the operand on the left, using the operand on the right.

shorthand	equivalent to
<code>x += n;</code>	<code>x = x + n;</code>
<code>x -= n;</code>	<code>x = x - n;</code>
<code>x *= k;</code>	<code>x = x*k;</code>
<code>x /= k;</code>	<code>x = x/k;</code>

In general,

$$x \square= y$$

is equivalent to

$$x = x \square y,$$

for whatever \square is.

Note that this isn't always exactly the same but you have to go out of your way to find cases where the results differ.

Shorthand Quiz

Given `int x = 12` and `int n = 2` and `int k = 3`, what is the resulting value of `x` in each of these expressions? (Suppose the values are reset for each case.)

Pop Quiz!

Question 1: `x++`;

Question 2: `x /= k`;

Question 3: `x *= n`;

Question 4: `x += (++n)`;

Question 5: `x += (n *= k)`;

Question 6: `x += n++`;

not!

In Java the negation operator looks like this: “!” — it’s the exclamation mark. In code you’ll see this quite often, for example in expressions like

```
if (x != y),
```

which is true if `x` is *not* equal to `y`, or like this:

```
if (!(x == y)),
```

which is true if the statement “`x == y`” is *false*.

In mathematical symbols we use the “negate” symbol \neg for “not”.

- The value of $\neg x_1$ is true if, and only if, x_1 is *false*.

Just to be clear, this not-equals operator isn’t like the ones you just saw, `+=`, `*=` and so on. It doesn’t modify the operand on the left by applying the operand on the left: it’s a simple comparison. Java apologises for this confusion. I apologise if I’ve made it worse.

6.4 Boolean Logic

Throughout your programming life you’ll use Boolean logic constantly. There are some simple rules you will need, which you probably already know so let’s just have a quick refresher:

And and Or

Let’s think about a set of variables, called x_1, \dots, x_k .

- The value of $(x_1 \text{ AND } x_2)$ is true if, and only if, both x_1 and x_2 are true.
- The value of $(x_1 \text{ AND } x_2 \text{ AND } \dots \text{ AND } x_k)$ is true if, and only if, all of the x_i are true.
- The value of $(x_1 \text{ OR } x_2)$ is true if, and only if, at least one of x_1 and x_2 is true.
- The value of $(x_1 \text{ OR } x_2 \text{ OR } \dots \text{ OR } x_k)$ is true if, and only if, at least one of the x_i is true.

In Java we write `&&` for logical AND, and `||` for logical OR.

iff

You know what? I'm getting tired of writing "if and only if" when there's a perfectly good shorthand for this expression:

iff

Other notation for AND and OR

In more advanced Computer Science and in Mathematics you'll also see this symbol for a logical AND:

\wedge as in $x_1 \wedge x_2$

also called *meet* or *conjunction*, and this one for a logical OR:

\vee as in $x_1 \vee x_2$

also called *join* or *disjunction*.

Logical pop quiz

Given variables $x = \text{true}$, $y = \text{true}$, $z = \text{false}$, what is the value of the following expressions?

Pop Quiz!

Question 1: x

Question 2: $\neg x$ $!(x)$

Question 3: $(x \text{ AND } y)$ $(x \ \&\& \ y)$

Question 4: $(x \text{ OR } \neg(y \text{ AND } z))$ $(x \ || \ !(y \ \&\& \ z))$

Question 5: $(x \text{ OR } y) \text{ AND } (\neg y \text{ OR } z)$ $(x \ || \ y) \ \&\& \ (!y \ || \ z)$

Can $x \wedge \neg x$ be satisfied ever?

Operator Precedence

Just don't try to remember them all.

Use parentheses.

7 Arrays and Iteration

7.1 Arrays

Storing a bunch of stuff

If you're anything like me you like to keep things approximately organised (the condition of my office should not be used as an indicator).

In your house, do you keep all your stuff in one big pile? No of course not: it makes much better sense to have all the things of the same type (e.g., books, CDs, clothes) in the same place. Books line up neatly in rows on a shelf; CDs have their own specially designed racks or shelves, and clothes go in drawers and on hangers.

It wouldn't make sense to have one shelf for each book, would it?

Arrays

In a computer the best way to store things is also in an organized way, and one of the simplest ways you can get for storing things of the same type is in an *array*.

Arrays are cool.¹³

Java is an OO language, and it has a special class called the `ArrayList` that stores its contents in an array.

In the next part we'll learn a little about how it works, at least in principle.

An array in programming is a *contiguous block of memory* containing things of the same type.

This enables you to access a collection of things very quickly: in fact, in a constant amount of time (effectively independent of the size of the array), if you know where they are in the array.

Arrays — creation

Here's how you create some arrays:

```

1  int [] x; // declare an array of integers, of unknown length
2  int y = 15;
3  double [] z = new double[y];
4          // declare and initialise an array of doubles
5  String [] myStrings = new String[y];
6          // declare and initialise an array of Strings

```

Line by line,

line 1: *declares* an integer array variable, but does not initialize it: it's `null` so far.

line 2: declares and initializes the integer `y` with value 15.

line 3: declares and initializes the array `z` with `y` doubles (all initially 0)

line 4: declares and initializes the array `myStrings` with an array of `Strings`, all initially `null`.

Arrays — Initialisation

You can either initialise arrays as we did above, explicitly providing the size of (number of elements in) the array, or you can use this handy method in Java:

```

1  public class InitArrays {
2      public static void main(String [] args) {
3          String names [] = new String [] { "Bill", "Ted", "Larry" };
4          System.out.println(names[0]);
5          System.out.println(names[1]);
6          System.out.println(names[2]);
7      }
8  }

```

which creates an array of `String` objects, the contents of which are the three strings “Bill”, “Ted” and “Larry”. The length of the array is set *implicitly* to the number of elements in the strings provided in braces.

You may have spotted an inconsistency in the way the arrays are initialised: sometimes the brackets come before the variable name, like this:

```

1  int [] x;

```

and sometimes after:

```

1  int y [];

```

¹³Like bow ties.

Well in fact it doesn't matter, and you can mess with the whitespace as well if you're bored:

```
1 int x[   ];
```

About the only things you might try that you can't do are declaring the size of the array, like this:

```
1 int z [14];
```

or something weird like this:

```
1 int [ w ];
```

which is just asking for trouble.

Probably the best way of writing it is like this, with the [] before the variable name:

item [] *var*

for an array of *items*, called *var*. If you think about it logically, that makes the most sense, because it interprets easily as “make an array of *items*, called *var*”: that is, the *type* of *var* is *array-of-item*. The reason that `int z[14];` can't work is that the type of the variable *z* should be “array of ints,” not “array of 14 ints”.

This leads neatly into the idea of declaring multiple variables, which we've not yet looked at. If you're writing

```
1 int x;
2 int y;
3 int z;
4 int w;
5 int a;
6 int b;
```

you may get a little concerned that this is taking up valuable space, or is wordy, or is boring, or... Well aside from the idea that this is in fact very clear, and enables you to put a comment next to each variable in a consistent way:

```
1 int x; // the x-coordinate
2 int y; // the y-coordinate
3 int z;
4 int w; // the weight
5 int a; // the first foo
6 int b; // the second bar
```

You can, if you have such concerns, compact the declaration into a much smaller space. **But don't do it if it just obfuscates your code!**¹⁴ Here's how:

```
1 int x, y, z, w, a, b;
```

Nice.

In particular this makes sense if you use the standard method of declaring an array — this:

```
1 int [] x, y;
```

declares two variables called *x* and *y*, both arrays of ints. So does this:

```
1 int [] x, y [];
```

but it's awful. So if you can, stick to the standard way, which has the [] on the left of the variables, next to the *item* type.

¹⁴Once you've looked up “obfuscate” you'll get my point even better.



Arrays — index

Did you see how we accessed the elements in the array? The numbering started at 0, not 1. This is often called “*the zero-th element*” or “0th element” of the array.

Arrays are numbered from 0 up to (size of the array) - 1.

To access the i th element of an array A we just use $A[i]$.

If i is less than 0 then this isn't defined and, trying it, we'd get a run-time error.¹⁵

If i is too big we also get an error. Because the array elements are numbered from 0, “too big” means “equal to or greater than the size of the array”.

The reason for this is that where the array is stored in memory is a value that's itself stored somewhere. Let's just call it “ a ” for the moment.

Then the address of the 0th element in the array is a . Because arrays are *contiguous* that means there may be more elements of the same type in the address locations immediately after the first one in the array. If all the items in the array are the same size (and they will be, because they're the same type, remember?) then all we need to do to get to the i th element is multiply i by that size and add it to a .

For example, say my array is stored at address location 200 in my computer, and I'm storing numbers in my array that each take up 4 bytes of space. Then the next element in the array after the first will be at address $200+4=204$, and the one following it will be at address 208, etc. These calculations are really quick: it is such a simple task that it doesn't matter how big the array is and you can quickly work out where the i th element is by simple multiplication and addition.

Pop Quiz

Write your answers down and bring them to your next lab.

Pop Quiz!

Question 1: Why is it very quick to access items in an array?

Question 2: How can we *find* the i th item in an array?

Question 3: Why should we only store items that are the same size in an array?

Question 4: How can we insert something into an array?

Arrays: example

We'll look at an example of how you might use an array in the following:

```

1 public class FindMax {
2     public int max(int [] nums) {
3         // ... we will fill in this part.
4         return 0;
5         /*
6          * In order for the compiler to be happy so far we must return
7          * some value here. We'll fix this!
8          */
9     }
10    public static void main(String [] args) {
11        FindMax fm = new FindMax();
12        int nums[] = new int[] { 23, -1, 5, 78, 22, 0 };
13        System.out.println(fm.max(nums));

```

¹⁵Write a program to test this behaviour: see what happens.

```

14 }
15 }

```

7.2 Iteration through an array

Finding the max

Ok before we begin to put in code we should know what we're going to put. This is a basic concept really: plan *first!*

Algorithm 9: FindMax (A)

```

1  let  $A$  be an array of numbers (at least one!)
2  let  $m$  be the result we will return
3   $m \leftarrow A[0]$  //  $m$  will be the first element of  $A$ 
4  for each ( $A[j]$ ) do { // this is a bit vague
5  · if ( $m > A[j]$ ) then {
6  · ·  $m \leftarrow A[j]$ 
7  · }
8  }
9  return( $m$ )

```

FindMax

Ok let's turn it into code.

Here's some I prepared earlier:

```

1  public class FindMax {
2      public int max(int [] nums) {
3          int m = nums[0];
4          for (int j = 1; j < nums.length; ++j) {
5              if (m < nums[j]) {
6                  m = nums[j];
7              }
8          }
9          return m;
10     }
11     public static void main(String [] args) {
12         FindMax fm = new FindMax();
13         int nums[] = new int[] { 23, -1, 5, 78, 22, 0 };
14         System.out.println(fm.max(nums));
15     }
16 }

```

for (..;..;..)

In the previous code there's a *for loop*. It is *iterating* through the array. Iteration is simply repetition of a process. In this case it's moving through the items in an array, but we will see more examples of iteration.

The *for* loop is one of the most common control flow structures you'll see in your programming life.

It's not the simplest one, but it's really common so let's look at it in some detail so we can see what the previous code is doing; then we'll get into more depth.

Iterating through an with for

As I mentioned the for loop looks like this:

for (initialisation ; condition ; update) expression

By far the commonest situation you'll see and use this is as something like this:

```

1  for (int i = 0; i < 10; i++) {
2      System.out.println("i = " + i);
3  }
```

for “for”

initialisation “int i = 0”

condition “i < 10”

update “i++”

expression “{ System.out.println(“i = ” + i); }”

FindMax revisited

```

1  public class FindMax {
2      public int max(int [] nums) { // array of ints
3          int m = nums[0]; // first element
4          // ( initialisation; condition; update )
5          for (int j = 1; j < nums.length; ++j) {
6              // statement in a block:
7              if (m < nums[j]) { // if m is less than the j-th element
8                  m = nums[j]; // then store this new largest value
9              }
10             // the block stops here
11         }
12         return m;
13     }
14 }
```

Running FindMax

```

> javac FindMax.java
> java FindMax
78
```

This would be pretty useless if we wanted to find the maximum of any *other* numbers than 23, -1, 5, 78, 22, and 0, so let's see if we can make the program deal with *arbitrary* integers, that we can enter when we invoke the program.

We'd like to be able to run it with

```

> java FindMaxFlex 34 1 6 1234 234 6 234 12 33 5 51 23 12
```

and the program return the correct answer (in this case 1234). How to do that? We'll use the `String [] args` argument that is received by the main method:

a more flexible FindMax

```
1 public class FindMaxFlex {
2     public int max(int [] nums) {
3         int m = nums[0];
4         for (int j = 1; j < nums.length; ++j) {
5             if (m < nums[j]) {
6                 m = nums[j];
7             }
8         }
9         return m;
10    }
11    public static void main(String [] args) {
12        FindMax fm = new FindMax();
13        int nums[] = new int[args.length];
14        for (int i = 0; i < args.length; ++i) {
15            nums[i] = Integer.parseInt(args[i]);
16        }
17        System.out.println(fm.max(nums));
18    }
19 }
```

Now when we run it on the command-line we can provide any collection of integers, like so:

```
> javac FindMaxFlex.java
> java FindMaxFlex 3 6 1 3 6 2 234 23 6 23 12 3 123
234
```

Size of the Java array

There is a very nice feature of the Java array that is built-in: for any array x that you define, the value $x.length$ is the length of the array.

7.3 Multi-dimensional arrays**Multi-dimensions**

Defining multi-dimensional arrays is also easy:

```
1 int [][] grid = new int[5][10];
```

creates a new 5×10 array of ints (all initially zero).

We access them in the same way as for 1-dimensional arrays:

```
1 grid[3][4] = 4;
2 grid[0][1] = 1;
```

etc.

```

1 public class TwoDArray {
2     public int[][] grid; // there will be an array of ints
3     public TwoDArray() {
4         grid = new int[5][10]; // make a 5x10 array
5     }
6     public void printAll(PrintStream ps) { // print to ps
7         for (int i = 0; i < 5; i++) {
8             for (int j = 0; j < 10; j++) {
9                 ps.print(grid[i][j] + " "); // the (i,j)-th int
10            }
11            ps.println(); // finish the line
12        }
13    }
14    public static void main(String [] args) {
15        TwoDArray t = new TwoDArray(); // make my 2D array
16        t.printAll(System.out); // call the method on the array
17    }
18 }

```

Oops! This won't compile. What's wrong?

```

~> java TwoDArray.java
Exception in thread "main" java.lang.NoClassDefFoundError: TwoDArray/java
Caused by: java.lang.ClassNotFoundException: TwoDArray.java
    at java.net.URLClassLoader$1.run(URLClassLoader.java:202)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:190)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:248)
~> javac TwoDArray.java
TwoDArray.java:6: cannot find symbol
symbol : class PrintStream
location: class TwoDArray
    public void printAll(PrintStream ps) {
                          ^
1 error

```

We need to tell the JVM¹⁶ what a `PrintStream` is. You won't know yet which things to import into the program but a good IDE will tell you.

We'll insert this line at the beginning:

```

1 import java.io.PrintStream;

```

which tells the compiler to load the `PrintStream` definition.

We should be good to go now.

¹⁶Java Virtual Machine: this runs the Java programs, remember?

```
1 import java.io.PrintStream;
2
3 public class TwoDArray {
4     public int[][] grid;
5     public TwoDArray() {
6         grid = new int[5][10];
7     }
8     public void printAll(PrintStream ps) {
9         for (int i = 0; i < 5; i++) {
10            for (int j = 0; j < 10; j++) {
11                ps.print(grid[i][j] + " ");
12            }
13            ps.println();
14        }
15    }
16    public static void main(String [] args) {
17        TwoDArray t = new TwoDArray();
18        t.printAll(System.out);
19    }
20 }
```

Running TwoDArray

```
~> javac TwoDArray.java
~> java TwoDArray
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
~>
```

And just to finish things off nicely we'll put in some values:

```
1 import java.io.PrintStream;
2
3 public class TwoDArrayFilled {
4     public int[][] grid; // there will be an array of ints
5     public TwoDArrayFilled() {
6         grid = new int[5][10]; // make a 5x10 array
7     }
8     public void printAll(PrintStream ps) { // print to ps
9         for (int i = 0; i < 5; i++) {
10            for (int j = 0; j < 10; j++) {
11                ps.print(grid[i][j] + " "); // the (i,j)-th int
12            }
13            ps.println(); // finish the line
14        }
15    }
16    public void setValue(int i, int j, int value) {
17        grid[i][j] = value;
18    }
19 }
```

```
19 public static void main(String [] args) {
20     TwoDArrayFilled t = new TwoDArrayFilled(); // make my 2D array
21     for (int i = 0; i < 5; i++) {
22         for (int j = 0; j < 10; j++) {
23             t.setValue(i, j, i*j); // times tables!
24         }
25     }
26     t.printAll(System.out); // call the method on the array
27     System.out.println(t.grid.length);
28     System.out.println(t.grid[2].length);
29 }
30 }
```

Running TwoDArrayFilled

```
~> javac TwoDArrayFilled.java
~> java TwoDArrayFilled
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
~>
```

Size of a multi-dimensional array

There's an obvious question that you might be wondering about now: what is the value of `grid.length` in the above?

Let's find out. With these lines inserted into our program...

```
1 System.out.println(t.grid.length);
2 System.out.println(t.grid[2].length);
```

we get

```
~> java TwoDArrayFilled
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
5
10
```

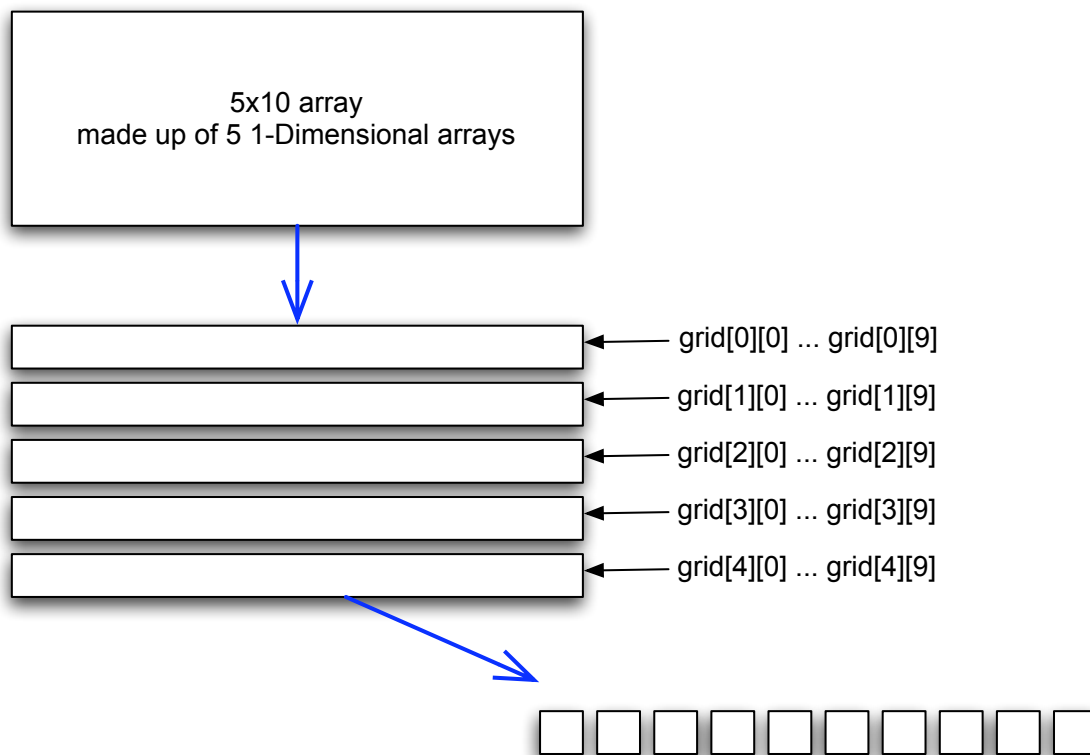
Arrays of arrays

To understand what's going on you need to know an important fact: multidimensional arrays are stored as arrays of arrays.



An array of k dimensions is stored as a 1-dimensional array of arrays of $(k - 1)$ dimensions each.

So in the listing above, `t.grid.length` is the length of the first dimension: it's the number of 1-dimensional arrays `grid[0][0] ... grid[0][9]`, `grid[1][0] ... grid[1][9]`, etc.



Pop Quiz

Are you getting sick of these yet?

Pop Quiz!

After this code executes:

```

1      int[] a = new int[15];
2      int[] b = a;
3      int[][] c = new int[5][];
4      c[2] = new int[19];
5      c[3] = new int[] { 4, 3, 6, 1 };
6      a[4] = 3;
7      a[7] = a[4];
8      a[4] = 0;

```

Question 1: what is `b.length`?

Question 2: what is `a[7]` ?

Question 3: what is `c[2][3]`?

Question 4: what is `c.length`?

Question 5: what is `c[2].length`?

Question 6: what is `c[3][4]`?

Question 7: what is `c[3][2]`?

```

~> javac ArraysQuiz.java
~> java ArraysQuiz
b.length = 15
a[7] = 3
c[2][3] = 0
c.length = 5
c[2].length = 19
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at ArraysQuiz.main(ArraysQuiz.java:18)
~>

```

There's an *exception thrown* when the JVM tries to access `c[3][4]`, as the correct range of indices for `c[3]` is `c[3][0] . . . c[3][3]`. Commenting that line out we get

```

~> java ArraysQuiz
b.length = 15
a[7] = 3
c[2][3] = 0
c.length = 5
c[2].length = 19
c[3][2] = 6

```

```
c = new int[5][];
```

all arrays `c[i]` are null!

```

null
null
null
null
null

```

```
c[2] = new int[19];
```

`c[2]` has 19 elements, all 0

```
c[3] = new int[] { 4, 3, 6, 1 };
```

`c[2]` has 19 elements, all 0
4 3 6 1

If you've not already started Task 1...

- Go to the e-learning (Blackboard) site and select the Assessments tab. Tasks are in there, and the skeleton code for Task 1 is there, called "SimpleCalc.java".
- Fill in the methods as described in the comments.
- Testing will be by running your program with command-line arguments, like this:

```
> javac SimpleCalc.java
> java SimpleCalc 2 3 4 5
```

and your program should print out the number of arguments, the maximum value, the mean (average) value, and the maximum difference between any two input numbers, like this:

```
4
5
3.5
3
```

Marking Task 1

Total marks: 2

1 automatic marking using Unit Tests

0.5 answering questions about your code in the lab

0.5 hand-marking of your code

Task 1 Automatic Marking

- Your program will be given 10 simple tests that it should pass based on different input.
- You may assume that all the input will be correctly formatted: no strings like "eleven" will be used!
- The auto-mark will be the number of tests passed / 10.
- You can test in lots of ways: a simple way is just to work out some correct cases and make sure your program prints out what it should, e.g.:

```
> java SimpleCalc 2
1
2
2
0
> java SimpleCalc 1 10
2
10
5.5
9
```

Task 1 Questions

- You must answer questions about your code, asked by your tutor in the lab.
- For full marks you should be able to explain any part of the code clearly.
- If you can't explain it then you can't get the mark for it.

Task 1 Hand-marking

Your code will be marked by the tutors according to the following scheme:

- 0.3 for good code quality, algorithm efficiency and logic;
- 0.2 for clarity and maintainability

Efficiency means you can't get full marks if your code goes through the list of numbers more than *necessary*.

A few words on clarity

Some general points:

- clarity does not mean spaced out
- use correct indentation: if in doubt, select all the code and press Control-I in Eclipse
- be consistent in your style
- comments should clarify, not make things worse

```
int x = 1; // setting x to 1, this comment is silly!  
int y = 10/x;  
/*  
 * I can put my huge descriptions in here between the  
 * slash-asterisk and asterisk-slash symbols  
 */
```

Task 1 Submission

- You must submit your Task 1 by Friday 5pm. Go to the Assignments tab in the info1103 e-learning site and submit your *zipped* SimpleCalc.java file there.
- You should have it finished in your lab so you can explain it: you can't get marks for understanding unfinished code!
- If you miss the deadline then you will have to e-mail your assignment to your tutor.
- If you miss by less than 24 hours you can get at most 1 mark.
- If you miss by more than 24 hours you can't get any marks for this Task.

8 Control Flow

(AN UNMATCHED LEFT PARENTHESIS
CREATES AN UNRESOLVED TENSION
THAT WILL STAY WITH YOU ALL DAY.)

Now you have some pretty good ideas of how to make programs do interesting things, like calculations on numbers and Strings, making arrays, and even using the “for” loop.

Next we'll go into several more control structures, which you should definitely try out and play with, so you'll get to grips with their usage quickly. Each has its advantages and disadvantages.

8.1 if

The “if” statement is explained in §5.1 in Horstmann.

Choosing among different options

There are two methods we’ll look at to choose among different options.

These are the `if` and the `switch` statements.

The if statement

- What’s it for?
 - to have different behaviour in a program based on whether something is true or false (this is a kind of *control statement*).
- What does it look like? “if (*condition*) *statement*”
- In code this looks like

```
1  if (x > 0) {
2      print("oh no! x is positive!\n");
3  }
```

Note the `if`, which says you’re going to use an “if” statement, the condition “`x > 0`” which won’t be true if say `x` is `-3` or `0`, and the statement that will be executed if `x > 0` is true.

These are the three ingredients in the “if” statement.

How if works

The “if” statement is probably the simplest control flow structure: it is used to test the value of an expression, to see whether it’s true, and if true, then to do something else.

Algorithm 10: ifdemo

```
1  if (it’s raining) then {
2      · I should remember my umbrella
3  }
4  if (x = 3) then {
5      · y ← x
6  }
```

Nesting logic with if

You can put logical expressions inside other ones, as we found with ANDs and ORs before:

“If (I’m not busy AND (I have some money OR I have a generous friend)) then I can go out”

Similarly in an `if` statement,

```
1  boolean x = true;
2  boolean y = false;
3  int n = 15;
4  if (x && n < 20) {
5      System.out.println("yay!");
6  }
7  if ((y || x) && (y && n > 15)) {
8      System.out.println("boo!");
9  }
```

Nested if

You can also construct as much complexity as you like by putting one *if* within another, like this:

Algorithm 11: If if_1 **if** (*this* AND \neg *that*) **then** { 2 · Do Thing 1 3 } **else if** (*this* AND *that*)
{ 4 · Do Thing 2 5 } **else** { 6 · Do Thing 3 7 } But this would be more *logical* as

1 **if** (*this*) **then** { 2 · **if** (*that*) **then** { 3 · · Do Thing 1 4 · } **else** { 5 · · Do
Thing 2 6 · } 7 } **else** { 8 · Do Thing 3 9 }

Extending if

if you had to write an “if” for each possible outcome you’d get bored quickly:

```

1   if (x > 0) {
2       // do something
3   }
4   if (x <= 0) {
5       // do something else
6   }
```

and we don’t want you to get bored.

(More importantly, a major limitation in software is not how quickly it *runs* but how quickly it can be *developed*— so we should make development as fast and painless as possible.)

8.2 if ... else

if ... else

The alternative to writing both cases out explicitly is to use *else*.

Here’s the syntax:

```

1   if (condition) then {
2       · statement
3   } else {
4       · alternative
5   }
```

which is short-hand for

```

1   if (condition) then {
2       · statement
3   }
4   if ( $\neg$ condition) then { // remember  $\neg$  means “not” or “the negation of”
5       · alternative
6   }
```

if ... else ... if

You will also often see several *if* statements strung together with *elses*, like this:

```

1   if (ch == 'a') {
2       // do thing 1
3   } else if (ch == 'b') {
4       // do thing 2
5   } else if (ch == 'c') {
6       // do thing 3
7   }
```

which can be hard to understand. It is equivalent to putting the expressions after the `elses` into their own separate blocks.

`if ... else ... if`

```
1  if (ch == 'a') {
2    // do thing 1
3  } else if (ch == 'b') {
4    // do thing 2
5  } else if (ch == 'c') {
6    // do thing 3
7  }
```

```
1  if (ch == 'a') {
2    // do thing 1
3  } else {
4    if (ch == 'b') {
5      // do thing 2
6    } else {
7      if (ch == 'c') {
8        // do thing 3
9      }
10   }
11 }
```

if in practice

Let's look at a more practical case of using `if`:

```
1  public static void main(String [] args) {
2    if (args.length > 0) {
3      System.out.println("You entered " + args.length + " strings\n."
4        + "The first is \"" + args[0] + "\".");
5    } else {
6      System.out.println("You didn't enter anything.");
7    }
8  }
```

What happens here: a simple test on line 2 checks to see if the length of the array is greater than 0, that is, if there are any elements in the input array `args`.

another if/else

```
1  import java.io.PrintStream;
2  public class IfElseElse {
3    public static void main(String[] args) {
4      if (args.length == 0) {
5        return;
6      }
7      PrintStream ps = System.out;
8      String str = args[0];
9      char ch = str.charAt(0);
10     if (ch == 'a' || ch == 'A') {
11       ps.println("The first letter is A");
12     } else if (ch == 'b' || ch == 'B') {
13       ps.println("The first letter is B");
14     } else {
15       ps.println("There has to be an easier way.");
16     }
17   }
18 }
```

Running IfElseElse

```

~> javac IfElseElse.java
~> java IfElseElse
~> java IfElseElse alfred
The first letter is A
~> java IfElseElse jimmy
There has to be an easier way.
~>

```

Writing `if...else if...else if...else` can be very wordy and time-consuming. Luckily there's a short-cut when you have many alternatives: the *switch*.

8.3 switch

Now we'll look at the second method of choosing among different options, the `switch` statement

`switch`

Syntax:

```

switch ( testvalue ) {      case1 value1 : statement1 [ break; ]      case2 value2 : statement2 [
break; ]      ...      [ default : statement ] }

```

The *switch* statement is an ideal way to choose among many options. Given the *testvalue*, in the *case* that it takes a given *value_i*, execute *statement_i*.

inside switch

switch: The reserved word to say we're using a switch statement

case: A given case, corresponding to a given value, matched exactly. Any number of cases are permitted, and they are tested in order. The same value shouldn't occur in multiple cases (it's a compile-error in fact).

break: Optional word at the end of each case statement, instructing the program to leave the *switch* statement. If absent, then continue with the next case statement.

default: Optional, covers all cases not previously found.

switch example

Here's a simple *switch* statement to show you how they work:

```

1 public class LetterSwitch {
2     public LetterSwitch() { }
3     public int score(char ch) {
4         int s = 0;
5         switch (ch) {
6             case 'a': s = 1; break; // skip the other cases
7             case 'b': s = 2; break;
8             case 'c': s = 3; break;
9             case 'd': break;
10            default: return 0; // if ch isn't a,b,c,d then return 0
11        }
12        return s;
13    }
14    public static void main(String [] args) {

```

```
15     LetterSwitch ls = new LetterSwitch();
16     System.out.println(ls.score('a'));
17     System.out.println(ls.score('e'));
18 }
19 }
```

Alternatively, you might write it like this:

```
1 public class LightSwitch {
2     public LightSwitch() { }
3     public int score(char ch) {
4         switch (ch) {
5             case 'a': return 1;
6             case 'b': return 3;
7             case 'c': return 3;
8             case 'd': return 2;
9             default: return 0;
10        }
11    }
12    public static void main(String [] args) {
13        LightSwitch ls = new LightSwitch();
14        System.out.println(ls.score('a'));
15        System.out.println(ls.score('e'));
16    }
17 }
```

...both of which produce:

```
1
0
```

switch — case

- The value put after the case reserved word must be a primitive type like int, short, byte. As of Java 7 you can also use String, but *don't do it for your assessments!*
- If the *testvalue* is equal to the case value then that statement is executed.
- Cases don't have to be in any particular order.

From Java 7, the case is tested using the `String.equals()` method, so you'd also have to convert all strings to lowercase (or all to uppercase) if you wanted to ignore case in a switch involving Strings.

switch — break

The `break` keyword is required if you want to skip the rest of the switch statement.

If you don't have the `break` there, then the next statement will be executed, like this:

```
1 public class ScrabbleSwitch {
2     public ScrabbleSwitch() { }
3     public int score(char ch) {
4         switch (ch) {
5             case 'a': case 'e': case 'i':
```

```

6     case 'l': case 'n': case 'o':
7     case 'r': case 's': case 't':
8     case 'u': // all cases a,e,i,l,n,o,r,s,t,u
9         return 1; // don't need a break here as we return
10    case 'd': case 'g':
11        return 2;
12    default:
13        return 0;
14    }
15 }
16 public static void main(String [] args) {
17     ScrabbleSwitch ss = new ScrabbleSwitch();
18     System.out.println(ss.score('a'));
19     System.out.println(ss.score('x'));
20     System.out.println(ss.score(' '));
21 }
22 }

```

switch — default

The last case to be executed will be the default case. You don't need to give it a value for comparison, just the keyword default.

default is *optional*:

```

1 public class SwitchBrief {
2     public int score(char ch) {
3         switch (ch) {
4             case 'a': case 'e': case 'i':
5             case 'o': case 'u':
6                 return 1;
7         }
8         return 0;
9     }
10    public static void main(String [] args) {
11        SwitchBrief sb = new SwitchBrief();
12        System.out.println(sb.score('a'));
13        System.out.println(sb.score('z'));
14    }
15 }

```

```

~> javac SwitchBrief.java
~> java SwitchBrief
1
0

```

You can miss out quite a bit from the switch statement with no compilation errors – the following is fine:

```

1 public class SwitchEmpty {
2     public int emptyChoice(int i) {
3         switch (i) {
4             }
5         return i;
6     }
7     public static void main(String [] args) {

```

```
8     SwitchEmpty se = new SwitchEmpty();
9     System.out.println(se.emptyChoice(4));
10 }
11 }
```

compiles just fine. Doesn't do anything interesting though.

common switch errors

Don't forget the break!



Kit-Kat is a trademark of Nestlé

If you miss the break then execution will “fall through” to the next case. This may be what you want, but it may not. *Begin* by putting the break in, and then remove it only if you're really really sure.



switch or if/else/if ?

Sometimes it's not clear which to use: `switch` or some `ifs` and `elses`. Neither is necessarily “correct” in such cases, so here are some guidelines that might help you choose:

- cases in the *switch* cannot check a range of values, only equality (unlike for example `if (x < 5)`);
- `switch` can be very compact (see previous examples)
- A control flow statement (`switch`, `if/else/if` etc.) that hides the structure is a poor choice
- A structure that is not very general, such as using `Strings` in the `switch` when you can't guarantee the version of Java, is a very bad idea.

9 Loops and iteration

This section is on *loops*, which enable you to *iterate* over collections, either a fixed or variable number of times. Essentially all loops work by alternately executing a block of code, and running a test to see whether the block should be exited. The different varieties are all in a sense equivalent to each other, in that you can perform the same iterations with each kind of loop. You should get used to writing all the different kinds though, because as you should by now have realised it's often the clarity of the code that's the deciding factor in how you implement something!

The simplest is the `while` loop, which we'll look at first, and then the much more complex `for` loop and finally the `do while` loop.

9.1 while

`while`

Syntax: `while (condition) statement`

See §6.2 in Horstmann.

“While the *condition* is *true*, repeat what’s in the *statement*.”

This is a very simple loop that’s perhaps under-used.

Steps of while:

1. *condition* is checked: if it is *true* then go on to Step 2
2. *statement* is executed

The while loop continues until when *condition* is checked at Step 1, it is *false*.

while example

Here’s (pseudo)code for a simple while loop, to print out even numbers from 0 to 8 inclusive.

Algorithm 15: evensWhile

```

1  let i be an integer
2  i ← 0
3  while (i < 10) do {
4  ·   print i
5  ·   i ← i + 2
6  }
```

The condition is “(*i* < 10)”

The statement is the block on lines 4 &

5

```

1  import java.io.PrintStream;
2  public class Loopy {
3      public void evensWhile(PrintStream ps) {
4          int i = 0;
5          while (i <= 8) {
6              ps.println(i);
7              i += 2;
8          }
9      }
10     public static void main(String[] args) {
11         Loopy loop = new Loopy();
12         PrintStream ps = System.out;
13         loop.evensWhile(ps);
14     }
15 }
```

while (true)

What’s this? `while (true)` will always be true, so surely any while loop starting this way will never finish!

Have no fear: `break` is here. Use `break` to leave (break from) the loop, like this:

```

1  public class WhileTrue {
2      public void printOdds(int n) {
3          int i = 0;
4          while (true) {
5              System.out.println(2*i+1);
6              i++;
7              if (i >= n) {
8                  break;
9              }
10         }
11     }
12     public static void main(String [] args) {
13         WhileTrue wb = new WhileTrue();
14         wb.printOdds(5);
15     }
16 }
```

9.2 for

for

Syntax: `for (initialisation ; condition ; update) statement`

“First, do the initialisation; then, while the condition is true, execute the statement. After each execution of the statement, execute the update.”

This is the most complex loop, so perhaps it's surprising that it's also one of the most commonly seen, though is a reasonably compact way of describing some very sophisticated behaviour. The four components are called (here, and in the course textbook) *initialisation*, *condition*, *update* and *statement*.

Inside for

Here's the order of execution of the for loop:

1. *Initialisation* always happens.
2. The *condition* is checked at the beginning of each iteration through the loop: if the condition is false, then the loop isn't executed, and won't be again.
3. The *statement* is executed.
4. The *update* is executed: usually this is incrementing or decrementing an index as you move through an array, but it can be anything you like.

Steps 2-4 are repeated until, when the *condition* is checked at Step 2, it is false.

for example

What will this print?

```
1 import java.io.PrintStream;
2
3 public class ForExample {
4     public ForExample() {
5     }
6     public void printLoop(PrintStream ps) {
7         int i = 10;
8         int j = 0;
9         for (j = 1; j < i; j++) {
10            ps.println("i.j = " + i + "." + j);
11            i--;
12        }
13    }
14    public static void main(String [] args) {
15        ForExample fe = new ForExample();
16        fe.printLoop(System.out);
17    }
18 }
```

```
~> javac ForExample.java
~> java ForExample
i.j = 10.1
i.j = 9.2
i.j = 8.3
i.j = 7.4
i.j = 6.5
```

We can move things around so we can access the index variables outside, like this:

```

1  int i;
2  for (i = 0; i < 10; i++) {
3      System.out.println(i);
4  }
5  System.out.println(i); // still defined

```

Options with for

If you miss out the condition, it will never be checked. That means it can never be *false*, so you must have some other way of getting out of the loop or it will go on forever.

If you put a semicolon at the end of the for loop like this:

```

1  int i = 0;
2  for ( ; i < 10; i++); {
3      System.out.println("i = " + i);
4  }

```

What will be printed? It's a very subtle error that often occurs and can be hard to spot: the semicolon ';' near the end of line 2 above means *there is no statement to execute in the loop*, so the `System.out.println` command isn't *in* the loop.

What about this?

```

1  for (int i = 0; ; ) {
2      System.out.println(i);
3  }

```

“Gotcha’s” with for

In fact you don't have to have all the ingredients there in the for loop, and you can leave any or all of them out:

```

1  for ( ; ; ) { }

```

will actually compile!

What will it do?

9.2.1 initialisation

This is executed *first*, even if the body of the loop is never entered: for instance, given the loop

```

1  for (int x = 5; x < 5; ++x) {
2      System.out.println("Hah! Made it!");
3  }

```

the message is never printed.

9.3 do...while

do...while

Syntax: `do statement while condition`

“Do *statement* at least once, and stop if *condition* is false.”

9.4 break

`break`

This is a special reserved word that only makes sense inside a loop or a *switch* statement (see §??). It means “break from the current iteration of this loop” when in loops, and “break from the switch” when in a switch. It isn’t allowed in an *if* statement.

9.5 continue

`continue`

This reserved word means “skip the rest of this iteration and go on to the next one”.

If you put *continue* into your loop part-way through then what follows inside the loop will not be executed this time: it’s as though the execution skips immediately to the *update*.

9.6 foreach

`foreach`

The “enhanced for loop” or “for each loop” of Java is a very cool feature of the language, that should speed up your development as well as making your code simpler to read when you can apply it. [Java](#)

It needs a special kind of object called an *iterator* to work:

A Lab work

The following pages detail the lab work you will do this Semester. Don’t worry that it looks a lot – it’s intended for you to work through consistently each week, and, if you do, you’ll do well.

A.1 The OneOff Template

To begin with you might be tempted to use `public static void main` to put all your coding in. This is a terrible practice but without knowing much yet about the Java approach to Object-Oriented programming, you may be left wondering, what is the alternative?

Here’s a simple template you can use to start little programs off, to give you a good idea as to where to put things and what to call them, etc.

```
1 public class OneOff {
2     public OneOff() {
3
4     }
5     public int method1() {
6         // put your code in here
7         return 0; // must return something even in a stub
8     }
9     public static void main(String [] args) {
10        OneOff one = new OneOff();
11        System.out.println(one.method1()); // get & print the int result
12        // call the method on the "one" object.
13    }
14 }
```

Lab 1 : Hello, World! and the Terminal

Topics covered: Compiling a Java Program, terminal

Aims: Learn to use the terminal (a.k.a. command-line) for some simple commands, including compiling and running a program in Java

Practice in: lab skills

Exercise 1 (approx. 15 minutes) : Before you write a Hello from the computer to the world, you should introduce yourself to the class. The tutor will start things off: when it comes to your turn, say who you are, and something *interesting* about yourself. If you think that being a ninja crime fighter in your spare time is interesting, then do please share: just don't expect *everyone* to believe you...

Exercise 2 (approx. 3 minutes) : Log in to a PC in the lab and start up a terminal window.

Change the current directory to your "U" drive using the `cd` command. Your tutor should be able to help you.

Exercise 3 (approx. 5 minutes) : Create a directory (a.k.a. folder) in your U drive for this unit, perhaps something like "info1103", with the command

```
> mkdir info1103
```

Change to that directory with `cd info1103` and then make another directory for your lab work. It's best to be organized from the beginning! You should make a directory for each lab too.

Exercise 4 (approx. 15 minutes) : Write out the "Hello, World!" program in a text editor. **Don't use NotePad or a word-processor like Word.** You *must* name your file "HelloWorld.java". If you're really hard core you can use a command-line editor like `vi`, `vim` or `emacs`. `vim` is pretty cool.

Compile it with the `javac` command. If there are any errors, see if you can work out what they mean and fix them.

Run the program by entering

```
> java HelloWorld
```

Exercise 5 (approx. 2 minutes) : Find out what version of Java is installed on your current machine. Do this by typing `java -version`.

Fill in the Java version here: _____

Exercise 6 (approx. 5 minutes) : Go on the internet and find the Java API. Fill in the answers below:

1. URL of the Java API:
2. API stands for:
3. The API is most useful to me for:

Exercise 7 (approx. 15 minutes) : Create a program called `Box.java` to print out characters to draw a box on the screen.

```
> javac Box.java
> java Box
+-----+
|         |
|         |
|         |
+-----+
```


Lab 2 : Variables and Eclipse

Topics covered: Eclipse, variables, assignment

Aims: To write a simple program in a well established IDE to do so some simple calculations




Practice in: IDE, pseudocode

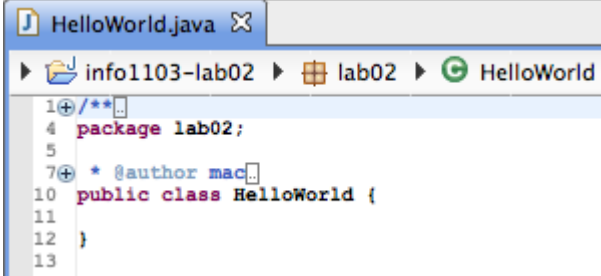
Exercise 1 (approx. 5 minutes) : What are the values of the variables below, at the places indicated?

Algorithm 16: variables

<pre> 1 let x be an integer 2 let y be an integer 3 x ← 2 4 x ← x + 3 5 y ← x - 1 6 x ← y 7 y ← x + y </pre>	<pre> // What does x equal? // What does x equal? // What does y equal? // What does x equal? // What does y equal? </pre>
---	--

Exercise 2 (approx. 10 minutes) : Start up the IDE *Eclipse* on you computer.

-  Create a Java project called “info1103-lab02”.
-  Within it, create a package called lab02.
-  Next, make a new class file in that package called HelloWorld.java.
You should then have something that looks like this:






```

HelloWorld.java
└─ info1103-lab02
   └─ lab02
      └─ HelloWorld
         1+/**
         4 package lab02;
         5
         7+ * @author mac
        10 public class HelloWorld {
        11
        12 }
        13

```

Copy your HelloWorld code in to that file from where you saved it last week.

Are there any errors? In this IDE and many others, syntax errors and even other compiler errors are highlighted. You can move the mouse over the error  or warning  image and a message should pop up to say what the problem is. Once your code is error-free, run the HelloWorld program from within Eclipse. To do that, make sure your HelloWorld.java file is selected on the left, then go to the run button .

Exercise 3 (approx. 5 minutes) : Change the view in Eclipse to *Debug* mode. Debugging is a **hugely** important skill. You shouldn't have any bugs *yet* but your goal here is to get practice in *tracing* code, which is a way of stepping through a program in tiny steps — down to a line at a time — and seeing exactly what's happening. Eclipse has a very good debugger.

Make sure your program is selected on the left – the file with your main method in it. Near the top left you should see a button with a picture of a bug on it  — click and hold that, and on the drop-down menu go down to “Debug as...” and select “Java Application”.

Exercise 4 (approx. 5 minutes) : Introduce a syntax error into your program. Eclipse (and most other IDEs) will alert you almost immediately where there is a syntax error, or where the compilation would fail. Write what you did below or in your workbook and then what the error message is.

Exercise 5 (approx. 5 minutes) : Now create a new class called `Assignments`, and when Eclipse asks you if you want to create a main method, say “yes”. Fill in code to do the calculations you worked out above and print out the answers.

Exercise 6 (approx. 10 minutes) : Write another program to print out the first 25 odd numbers.

You might find useful:

```

1  int x = 10;
2  System.out.println(x); // will print out x
3  System.out.println(x+4); // will work out what x+4 is and then print it
4  System.out.println("the answer is " + x); // will print out a nice message!
```

Exercise 7 (approx. 30 minutes) : This is programming project from another text by Horstman, Big Java (2nd ed). I’m reproducing it here slightly more briefly):



The Flesch Readability Index (FRI) is a measure devised by Flesch to gauge the legibility of a piece of text. It works like this:

Algorithm 17: *Flesch (F)*

```

1  let F be a text file
2  let w be the number of words in F
3  let s be the total number of syllables in F
4  let l be the number of sentences in F
5  I ← 206.835 – 84.6 × (s/w) – 1.015 × (w/l)           // rounded to nearest int
```

For these purposes, a *word* is any sequence of characters delimited (surrounded) by white space, whether or not it’s an actual word; a *syllable* is counted for each group of vowels together, so “real” counts as one syllable and “regal” counts as two; and a *sentence* is a set of words that finishes with a full stop ‘.’, colon ‘:’, semicolon ‘;’, question mark ‘?’ or exclamation mark ‘!’.

Write a program to read in a text file and calculate and return its FRI. For information, here are some typical values of FRI for documents you might have experienced:

document	FRI
comic	≈95
consumer advertisement	≈82
tabloid	65
<i>Time</i> magazine	57
<i>New York Times</i>	39
this L ^A T _E X document	19
approximate age	
10-11	91-100
9-10	81-90
8-9	71-80
7-8	66-70
6-7	61-65
high school	51-60

Extension: Write a program to print out a list of all the prime numbers in the range 1 to 1000.

Extension: Write a program to provide a prime factorisation of an input integer: e.g., for the input 12, it should print out 2 2 3.

Extension: The game of *Nim*.

This is a very old game in which players take turns to choose a number of pieces (counters, coins, rocks) from a pile. The object of the game is to avoid being the last player taking any pieces from the pile. Each player must take at least 1, but must not take more than half of the remaining pieces.

Write a program in which the computer plays *Nim* against a human opponent. Here's some pseudocode for how your program should work:

Algorithm 18: Nim

```

1  let n be a random number in the range 1.. 100 inclusive
2  decide who goes first at random
3  decide at random (50-50 chance) whether the computer plays smart or stupid
4  if (stupid) then {
5    · the computer chooses any legal number of pieces at random
6  } else {
7    · the computer chooses enough pieces to make the remainder one less than a power of 2
8  }
```

Do realise that the pseudocode above is horribly incomplete: it doesn't tell you to keep going until the pile of pieces is empty, for example, or to take turns. The description above should be all you need, however!

Solutions

Exercise 1 on assignments

The assignments should be very straightforward: if you have managed to understand the lectures and written a HelloWorld program you might have started by coding it as a program, like this:

```
1 package lab02;
2
3 import java.io.PrintStream;
4
5 public class Variables {
6
7     public static void main(String[] args) {
8         int x = 2;
9         x += 3; // or x = x+3
10        System.out.println("x = " + x);
11        int y = x-1;
12        System.out.println("x = " + x + " and y = " + y);
13        x = y;
14        System.out.println("x = " + x + " and y = " + y);
15        y += x; // or y = x + y;
16        System.out.println("x = " + x + " and y = " + y);
17    }
18 }
```

which would give you the correct answers:

```
x = 5
x = 5 and y = 4
x = 4 and y = 4
x = 4 and y = 8
```

Exercise 6 on printing out the first 25 odd numbers requires you to use a loop:

```
1 package lab02;
2
3 public class Odds {
4
5     public static void main(String[] args) {
6         for (int i = 0; i < 25; i++) {
7             System.out.println(2*i+1);
8             // this is the really easy way!
9         }
10    }
11 }
```

In pseudocode it's just like this: **Algorithm 19: Odds**

```
1 for (i in the range 1 to 25) do {
2     · print 2i+1
3 }
```

Running it in from within Eclipse we just get this:

```

1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
41
43
45
47
49

```

as desired.

We could have written that loop in several different ways, including these two:

```

1 for (int i = 1; i < 50; i += 2) {
2     System.out.println(i);
3 }

```

```

1 for (int i = 0; i < 50; i++) {
2     if (i % 2 == 1) {
3         System.out.println(i);
4     }
5 }

```

which are also fine, but the last one isn't quite as efficient as it goes through a loop that is twice as long as it needs to be, and only prints out a number when it's odd. This certainly wouldn't work well for printing out, say, all the numbers that were multiples of 1000!

Exercise 7 is much more challenging. Did you manage to make it work? If so you should congratulate yourself whole-heartedly for your brilliance. (You may have skipped this exercise and gone on to the extension: if so, just read through these notes and see if you understand what's going on.)

So the goal here is to open a file

how do I open a file?

and read it

how do ai read a file?

and then somehow count words and sentences and syllables...

argh

Let's look at how to open a file first. Remember that a file is just a document in your computer that has stuff in it, like text or an image. To use a file you have to open it, just like reading from or writing in a book: you have to open it to read from or write in it.

Your first approach to this might be to try out code, like this,

```

1 infile = open("thenameofmyfile");

```

but remember, in Java at least we have to declare variables before we use them. What kind of thing is a file? It's a File.

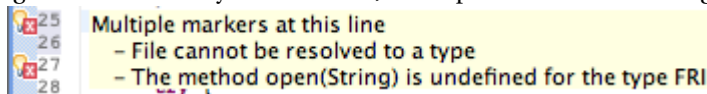
Let's expand our code accordingly:

```

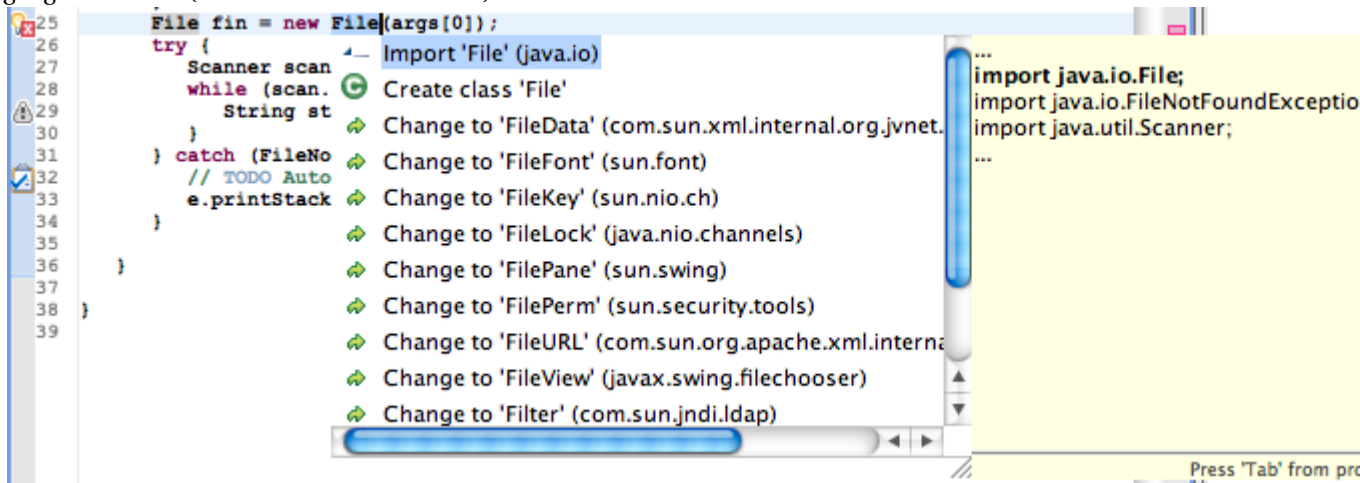
1 File infile = open("thenameofmyfile");

```

but sadly, that won't work either. The compiler will definitely complain that "open" isn't defined. One thing Eclipse will also do is say there's an error at this line, because File isn't defined either. But there's some help provided because when you mouse-over the error (remember the errors on the left hand margin of the window your code is in) it will produce a little floating box that says



So: File isn't defined and the method "open" isn't defined for our program either. Fortunately Eclipse helps you out here: when you click on the error badge a menu pops up allowing you to choose a suggested solution: the first one makes lots of sense, to import the definition of File from the Java language definition (it's the one selected below).



Now we still have the problem of how to open a file so we can use it. Until you know how to do this, check the Java API to see what the language definition is, or look up Files in your textbook.

To create a File variable you need to use the following syntax:

```
File file = new File("filename");
```

which creates the File variable for you. But you're still not done: you now have to *access* the file, and that's through using another kind of variable, the Scanner. A Scanner enables you to check whether the file has any more text in it, to read numbers or Strings from the file, or to read whole lines.

The syntax for creating a Scanner when you have a File is like this:

```
Scanner scan = new Scanner(file);
```

so your code to open a file up for reading (scanning) would look like this:

```

1  package lab02;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.util.Scanner;
6
7  public class FRI {
8
9      public FRI () {
10     }
11
12     public static void main(String[] args) throws FileNotFoundException {

```

```

13     File fin = new File("filename");
14     Scanner scan = new Scanner(fin);
15 }
16 }

```

... where I've allowed Eclipse to just fix my compilation error

```

15 public static void main(String[] args) {
16     File fin = new File("filename");
17     Unhandled exception type FileNotFoundException
18 }
19
20 }
21

```

by throwing an exception (basically, just quitting and reporting an error):

```

14
15 public static void main(String[] args) throws FileNotFoundException {
16     File fin = new File("filename");
17     Scanner scan = new Scanner(fin);
18 }
19
20 }
21

```

- FileNotFoundException - java.io
- IOException - java.io
- Exception - java.lang
- Throwable - java.lang

Right.

This is quite a bit of work just to get access to a file, but don't worry, we're nearly there.

Each line in the file can be read in, one at a time. There are built-in methods of the Scanner class, which we can use here to read a line at a time. Here's how:

```
String line = scan.nextLine();
```

... and this will work if there is a next line in the file for the scanner to look at.

Your complete code to read in a text file line by line would look like this:

```

1 package lab02;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public class FRI {
8
9     public FRI () {
10    }
11
12    public static void main(String[] args) throws FileNotFoundException {
13        File fin = new File("filename");
14        Scanner scan = new Scanner(fin);
15        while (scan.hasNextLine()) {
16            String line = scan.nextLine();
17            // ...
18        }
19    }
20 }

```

but we still haven't counted anything!

Let's think about what we need to count. Sentences are separated by the punctuation marks given: .;:?!

If we can count these we will count the number of sentences, assuming that we only have one of these punctuation marks at a time: the number of sentences will be one more than the number of these punctuation marks.

A very simple way – but not the most efficient way – do to this is to use the tools you have so far: a loop and the ability to compare chars using `==`.

We'll access the *i*th character in a String with the method `charAt`, like this:

```
String s = "hello";
char ch = s.charAt(0);
char ch2 = s.charAt(2);
```

and so on.

We can then write in our main method:

```
1 File fin = new File("infile.txt");
2 Scanner scan = new Scanner(fin);
3 int numSentences = 0;
4 while (scan.hasNextLine()) {
5     String line = scan.nextLine();
6     for (int i = 0; i < line.length(); i++) {
7         char ch = line.charAt(i);
8         if (ch == ':' || ch == ';' || ch == '.' || ch == '?' || ch == '!') {
9             numSentences++;
10        }
11    }
12 }
13 System.out.println(numSentences);
```

Given the input file `infile.txt`:

```
Hello! My name is Earl!
I live in a house: it is good. I like the house because
it has a nice garden. Do you like my garden?
```

the output gives the correct number of sentences, 6.

Counting the number of words can be done in a similar way: if you move through the line, you can check each character to see if it's a word or it's whitespace (spaces, tabs or newlines). Every time you reach a whitespace character then you must either have just finished a word (in which case you increment the number of words counter) or just encountered a whitespace character in the previous position, so you don't need to increment the word count.

The code for this is straightforward too, just not very pretty:

```
1 package lab02;
2
3 import java.io.File;
4 import java.io.FileNotFoundException;
5 import java.util.Scanner;
6
7 public class FRI {
8
9     public FRI () {
10    }
11
12    public static void main(String[] args) throws FileNotFoundException {
13        File fin = new File("infile.txt");
14        Scanner scan = new Scanner(fin);
15        int numSentences = 0;
16        int wordCount = 0;
17        boolean lastCharWasWhitespace = true;
18        while (scan.hasNextLine()) {
```

```

19     String line = scan.nextLine();
20     for (int i = 0; i < line.length(); i++) {
21         char ch = line.charAt(i);
22         if (ch == ':' || ch == ';' || ch == '.' || ch == '?' || ch == '!') {
23             numSentences++;
24         }
25         if (ch == ' ' || ch == '\t' || i == line.length()-1) {
26             if (lastCharWasWhitespace) {
27                 // don't do anything
28             } else {
29                 wordCount++;
30             }
31             lastCharWasWhitespace = true;
32         } else {
33             lastCharWasWhitespace = false;
34         }
35     }
36 }
37 System.out.println("numsentences = " + numSentences);
38 System.out.println("word count = " + wordCount);
39 }
40 }

```

This is building up on the first version, which just printed out the number of sentences, and now is counting the number of words too. Counting the number of syllables can be done the same way: looping through the line that's read in each time.

There is a more compact way of doing this, using a method built in to the `String` class called `split`. That method splits `Strings` into chunks separated by any kind of pattern you care to use, described with a regular expression. You won't be examined on regular expressions or the `split` method!

You split a `String s` into words like this:

```
String [] words = s.split("\\s+");
```

Extension 1 is to print out prime numbers. This is pretty straightforward once you remember about what is special about prime numbers, and you can use the code you wrote above for printing out the odd numbers as a guide.

Here's some pseudocode to get you going:

Algorithm 20: Primes

```

1   for (n in the range 2, ..., 1000) do {
2     ·   if (n is prime) then {
3     ·   ·   print i
4     ·   }
5     }

```

So it just remains how to test whether the number is prime. Remember that an integer n is prime if it only has the factors n and 1 as factors. That means when we divide it by anything else, there's some left over — a remainder.

In a recent lecture you saw this operator: `%`, which means “modulo” or “the remainder after division by”. So $4 \% 2$ is 0, and $4 \% 3$ is 1 as $4/3$ is $1 \frac{1}{3}$.

Testing whether a number is *even* is easy: n is even if and only if $n\%2$ is 0. So testing whether a number, say n , has x as a *factor* is just the same: n is something-times- k if $n\%k = 0$.

For a number to be prime, we need it to not have any factors other than itself and 1, so let's test them all:

Algorithm 21: Primes

```

1  for (n in the range 2, ..., 1000) do {
2  ·   for (k < n) do {
3  ·   ·   if (n%k = 0) then {
4  ·   ·   ·   n can't be prime so don't print it
5  ·   ·   }
6  ·   }
7  ·   if (n is prime) then {
8  ·   ·   print n
9  ·   }
10 }

```

We could start coding with what we have now but let's just make sure we're not being inefficient here: do we really need to check all the integers less than n ? No, we only need to check those integers that are less than or equal to the square root of n . If we find a factor k that's less than \sqrt{n} , then we know there's another factor that's bigger than \sqrt{n} , but we just don't care what it is.

So after one more modification:

Algorithm 22: Primes

```

1  for (n in the range 2, ..., 1000) do {
2  ·   for (k <  $\sqrt{n}$ ) do {
3  ·   ·   if (n%k = 0) then {
4  ·   ·   ·   n can't be prime so don't print it
5  ·   ·   }
6  ·   }
7  ·   if (n is prime) then {
8  ·   ·   print n
9  ·   }
10 }

```

we're ready to code:

```

1  package lab02;
2
3  public class Primes {
4
5      public static void main(String [] args) {
6          for (int n = 2; n < 1000; n++) {
7              boolean prime = true;
8              for (int k = 2; k*k < n; k++) { // tests to see if k < sqrt(n)
9                  if (n % k == 0) {
10                     // n can't be prime
11                     prime = false;
12                 }
13             }
14             if (prime) {
15                 System.out.println(n);
16             }
17         }
18     }
19 }

```

```
17     }
18   }
19 }
```

There's one more efficiency we should really put in here though. Look at the logic above: I have a test to see if there's a factor of n , and if there is, I set the value of `prime` to be false, so I know that n can't be prime. But then instead of simply moving on to the next value of n to test, I continue to check for other factors, even though I know this n isn't prime! Not very efficient. What makes more logical sense is to break out of the loop as soon as `prime` is false. This is how:

```
1 package lab02;
2
3 public class Primes {
4
5     public static void main(String [] args) {
6         for (int n = 2; n < 1000; n++) {
7             boolean prime = true;
8             for (int k = 2; k*k < n; k++) {
9                 if (n % k == 0) {
10                    // n can't be prime
11                    prime = false;
12                    break;
13                }
14            }
15            if (prime) {
16                System.out.println(n);
17            }
18        }
19    }
20 }
```

We'll learn more about the `break` keyword next week.

Lab 3 : Calculations and Operators

Practice in: pseudocode, operations, operators

Exercise 1 : Show and explain your Task to your tutor when they ask. Make sure you have submitted it, and your submission has been witnessed by your tutor, before the end of the lab.

Exercise 2 (approx. 25 minutes) : Get into groups of three or more, and as a group choose one of the following algorithms to write in pseudocode. Write the pseudocode individually – do *not* consult with each other.

WC count the number of words, letters and lines in a text file.

Example:

```
> java WC textfile.txt
#words = 234
#letters = 668
#lines = 15
```

GREP scan a text file for a given pattern and return the a listing of which line(s) contain(s) that pattern.

Example:

```
> java GREP foo textfile.txt
line 3: which was never a great foodstuff for a gerbil
line 16: and led inevitably to the destruction of football
line 26: Later, the foolhardy footpad's footprints on the
line 80: completely wrong-footed. What a story!
```

GO read in a set of steps on a grid as N, S, E, W, NE, SE, NW, SW and determine the final location, beginning at (0,0) (E-W is the x-coordinate and (N-S) is the y-coordinate).

Example:

```
> java GO N N NE SW SW W SE N NW S
origin: (0,0)
destination: (-2, 1)
```

TRIANGLE fill in the numbers in Pascal's triangle up to the given row number:

Example:

```
> java TRIANGLE 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Exercise 3 (approx. 15 minutes) : Write a program to perform operations on an input integer (as listed below) and print out the results each time. Your program can be called `Operators.java`. Use the `OneOff.java` template (§ A.1).

Operators to use:

```
1  ++
2  --
3  +=
4  -=
5  *=
6  /=
```

Exercise 4 (approx. 10 minutes) : In the program above add a method that uses the *ternary* operator **??** to return the maximum of two integers.

Extension: Implement the algorithm you put into pseudocode.

Lab 4 : Quiz1

This lab is in two halves so you can do the Quiz with enough space around you. You will have been told which hour to attend so please don't miss it — you'll have to reschedule — or show up early you'll have to wait.

Lab 5 : Loopyness

Practice in: the while loop and variations

Exercise 1 : Show and explain your Task to your tutor when they ask. Make sure you have submitted it, and your submission has been witnessed by your tutor, before the end of the lab.

Exercise 2 (approx. 25 minutes) : Write a Java program to print out the first 27 odd numbers using each of these formats:

- for (*initialisation* ; *test* ; *step*)
- do *statement* while *condition*
- while *condition* *statement*

Obviously when we say “write a program” we mean “write a program that compiles and runs and does what it should, and which you’ve tested.”

Lab 6 : The Switch, and Modularisation

Topics covered:

Aims:

Practice in:

Important! For next lab you'll need all the different versions of your assignment. Take the opportunity now to get hold of them all and get them organized so you can look through them next time.

Lab 7 : Task 3, Mock Practical and some Reflection

Aims: To complete a practical “mock test” on coding in Java, and to gain understanding of how your code has developed through the different stages

Exercise 1 : Show and explain your Task to your tutor when they ask. Make sure you have submitted it, and your submission has been witnessed by your tutor, before the end of the lab.

The next part of the lab is a “mock practical” test: it is designed to get you used to the pressured environment in which you’ll have to code correctly and quickly.

Before you begin, make a new Java project in your workspace called `info1103-mocktest`. Inside it, create a package called `mock`. For each of the exercises below you should create a new class, with its own `public static void main` method (“pvsM”).

Here is the set of tasks you must do. For these, *do not* look up answers on the internet! *Do not* look up the answers in your book if you can avoid it! If you’re completely stuck then ask your tutor for help. The aim of this exercise is to get you ready for your real practical test next week so cheating now won’t help you.

Exercise 2 (approx. 15 minutes) :

Squares

Write a Java program to list all the square numbers from 0 to 5000 inclusive. Put your code in the main method of a class called `Squares`. A square number is one that is the square of an integer. You’ll compile and run it like this:

```
> javac mock/Squares.java
> java mock/Squares
```

... but I won’t put the output in here as it’s slightly too wide to fit on a page...

Exercise 3 (approx. 20 minutes) :

Words

Write a Java program called `Words` to read in words from the command-line and print out a count of how many there are of each length observed. You may assume there will be no words longer than 100 characters.

Here’s how it should look when you compile and run it:

```
> javac mock/Words.java
> java mock/Words Twas brillig and the slithy toves
3 2
4 1
5 1
6 1
7 1
```

In the above the first column is the length of the word, and the second column is the number of words of that length.

Exercise 4 (approx. 15 minutes) :

Box

Write a Java program called `Box` to draw a box on screen whose width and height are provided as arguments to the program, like this:

```
> javac mock/Box.java
> java mock/Box 5 3
+---+
|   |
+---+
```

Once your programs have all been written, get into groups of three (or more if necessary), preferably without having to move far. Each person will then mark the *next* person's programs.

Mark each of these out of 6 as follows:

- 0** no effort made, program shows no understanding
- 1-2** some effort made, but with major flaws: the program probably doesn't compile, or if it does gets the answer completely wrong.
- 3-4** the program does basically what it should: it's written sensibly, it compiles (though it might have some warnings) and it shows good understanding of the problem and the programming concepts required.
- 5-6** an excellent piece of work that's clean and does exactly as it should. The very top marks are for code that is also easy to understand, with sensible variable names, good layout etc.

Ask yourself when you're marking, whether the programs you're marking deal with extreme cases properly. For instance, what should be printed out if Words is called with no arguments? What if a word of length 100 is given as input? What if the Box program is called with width and height arguments less than 2?

Now, in the same groups, gather your different versions of Assignment 1, in particular Stages 1 2 and 3. Look back at your code and answer the following questions:

[TBC]

Lab 8 : Practical Test

Aims: To demonstrate competence in programming fundamentals by implementing some simple programs.

B Tasks

The following pages detail the 5 “mini-assignments”, each worth 2% to your final mark.

Each of these Tasks must be submitted on time! If you’re late in submitting your work then you will lose marks.

If you submit after the due date then you will lose 1 mark automatically, and if you submit more than 1 day late you will get no marks for that Task.

The idea with these Tasks is that they will require you to do programming consistently through the semester. It’s well known that students who *practice* coding do much, MUCH better than students who don’t. The best incentive I’ve found to encourage this kind of behaviour is, yes, you guessed it, *marks*. The marks involved aren’t huge: it’s just 10% in total out of your 100 for the unit, but those 10 percent could easily make the difference between passing and failing, or getting a D or an HD.

The other benefit of this kind of continuous feedback through the Semester is that you can easily tell how well you’re doing. Not only will you be tested on the concepts involved in programming, from the simplest ones like how an “if” statement works and what a variable is, up to what is the difference between an abstract class and an interface; you’ll also get real feedback on how you’re putting the concepts into practice.

Don’t get someone else to do these for you. You’ll get a nasty shock at the practical test if you’ve not had enough practice in coding, and possibly a nastier one when you get to the final exam.



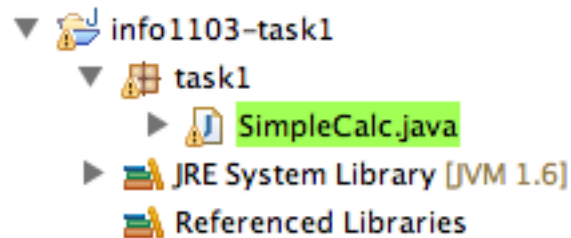
Task 1: SimpleCalc

Skills needed: command-line input, parsing Strings as ints, simple logic, compiling and running a Java program, mathematical operators, comparison of values.

Write a Java program to read in a sequence of integers and print out the following quantities:

1. the number of integers read in
2. the maximum value of the integers
3. the average value — which need not be an integer!
4. the maximum difference between any of the integers

The main class file of your program *must* be called SimpleCalc in order for the testing to work.



You might also find useful to the following code, which shows how you can convert a String variable to an int:

```
1 String str = "123";  
2 int x = Integer.parseInt(str);
```

Your program, if you run it from the terminal, should perform like this:

```
> javac SimpleCalc  
> java SimpleCalc 3 4 6 8 1  
5  
8  
4.4  
7  
>
```

Think about how many numbers might be read in: will you know in advance?

Think about how you'll calculate the maximum difference between any two numbers (there's an easy way and many harder ways).

Make sure your program prints out the results in precisely the order given above: number of integers, maximum, average, maximum difference.

Task 2: Sorter

Practice in: arrays, iteration, pseudocode

Write a Java program called `Sorter` to read in a sequence of integers, as before, but to do more with them:

1. store the integers in an array with `int []`, *not* using the `ArrayList` class;
2. sort the integers from highest to lowest value;

You must fill in the methods marked in the template you'll download: don't change the names of the methods else they won't work properly and you'll be sad.

In order to make this program work correctly you will have to get to grips with how to create arrays, how to access their elements, and how to convert from pseudocode into code. This shouldn't be too hard but start as soon as you can!

Skills needed: arrays, passing arguments, converting `Strings` to `int` values, copying pseudocode into code

Task 3: GeneScan

Skills needed: Map, file input, parsing

Practice in: Write a Java program called GeneScan that reads in a text file, whose name is provided on the command-line, and processes the contents of that file.

The goal of this program is to read text files in a very simple format that has species names, gene names and DNA “sequences”. Each (organism, gene, sequence) triplet will be stored in a *record*, and you will have to process a collection of these records.

Several useful pieces of information are desired, and you will fill in the method stubs provided to return the number of records, the number of organism names, the number of gene names, and the name of the gene that occurs in the greatest number of organisms.

Background In modern biology, in particular in *bioinformatics*, there is a huge amount of research that uses gene sequences (DNA) to compare organisms in a systematic way and learn about their evolutionary relationships. In order to estimate a phylogenetic tree that describes these relationships it's common to find genes with a good *coverage* of species, that is, genes for which sequence data are available for as many species as possible. This tiny program introduces you to one of the core activities in bioinformatics: processing potentially large sequence data files to find useful data for further processing.

Your program should have the following functionality:

1. Open a file for reading
2. Make a collection of Records (class file provided)
3. Parse the file and store the information in the collection
4. Find the number of records in the file
5. Find the number of unique organism names and unique gene names
6. Find the name of the gene that has the best species coverage (that is, has sequence information for the most species)

You must fill in the method bodies in the skeleton provided, but don't change the Record class.

You should be able to run the program like this:

```
> java task3/GeneScan data1.txt
```

To test your program you might find it useful to print out the values described above for a test input.

Task 4: MediaLibrary

This Task is to

Task 5: TreeOfGenes

This Task is using trees to store information. Sequences can get very long (the human genome has approximately 10^9 base pairs) and storing sequences can become troublesome if a compact structure isn't used. We can construct a *keyword tree* by storing one character per node, like this:

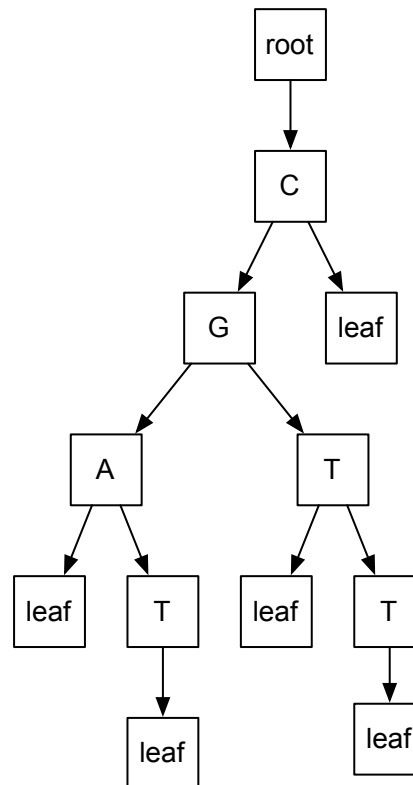


Figure 1: Example sequence tree

This tree stores the sequences C, CGA, CGAT, CGT and CGTT.

You will write a program to store DNA sequences in a tree structure containing two classes of nodes, called `InternalNode` and `LeafNode`.

Each class should be derived from the abstract class `Node`. `InternalNodes` should contain a small array of references to child nodes: one for each of the possible letters A, C, G and T¹⁷, and one for the `LeafNode`. `LeafNodes` should have no references to child nodes, and are just used as placeholders to show that a sequence ending at the parent of the leaf node is in the collection.

¹⁷corresponding to the four nucleotides Adenin, Cytosine, Guanine and Thymine

C Assignments

About assignments

The assignments you'll do for this unit are to be individual: don't share your code, or get someone else to do it! But please do feel free to discuss your ideas and problems with your tutor and classmates.

C.1 For both assignments

The assignments must be written and submitted in a particular way:

- The program *must be written in Java* and must compile and run on the School of IT Lab computers with the Java compilers installed there.
- Submission must be *via* the “push” button in Eclipse. If you have used the *push* button more than once (and you really *really* should have!) before the submission date then the last pushed version before the due date will be used. Submission after the due date will incur a penalty of 10% per 24 hour period or part thereof.

*E.g., if you are 5 minutes late you will get at most 90% for this assignment.
If you are 25 hours late you will get at most 80% for this assignment, etc.*

- You will be required to *explain* your code to your tutor, or to the Unit Coordinator (i.e., me). If you can't explain what it does, you won't get a mark.¹⁸
- Don't get other people to do your assignments for you. really.

D Assignment 1

D.1 The main task

Your task is to write a program to play a game. If you have ever played Connect Four you'll know quite a bit about what's required, but don't worry if you haven't. I'll explain. Actually, let's get Kurt to explain:



Consider an $n \times n \times n$ discrete grid, which can be filled from one side designated bottom. The game is played by two players, each of whom has a colour. The players put counters on the grid, filling from the bottom in alternating colours. The goal is to be the first player to place n counters of their own colour in a straight line, where both orthogonals (single column, single row, or single tower) and diagonals (from any corner to any other corner, that is not already an orthogonal) are permitted. Each game begins with the grid empty. The first player to gain a line of n counters in their own colour wins.

Now, in fact, it's not too bad. For a start, $n = 4$. That means at most you'll need to make a grid of $4 \times 4 \times 4 = 64$ elements, which at 2 bytes per `int` is just 128 bytes. Tiny!

Also, you won't have to actually construct a physical board either. (I did, but that was for a gift. You can pretend if you like.)

D.2 Stages

The assignment will be most easily completed in several steps. There will be three milestones — Stages 1, 2 and 3 — and the last Stage is when you do the final submission.

Each stage is given below. **Important: every time you complete and submit a stage, create a complete copy of that stage with all its files, to work on for the next stage.** You will almost certainly go back

¹⁸If you can't understand it, why would you submit it?

and change aspects of your code when you learn more, and by keeping copies of the original you will be able to see what you have changed, and why. A later assessment of this unit will require you to look at all the stages you've stored.

Stage 1 Your program should be able, given an input of a Board object with some places “filled”, to determine whether the placement of counters is valid.

Stage 2 Your program should be able, given an input of a Board object as above, determine whether anyone has won — and if they have, return the colour of the winning player.

Stage 3 Your program should in addition be able, given a Board object as above, to propose a move, with an aim of completing a line and winning. If you wish you might use some clever techniques to look ahead at possible future moves. If you want your code to beat your friends' code then this might be a good idea!

Steps Now within the Stages above you might want to proceed in smaller steps. Lots of software development works this way after all, the idea being that at every step you should have code that compiles and works. You don't start off attempting to get all the functionality at once: rather, you try to keep the program working all the time and *add* functionality, testing as you go.

1. In tutorial, get the program “Fours” to compile and run as is.

Store a copy of the original program as <login>a1stage1

2. Write the constructor for the Board class, to create the $4 \times 4 \times 4$ grid. Don't use an ArrayList. Note that the Board class is a subclass of the AbstractBoard class¹⁹.
3. Write your first Unit Test: using JUnit, create a unit test that checks to see whether a newly created Board instance is empty. To do that you will have to finish the “isEmpty()” method that's already in the code skeleton.

Here's an example of a unit test:

```

1  package mike9999;
2
3  import static org.junit.Assert.*;
4  import org.junit.*;
5
6  /**
7   * @author mac
8   *
9   */
10 public class EmptyBoardTest {
11
12     /**
13      * @throws java.lang.Exception
14      */
15     @Test
16     public void newIsEmpty() throws Exception {
17         Board board = new Board(4);
18         assertTrue(board.isEmpty());

```

¹⁹No this doesn't mean “only make changes if you want to” or “make changes if you think it's badly done”: make NO changes, ok?

```
19     }  
20 }
```

It's the `assertTrue` method that JUnit uses to check whether what you *think* should happen, actually happens.

4. Write a method in your `Board` class that checks to see if the colours assigned are valid (no pieces are floating in the air, and the colours are in the right range).
5. Write a method in your `Board` class that works out whether there is a proper line of one colour, and if so, what colour it is.
6. Write a method that, given a board and valid placement of counters, proposes a valid move.
7. Write a simulation program that creates a board and two players and creates multiple games where the two players play against each other.

D.3 Requirements

You must write code in Java that fulfils some very simple requirements. Obviously it has to compile! But there's more to it than that. You must write code that fits the following *interface*:

D.4 Assessment

The assessment for this assignment will be partly based on fulfilling certain *standards*, and partly *competitive*. In order to pass you must fulfill the standards, but in order to get the best marks you must also do better than your classmates.

Index

- ++, 28
- =, 28
- Boolean, 20
- ArrayList, 36
- iff, 35

- array, 35
- assessment
 - overview, 5
- assignment submission, 83

- bow ties
 - coolness of, 36

- contact
 - coordinator, 5
 - txtspeak, 5
- control statement, 48

- enhanced for loop, 54

- floating-point number
 - comparison, 23
- floating-point numbers
 - comparison
 - bad idea, 31
- for
 - syntax, 52

- getting help, 6

- if, 48

- Java
 - version, 3

- lab 1, 56
- lab 2, 58
- lab 3, 69
- lab 4, 71
- lab 5, 72
- lab 6, 73
- lab 7, 74
- lab 8, 76
- loops
 - for each, 54

- operand, 28
- operator
 - ++, 28
 - =, 28
 - binary, 28
 - definition, 28
 - ternary, 28
 - exercise, 70
 - unary, 28

- plagiarism, 3
- pseudocode, 16

- template
 - OneOff, 55
- tutors, 6

- while
 - syntax, 51