Queen Mary
University of London

Procedural Programming ECS401

Workbook 2020-21

Part 1 Units 1 to 4

Paul Curzon

School of Electronic Engineering
and Computer Science

ECS401 Learning Checklist

Here are the main learning objectives for this module, section by section. This is what you should be able to do by the end of the module. Tick them off as you are sure you can do them.

Overall you should be able to:

- ◻ Write Java programs using the concepts covered in the module
- ◻ Program defensively
- ◻ Explain programs and the concepts covered
- ◻ Compare and contrast related concepts covered
- ◻ Illustrate your answers with appropriate examples
- ◻ Read and analyse programs to determine what they do, justifying your answers
- ◻ Find, explain and correct errors in programs
- ◻ Discuss / argue topics on issues related to programming

0: Getting Started

- ☐ use the linux operating system to do basic things: view web pages, use editors, …
- ☐ compile and run simple programs written by others
- ☐ read, write, compile and run simple programs you have written

1: Following Commands

- ☐ explain what is meant by an algorithm and defensive programming
- ☐ read, write programs that print out a message to the user,
- ☐ read, write programs that read input from the user and store it in a variable,
- ☐ read, write programs that print out messages that include the contents of variables,
- ☐ read, write programs that are split into methods in simple ways
- ☐ read, write and use methods that both do and don't return results
- ☐ explain how your above programs work and the concepts involved.
- ☐ explain what is meant by a method, function and procedure

2: **Manipulating Values, Using Types**

- ☐ read, write, debug and run simple programs that manipulate data
- ☐ read, write and debug programs that do calculations on values stored
- ☐ read, write and debug programs that print messages that include results of calculations.
- ☐ read, write and debug programs that store and manipulate different types of values.
- ☐ read, write and debug programs that create and use simple record types.
- ☐ explain how your above programs work and the concepts involved.
- ☐ explain the importance of types

3: **Making Decisions, more on methods and records**

- ☐ read, write and debug programs using boolean expressions
- ☐ read, write and debug programs that make decisions
- ☐ explain the use of boolean expressions
- ☐ explain what is meant by if statements
- ☐ trace programs containing if statements
- ☐ read, write and debug methods that take arguments and return results
- ☐ read, write and debug programs that use accessor methods
- ☐ explain concepts related to and the use of methods

4: **Bounded For Loops**

- ☐ read, write and debug programs that follow instructions a fixed number of times
- ☐ explain what is meant by a for loop
- ☐ trace the execution of programs containing for loops

5: **Arrays, Linear Search**

- ☐ read, write and debug programs that process bulk data using arrays
- ☐ read, write and debug programs that search arrays for information
- ☐ explain how arrays are declared and used
- ☐ trace the execution of programs containing arrays

6: While loops

- ☐ read, write and debug programs that follow instructions repeatedly where the number of times is not known at the outset
- ☐ explain what is meant by for loops and while loops, their similarities and differences
- ☐ trace the execution of programs containing while loops

7: **More on Methods, abstract data types**

- ☐ read, write and debug programs that are well organised as a series of methods
- ☐ read, write and debug programs that pass information to methods using parameters
- ☐ explain what is meant by a method, function, procedure and parameters/arguments
- ☐ explain what's meant by abstract data types (ADT) in procedural programming contexts
- ☐ explain how abstract data types can be implemented from existing types
- ☐ explain a variety of data structures in terms of abstract data types
- ☐ read, write/debug programs that create / use ADTs in a procedural programming context
- ☐ discuss the importance of abstract data types

8: **References, the heap and the stack: Passing Arrays to Methods, Storage**

- ☐ compare and contrast the way integers, arrays and objects are stored
- ☐ explain how information is stored in memory in the heap or stack
- ☐ explain how information is passed to methods
- ☐ explain and compare and contrast pass by reference and pass by value

9: **Recursion, Divide and Conquer, Binary Search**

- ☐ read, write and debug programs using recursion
- ☐ implement divide and conquer algorithms
- ☐ explain how to search ordered arrays efficiently
- ☐ explain concepts related to recursion and divide and conquer algorithms
- ☐ explain grammars and parsing and how recursion is used in parsing

10: Sort Algorithms and Multi-dimensional arrays

- ☐ explain several different ways to sort arrays of data into order
- ☐ read, write and debug programs to sort arrays
- ☐ read, write and debug programs containing multi-dimensional arrays
- ☐ explain how to declare and use multi-dimensional arrays

11: Files and Character Representations

- ☐ read, write and debug programs that store information in files
- ☐ read, write and debug programs that retrieve information from files
- ☐ design simple character-based file formats
- ☐ explain how data can be stored persistently
- ☐ explain different ways characters are stored

AND ALONG THE WAY: Computational Thinking

- ☐ **algorithmic thinking:** being able to solve problems by writing algorithmic solutions
- ☐ **abstraction:** be able to use both procedural and data abstraction
- ☐ **decomposition:** be able to solve problems by splitting them into simpler problems
- ☐ **generalisation:** be able to generalise solutions to be applied in different situations

Introduction

Programs

Programming is concerned with writing instructions (‘**programs**’ or ‘**code**’) in a way that a machine, with no intelligence, can follow them blindly. By doing so the program should *guarantee* to achieve the result required every time it is run.

This module is about learning to write programs, but it is especially about *learning to program well,* though in one particular style.

Programming paradigms: Procedural Programming

This module is about **Procedural Programming**…you will need to show you can write **procedural-style programs**.

There are lots of different styles of programming: including **procedural programming**, **functional programming**, **object-oriented programming** and **logic programming**. These are called paradigms and they are just different ways to think about what a program is. There are many programming languages designed for writing programs following each paradigm. ***All are important*** and different ones are best for programming different kinds of application.

As this is an introductory programming module, we will just be looking at *one* simple style or paradigm of programming called **Procedural Programming**. It is a particular way to write programs supported by many languages. It involves thinking of a program as being a bit like a book of recipes, instructions to follow broken in to parts that each do a well-defined thing. It is a very important paradigm that is widely used. It is also a a stepping stone to other more complex paradigms including Object-oriented programming (OOP). This module does not extend to this more complex way to write programs that is OOP, though aims to make the transition easier by showing how one of the main foundational ideas used in OOP, **abstract data types**, can also be used to structure procedural programs.

We will use one language (Java) which can be used to write both procedural programs and object-oriented programs. It was chosen to make the transition to object-oriented programming easier. You will learn Java but this module is really most about learning the concepts of procedural programming in general. Focus on understanding the concepts and you will quickly be able to learn to program in any new procedural programming language.

Defensive programming

Programming elegantly and well means a lot of things, not just writing programs that do the right thing. You will also develop some of the very basic skills and ideas behind **defensive programming**. This is ultimately about:

> *writing code in a way that improves the chances it will still work even in situations you did not as a programmer expect, including where it is subsequently changed.*

Defensive programming is vital if your code is to be secure to malicious attackers, or is used in safety-critical situations (like medical devices), but it also is important for any code if it is to be relied upon.

Most code is modified to do new things or the same things better over its lifetime. A really important part of defensive programming is therefore writing code that is **maintainable.** That means it must be easy for others to understand as only then will they be able to make changes without introducing **bugs** (mistakes). A key part of this is ALWAYS writing elegant code that has **good style.** There are a lot of different aspects to this that we will see and all are important in all programs. Examples are always including in the program explanations of what it does, laying it out in a way that makes its structure clear, naming things in a way that make clear what they are for, and so on.

Another aspect of writing elegant code that is less likely to contain mistakes and easier to maintain is that of **decomposing** it in to separate easy to understand parts. In this module we will just look at procedures as a way of doing this: the simplest form of decomposition. Linked to decomposition are the ideas of **abstraction** (writing code in a way that hides details) and **generalisation** (writing code so it can be used in more than one situation)

A further important part of defensive programming is **input validation**. That is the idea of writing code so that it can deal with ALL kinds of unexpected input that could possibly occur. It should not crash or do the wrong thing if the user of the program does something unexpected.

Code that is easy to understand is also important for what is called **code audit**. This is a process where you (or better others) review the program step by step to check it does as intended. **Dry running** or **tracing** code where you work through a program step by step to check what it does, is one part of code audit that is also useful to do when **debugging** (finding and correcting mistakes in a program). This complements **testing,** where you actually run code to see if it does the right thing. To properly test code you need to run it many times with different input values and check it does the right thing in each situation possible.

Writing good documentation and being able to give clear explanations of code is also important in defensive programming to help others understand your code to both check and change it. That means writing and explanation skills are very important too and it is important you practice and so improve your writing skills as well as coding skills.

We will be expecting you to write **literate programs**. These are defensive programs with lots of explanation as part of a special literate programming notebooks. Writing them has an extra advantage in that it will help make sure you learn the programming concepts deeply.

So by the end of the module you should be aiming not just to be able to write procedural programs, but to write procedural programs elegantly. Your aim is to gain both understanding concepts and **mastery of the skill of programming**. To do that you will need to write LOTS of programs. That means writing far more programs than just the assessed exercises if you want to become good at programming. The more you write the better. Focus on doing lots of small examples on a topic so that you master it, before moving on to the next topic.

Unit 1 : Simple Programs

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- explain what is meant by an algorithm,
- read, write, compile and run simple programs,
- read, write programs that read input from the user and store it in a variable,
- read, write programs that print out messages that include the contents of variables,
- read, write programs that are split into methods in simple ways
- read, write and use methods that both do and don't return results
- explain how your above programs work and the concepts involved.
- explain what is meant by a method, function and procedure

Do the Interactive Notebooks

This unit comes with a series of interactive exercises in interactive notebooks. Before moving on to these exercises you should have worked through the Unit 0 interactive notebook

- **Interactive Notebook: Unit 0-1 Getting Started - Write a Chatbot**

(see QM+ - Unit 0 if you haven't done it yet) You may wish to do them before reading through these notes. Whether before or after, you MUST work through those exercises. These include

- **Interactive Notebook: Unit 1-1 Assignment and Input**
- **Interactive Notebook: Unit 1-2 Input Method**
- **Interactive Notebook: Unit 1-3 Further Programming Exercise**

Find the notebooks here: **https://jhub.eecs.qmul.ac.uk**

Non-technical Reading

The following booklets may help you understand concepts if struggling. They are available from the activities and reading section of QM+.

- **Chapter: 1 of Computing without Computers**
- **Chapter: 2 of Computing without Computers**

Lecture Slides/Notes

What is an algorithm?

- An algorithm is a set of precise instructions that when followed achieve some task.
- It also says what order the instructions should be followed.
- There are many ways (algorithms) to achieve one task (eg Obtaining Pizza)
- A computer can appear to be intelligent just by following instructions blindly (eg Noughts and Crosses)

What is a program?

- A program is an algorithm written in a special language.
- It is a piece of text written in a way that the computer can understand.
- The language has to be very precise.
- There can't be any confusion over what is meant.
- It must cover all possibilities
- Human languages leave to much scope for confusion.

Noughts and Crosses Algorithm

1. Go in a corner.
2. If the other player went in the opposite corner then go in a free corner.
Otherwise go in that opposite corner.
3. If there are two Xs and a space in a row then go in that space.
Otherwise if there are two Os and a space in a row then go in that space.
Otherwise go in a free corner.
4. Repeat Instruction 3 above.
5. Go in the free space.

Full programs

In the practical activities that you do in the Interactive (Jupyter) notebooks, you will not be writing full programs to start with, just fragments of program. However, it is important you understand what a full program looks like, so we will outline the missing parts.

In the interactive notebook exercises we might write a fragment like

```
public static void helloMessage ()
{
   System.out.println("Hello World!");
   return;
} // END helloMessage

helloMessage();
```

The interactive notebooks allow you to miss some of the structure of real programs (they add them for you invisibly).

An example full program

The full version of the above code fragment, as you would need to write it outside a notebook (with the above fragment in bold) is:

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage

} // END class hello
```

We will step through the extra bits below. As you need to install software to write full programs, we will leave actually writing full programs for now, but here we will talk you through the parts so you know what to expect.

Grouping the parts of the program together

- All the separate parts (the procedures) of the program are bound together into a class.
- The important thing here is the open and close curly brackets. Think of them for now as meaning this is the beginning of the program and this is the end of the program.

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage

} // END class hello
```

```
class name
{

    <the program goes here>

} // END class hello
```

Name of a program

The name of the program comes immediately after the word class. When you save programs in files, you need to use this as the file name (with a .java ending).

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage

} // END class hello
```

Where the work gets done
The actual instructions doing the work (here printing messages) are buried deep in the program.

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage

} // END class hello
```

Printing messages to the screen

- This program prints messages to the screen
- To print messages to the screen use:
- System.out.println("The message goes here!");
- System.out.println is the command you use to tell the computer to print a message.
- The message goes inside the brackets and inside the quotes.

Input and Output

- Java input and output is more complicated than the simple thing of just saying print this to the screen!
- There are also many variations you could be trying to do it
    - so different ways to achieve the same aim.
- Different text books all do it in slightly different ways
- We will use one simple way in the lectures but it doesn't really matter which you choose to use

Programs as a recipe book

- Think of a program like a recipe book with a series of recipes (the **methods**).
- Each recipe (method) says how to do one specific task.

main: where the program starts

- All Java programs have a method called main
    - It is where the computer goes to start
    - It then says which other code to execute
    - Like saying which meals are on a menu and the order

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage

} // END class hello
```

Methods

- **Methods are instructions grouped together and given a name**
- This program has 2 methods called main and helloMessage
- Note the instruction in main that says
- "go do the instructions in method helloMessage"
- this is know as a method **call**

```
class hello
{
    public static void main (String[] param)
    {
        helloMessage();
        System.exit(0);
    } // END main

    public static void helloMessage ()
    {
        System.out.println("Hello World!");
        return;
    } // END helloMessage
} // END class hello
```

System.out.println

- System.out.println is just a method too (a recipe or chunk of code you've given a name to) that the inventors of Java wrote for you
  ```
  System.out.println("The message goes here!");
  ```
- This is actually just jumping to a chunk of code someone else wrote called ***System.out.println***
- Compare it with the command:
  ```
  helloMessage( );
  ```
- The brackets say "this command is a method call - go execute the code linked to the name of the method then come back here (to this instruction) when done"

return

- Every method ends with a return command.
- It returns control back to the method that called it so it can carry on where it left off.
- main is an exception here it uses System.exit to do a similar thing – it returns back to the operating system

A bunch of hieroglyphics

- The rest is stuff just just accept goes in all your programs
- You do not have to understand every detail to write simple programs
- we will talk you through the detail gradually as we cover the concepts involved

Structure

- The curly brackets mark the *structure* of the program
- where the program or the methods with in it start and end

Some terminology

- **method**: a named operation (like main, System.out.println)
- **class**: a named group of related methods (hello)
- **keyword**: (or reserved word) a word that has a special meaning to Java and cannot be used for anything else, e.g. if, new, return, while

Output Exercise

- Write a program based on the above that prints out: *Hello my name is C3PO* to the screen.

Variables: Storing things for use later

- Variables are what languages use for storing information in a way that lets you get at it again later.
- A variable is like a named box into which you can put a piece of data.
- However it is like **a box with built in photocopier and shredder**

Variables

- If you want to use a box to store something in, you have to do two things:
  - get a box of the right kind/size
  - put something in it
- It's the same with variables.
  - declare the variable (tells the computer to create the kind of box you asked for)
  - initialize it (tells the computer to put something in)

Declaring Variables

- You include a statement in the program introducing the variable:
- String name;
- This tells the computer to create a variable (a storage space) called name suitable for holding a string of characters.
- It does not put anything in the box. It declares the variable called name

Putting values in Variables ("Assignment")

- The command:
```
name = "fred";
```
- puts the string of characters "fred" in a box called name
  "Box ***name*** gets the string ***"fred"*** "
  - When you put something in a variable **whatever was there already is destroyed** (thats where the 'shredder' comes in!)

Accessing the contents of a variable

- You get at the contents of a variable by giving the variable's name
```
name1 = "fred";
name2 = name1;
```
- This first stores a string of text "fred" in a box called name1 and then makes a copy of it and puts that into name2.

- That means we now have 2 boxes with the same text stored in both.
    - When you put get something from a variable **you dont change what is there you just make a copy of it** (thats the 'photocopier' bit of the box!)

Getting Information from the user - input

- We've seen how to print things to the screen.
- How do we get data from the keyboard in to a program?
- There are several ways based on different libraries
- One easy way is to use the Scanner library. First you need to import all its methods:

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

- The interactive notebooks preload libraries like this for you so when writing fragments in a notebook you don't have to think about them. When writing a full program you need to load any library that program needs.
- Then create an actual scanner linked to the keyboard (which is referred to in a program as System.in). This is done with the line

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

- This creates a method called `scanner.nextLine`
- It returns a String consisting of whatever line of text is typed
- We can store the string it returns in to a variable to be used later eg

```
name = scanner.nextLine();
```

- However before you do that you need to print a message to the screen to tell the user they have to type something!
- Here is a **full** program that does that.

```
import java.util.Scanner;
class keyboardinput
{
 public static void main (String[] p)
  {
     askquestions();
     System.exit(0);
  } //END main

  public static void askquestions ()
  {
     String name;
     String theylove;

     // create a scanner so we can read the command-line input
     Scanner scanner = new Scanner(System.in);

    //  ask for the person's name
    System.out.println("What's your name?");

    // get their name as a String and store in variable
    name = scanner.nextLine();

    //  ask a question
    System.out.println("Who do you love?");

    // get their input as a String
    theylove = scanner.nextLine();
    System.out.println("Oooh! Everyone listen! " + name
                       + " loves " + theylove);
    return;
  }// END askquestions
}
```

- The scanner library has many different methods to make input easy. Google it to explore the possibilities. Experiment by writing simple programs to make sure you understand (and share what you find)!

Methods, Functions and procedures

- **Methods** can be either **functions** and **procedures**.
- A **function** "returns" a value that you can use later in the computation and is used eg on the right hand side of an assignment to the value it returns gets stored in the variable:
    - `name = scanner.nextLine();`
- A **procedure** simply does something (no value returned to put somewhere):
    - `System.out.println("Hellooooo");`

Returning Values from methods

- You can write your own methods that return results
- Just include a value as part of the return statement

  `return name;`

  rather than just

  `return;`
- This passes back as the answer from calling the method whatever is in variable **name**;
- You must also change the keyword void in the header line of the method to **String** eg

  `public static String askname ()`
- It just says the method is to return a String rather than nothing.
- It allows the compiler to check that it does and warn you if you forget to return something!

```
public static String askname ()
  {
     String name;
     Scanner scanner = new Scanner(System.in);

     System.out.println("What's your name?");
     name = scanner.nextLine();

     return name;
  } // END askname
```

Calling a method that return values

- To call a method that returns a value you must do something with the value returned
- One way is to put it on the right hand side of an assignment

  `yourname = askname();`
- As we've seen with the methods in the system - compare the above with

  `name = scanner.nextLine();`
- The difference is just in the name of the method using one we've defined instead of an existing one.

Input/Output Using pop up windows

- The following program uses pop-up windows to do input and output:
- Notice how it just uses different methods to do the work

- They are taken from a different library (the swing library) loaded at the start .

```
import javax.swing.*;  // import the swing library for I/O
class inputbox
{
    public static void main (String[] param)
    {
        askForFact();
        System.exit(0);
    } // END main

    public static void askForFact()
    {
        String userfact = JOptionPane.showInputDialog
                     ("Go on tell me something you believe!");
        JOptionPane.showMessageDialog(null,
               "So... you think...″ + userfact + " do you");
        return;
    } // END askForFact
} // END class inputbox
```

An input method using pop up windows

- If you write the following at the top of your program then it makes available more sophisticated methods with names starting JOptionPane for input and output
  import javax.swing.*;
- The method JOptionPane.showInputDialog both prints a message to the screen and waits for the user to type something. That string of characters is then assigned into the variable by a normal assignment

```
String userfact = JOptionPane.showInputDialog
                 ("Go on tell me something you believe!");
```

A new output method

- ***JOptionPane.showMessageDialog*** is a method that prints a message to the screen a bit like System.out.println but in a popup box with an OK button.

```
JOptionPane.showMessageDialog(null, ″Hello″);
```

- Prints Hello to the screen in a box
- You can combine text with the text stored in variables.
- If variable (box) called *userfact* holds the string of characters “Jo loves me” because that is what the user typed in earlier. Then:

```
JOptionPane.showMessageDialog(null,
       "So... you think...″ + userfact + " …do you");
```

- Prints "So... you think…Jo loves me … do you” to the screen

Input Exercise

- Write a program that asks the user for their favourite film, then prints out that it likes it too eg:

```
What is your favourite film?
The Matrix
The Matrix! I like that too
```

- where the bold part is typed by the user, the rest is typed by the computer.

Further Programming Exercises

Before moving on to these exercises you should have worked through the Unit 0 interactive notebook (see QM+ - Unit 0 if you haven't done it yet) as well as the first two Unit 1 Interactive Notebooks (See the start of this unit).

These exercises are in the third Interactive notebook (so you may already have done them!)

- **Interactive Notebook: Unit 1-3 Further Programming Exercise**

1. Here is a program fragment (a method).

```
  public static void storeHelloMessage ()
  {
     String myWelcome;
     myWelcome = "Hi, there. This is Eddie your shipboard computer";
     System.out.println(myWelcome);
         // Note giving the variable name here is as
         // though the whole message was typed here
      return;
 } // END storeHelloMessage
```

Write out a new version of this method to print a different message, but without changing the System.out.println command.

(e.g. Make it print "Go ahead Punk. Make my day!)

2. Write a method (based on the above) to store the message to be printed in a variable with the name **happymessage** instead of **myWelcome**. You will need to change the variable name everywhere that it appears not just once.

3. Modify it so that the program prints out your name and address formatted for an address label.eg. It should store the name, address and postcode in different variables.
Paul Curzon
54 Programming Towers
Shell Street, Java Land
E1 4NS

4. Write a program fragment that prints out your initials in large by printing a pattern of letters. Have each initial printed out by its own method.

eg my initials are PC so my program would print:

```
PPPPPP
P    P
PPPPPP
P
P

CCCCCC
C
C
C
CCCCCC
```

5. Take your program that prints your initials out in large letters above and make it print each letter twice by calling the methods twice. You should only have to write commands to spell out each letter once.

6. Write a program that combines two messages using concatenation: “Dave is the best rapper ever.” and “Blondie is the best rock-pop” band ever”, It should be concatenated on to the end of fullmessage declared with

```
    String fullmessage = “”;
```

which is then printed.

7) Download the program **keyboardinputwithmethods.java**. It is based on an earlier simple program that inputs data and prints a message like: “Oooh! Everyone listen! Jim loves Jo”. Here it is split in to methods that return answers, rather than just printing things. Those answers are stored in variables. Compile and run it and see if you can work out how the information that is input is passed back from the methods to then be used.

Modify the program to have another method that asks for a third name “Who loves you?” and then print a statement like: Jim Loves Jo but Sam loves Jim.

8) Write a program that writes a message about the user’s country of birth. First write a method that asks for the country where a person was born. Then write another method that asks for their favourite city in that country. Finally, write a third method that calls the others and prints a message about their answers: eg if the answers were Poland and Krakow it prints: “Krakow is the nicest city in Poland”.

9. Understanding bugs and error messages

Take the hello world program (hello.java) from unit 0 and experiment with making single changes that introduce errors that cause the compiler to complain. Compile them, make a note of the compiler's error message and work out what the error message means given you know what mistake you introduced. If it compiles then run it to see if it still runs or if that gives errors. Then correct the program and recompile it, before introducing the next error so that there is always only one problem with the program. For example:

a) Delete one of the semicolons.

b) Change the word main to man.

c) Delete a single line

d) Remove the quotation marks from around Hello World!

Catching Bugs

Everyone catches colds. Everyone who writes programs gets bugs. One of the skills of programming is to come up with ways that help you work out what you did wrong. Early in your programming career it will mainly be getting bits of the language wrong: mis-spelling things, getting the punctuation wrong,... Those things the compiler spots for you, but you have to learn how to work out what the compiler is telling you.

TIP 1: Keep a bug book - where you record the error messages, what the problem was and what the fix was...so when you do it again you won't waste time working it all out again.

TIP 2: If there are multiple errors then just focus on the first one, until you are sure you have solved it (some of the others may be caused by it anyway.

TIP 3: The mistake may not be on the line indicated but shouldn't be beyond it. It could be at the end of the previous line (like a missing semicolon!)

Compile time errors are grammar and spelling mistakes that the compiler picks up - you haven't created a complete program yet. Once compiled there are run-time errors to come. Run time errors are when the program does run but does the wrong thing in some way. The simplest kind are ones where it just prints the wrong message! Note that spelling mistakes in the messages a running program prints are run-time errors. Spelling errors in the program commands themselves (other than the messages) are likely to be compile-time errors (the compiler will complain)! There are much more serious run-time errors to come though!

TIP 4: Check the messages printed by your program very carefully when it runs - are there any spaces missing or two many. Should it move on to a new line but doesn't? Are there any spelling mistakes in it or in variable names?

TIP 5: (This happens to everyone at some point) If you made a change and nothing at all seemed to change when you run it, double check you did save the file (and in the right place - check its time stamp changed) and also that you did compile it (did you compile the right file). Check the name of the file is the one you are running. Try adding in an extra statement to print the message "ARRRggggggHHH!" (or other suitable message) at the start. If even that doesn’t appear when you run the program, you almost certainly didn’t save it or compile it (or you are running the wrong file). If you haven’t recompiled it you will just be rerunning the old program over and over.

Avoiding Bugs

It’s always better to avoid a bug in the first place, rather than cure it afterwards. That is where programming "style" comes in...Be a stylish programmer and make your programs easier to follow.

TIP 1: When ever you type an open bracket of some kind ('(', '{') or double quote. Immediately type the matching quote. Then go back and fill in between them. It is so, so easy to forget the end one otherwise. I DO RESEARCH ON HUMAN ERROR SO BELIEVE ME ON THIS - you will save yourself so much anguish if you get into the habit of doing this.

TIP 2: When writing a close curly bracket, add a comment to say what it is closing. Missing matching brackets is very easy and this way it will be easier to see which one is missing.

TIP 3: Use indentation to help you see the structure and separate parts of a program. Put each curly bracket on a newline and then push inwards (indent) by the same amount all the commands that follow, up to the close curly bracket. That way it is easy to see where a method or class starts and ends and that the brackets match.
The structure of a program then becomes easier to follow:

```
class wombat
{
    public static void main (String[] param)
    {

    } // END main

    /* *************************
    */
    public static void carrots()
    {

    } // END carrots
} // END class wombat
```

Some common compile time errors

```
pc$ java hellotwice
Exception in thread "main" java.lang.NoClassDefFoundError: hellotwice
```

Oops I forgot to compile my program (turn it into a form to execute it).

```
pc$ javac hellotwice.java
pc$ java hellotwice
Hello World on this most wonderful day, it's a joy to be alive!
Hello World on this most wonderful day, it's a joy to be alive!
```

Input boxes

```
pc$ javac inputbox.java
inputbox.java:34: cannot find symbol
symbol  : method showMessageDialog(java.lang.String)
location: class javax.swing.JOptionPane
        JOptionPane.showMessageDialog("So... you think..." + userfact + "do you");
                   ^
1 error
```

Oops, I forgot to add the null argument at the start of the showMessageDialog command. Notice the error is about finding symbols - it is confused thinking you must have meant a different command as you didn't give enough information for this one.

```
JOptionPane.showMessageDialog(null, "So... you think..." + userfact + "do you");
```

Unit 1 Exam Style Question

Attempt the following questions in exam conditions. A model answer can be found on the ECS401 website. Compare your answer with the model answer. How good are your explanations? How good is your code?

Question 1 [25 marks]
This question concerns writing simple programs
a. [9 marks]
Explain what the following program fragment does, both line by line and overall.

```
String name;
Scanner scanner = new Scanner(System.in);

System.out.println("What is your name?");
name = scanner.nextLine();
String response = "Hello " + name;
System.out.println(response);
```

b. [6 marks]
The following program fragments contain bugs. **Identify** and **Correct** the bugs justifying your corrections.
i)
String hello1 = "Hello " + "There;

ii)
String hello2 = "Hello" + "There";

iii)
strng hello3 = "Hello There";

c. [10 marks]
Write a Java program using that asks the user what colour eyes they have, then prints out a message confirming it:

What colour are your eyes? blue
So your eyes are blue are they. I like blue.

A second run might go (where the user types brown rather than blue):

What colour are your eyes? brown
So your eyes are brown are they. I like brown.

Explaining concepts

To show that you understand the programming concepts you will be asked to explain them. An "explain" question is basically asking you to teach. Someone reading your explanation ought to then understand it themselves and so be able to clearly explain the concept to someone else. A good idea when giving explanations is to use an actual example (in an exam think your own up to show you can) and then explain the example.

Here is a typical question.

1. Explain what is meant by a *variable* with respect to programming languages.

Here is a very poor answer:

*A **variable** is a named place that a program can store values in.*

Whilst it is strictly correct it only barely explains the concept, and unless the question was only worth 1 mark you would not get much credit for it.

Here is one good answer (there are many good answers not a single "right" answer):

A variable is a named place used in a program to store data as the program is executed. Data (a value) is stored in the variable so that it can be accessed at a later point by using the name of the variable. The variable can store different values at different times depending on what is put in to it. However, if something new is stored the previous value is lost. In languages like Java, variables have a type associated with them. It determines what kind of data (eg numbers or Strings) can be stored in the variable.

In Java variables have to be declared before they can be used. For example, the statements

```
int a;
String b;
```

*declare two variables, one called **a** that can hold integers (as indicated by int) and the other called **b** which can hold strings of text.*

Once declared the variable can be used in commands that are later executed. For example

```
a = 5;
```

stores the value 5 in variable a. If later in the execution of the program we executed the statement

```
System.out.println(a);
```

then, unless it had been changed by another instruction in the interim, the 5 would be retrieved from the variable a and printed to be screen.

*A variable can be thought of as a little like a box. Things can be put in the box for safe keeping and later retrieved. The box has a name (like **a** above) so we know which box we are referring to. A difference to physical boxes though is that when we use the value of a variable we do not remove it but make a copy of the value that was stored. Also adding a new value destroys the previous value that was in the box.*

For example

int c = a;

copies whatever was in variable a into new variable c. Variable a still retains the value though so both would hold value 5 after this is executed if 5 had previously been stored in a.

This is a good explanation because it not only explains what a variable is in detail, but also gives examples and explains how they illustrate the concepts concerned. Note the examples are not just dumps of whole programs but small fragments to illustrate specific points.

This answer could be improved still further by, for example, using diagrams to illustrate the answer.

Example questions:
Here are some of the concepts you should be able to explain by now (possibly after doing some reading around the topics first)...

Explain what is meant by:

a) a compiler
b) compiling a program
c) executing a program
d) a variable
e) a value
f) declaring a variable
g) initialising a variable
h) assignment
i) an input method
j) an output method
k) an algorithm
l) a keyword
m) a program
n) a method
o) a class
p) a statement

Try answering the above questions, thinking up your own examples to illustrate your explanations.

Concept maps
Try drawing a concept map of how the concepts you have come across are linked. Draw a circle for each concept with arrows between linked concepts. Label the arrow with an explanation of how they are linked. For example a program is made up of a series of methods, so the program circle would be linked to the method circle. The arrow would be labelled "consists of a series of". An assignment is an example of a statement. You can write concept maps about the syntax and structure as above but also about how they execute.

Draw a concept map for the concepts of

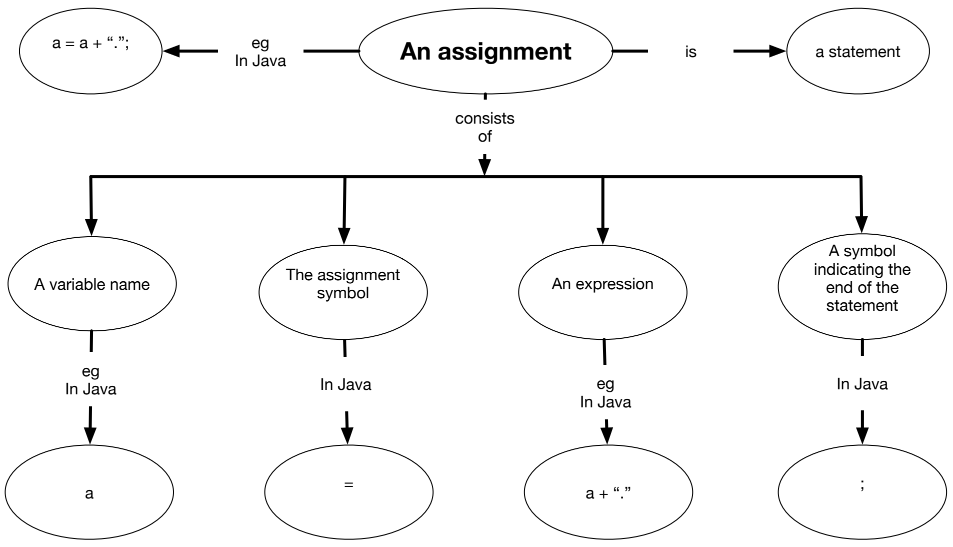
i) "The syntax/structure of assignment statements" and then
ii) "The execution of assignment statements"

Do this before looking at my "expert" ones overleaf. Compare yours to mine and work out what you missed or misunderstood.

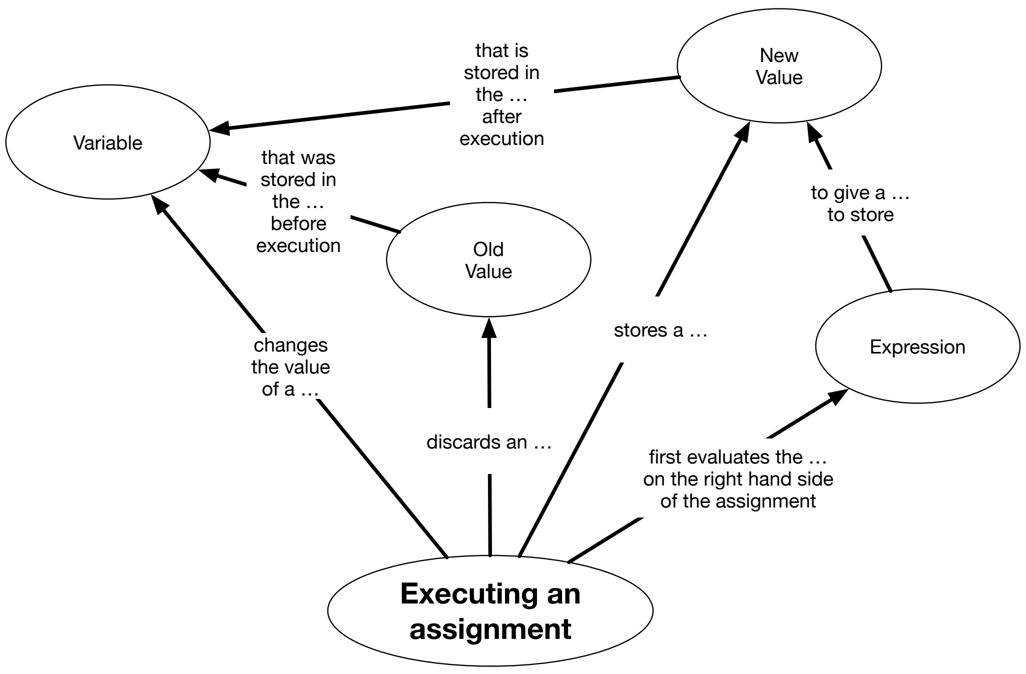
Keep drawing Concept maps
It can be helpful to draw concept maps for each set of new concepts as you come across them. As you understand more you will find you can draw better concept maps, and drawing new ones about old topics is a good way to support your learning.

There are more examples of concept maps on the QM+ site (but you will get most benefit by trying to do them yourself before you look at my expert ones).

An assignment : syntax/structure



An assignment: semantics
What happens when it executes



Executing an assignment
Variable
New Value
Old Value
Expression
that is stored in the … after execution
that was stored in the … before execution
to give a … to store
changes the value of a …
discards an …
stores a …
first evaluates the … on the right hand side of the assignment

An exam question marking exercise - variables

a) **Explain** what is meant by a variable assignment, **illustrating** your answer with reference to the following statement: a = a + "."; **[3 marks]**

Here are some different student answers to the above mid term test question. Rank them in order of best answer to worst (justify why), and then award 0, 1 2 or 3 marks to each. (My rating is on QM+).

Student A

Variable in a computing language is a box where we can store values, which can be used later when called. In a programming language there are different types of variable. For e.g.:- int variable, int stands for integer, these kinds of variables can store integer values. The other types of variable are string, character (char) etc

To define a variable, we first need to define its type.

Int variable = 20;

String variable = “computer science”;

Char variable = a;

The variable type is always written on the L.H.S i.e. before the variable. The value of the variable can also be updated. When you assign the variable with different value. When a new value is initialised in a variable the old value is lost. For eg:-

int a = 1; // Here the value of variable a is 1

int b =2;

a=b; // here the value of variable a is updated to 2

Student B

A variable is a named place where you can store data of a certain type that can be used later on by calling the name. A variable assignment is when you store something into that variable or add something to it. So if “a” is a variable and you initialise it

a = “Bye”

and then you assign it

a= a+ “.”

the new “a” will print the following message (if you did print it):

“Bye.”

Student C

As the name suggests variable assignment refers (in computer programming) to assigning variables to the program e.g. a = a+ “.”; here the assigned variable is “.” In computer programming various different types of variables can be assigned. The assigned variable values differs from program to program. Another example of an assigned variable can be (Where “:” is the assigned variable.) – System. out. println (answer 1 + “:” + test 1);

Student D

Variable assignment is changing the value of a variable which uses the equal sign ‘=’ So in the above example a is now equal to a + “.” if a was equal to “Hello” before, it would now be equal to “Hello.” and whenever we use a it will return the string “Hello.” Anything previously in the variable is lost.

Student E

Variable assignment is the instruction by code to give a value to a named variable. In this example, the variable could have already obtained a value which could be eg a=”green”; so assigning a variable with a= a+ “.”; would mean that the variable when displayed would show “green.” instead of “green”.

Student F

A variable assignment is a command in a programming language that stores a value for use later in the program. In doing so it destroys any previous value stored there. For example, a = a+ “.”; assigns a new value to the variable with name a. The name on the left hand side of the = (ie assignment) symbol (here a) indicates which variable is to be changed. The expression on the right hand side of the = symbol evaluates to a value when the assignment is executed. That value is first determined and then stored in the variable. In the example, suppose a originally holds the value “blah”. Then evaluating the expression creates a new value by concatenating “.” on to it to give “blah.” This string “blah.” is stored in to variable a. If a is accessed later in the program then “blah.” is now retrieved. The original value is lost.

Unit 2: Types and Values

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- read, write and run simple programs that manipulate data
- read, write programs that do calculations on values stored
- read, write programs that print out messages that include the results of calculations.
- read, write programs that store and manipulate different types of values.
- read, write programs that create and use simple record types.
- explain how your above programs work and the concepts involved.
- explain the importance of types

Before moving on to this section you should be able to:

- explain what is meant by an algorithm,
- read, write, compile and run simple programs,
- read, write programs that print out a messages,
- read, write programs that read input from the user and store it in a variable,
- read, write programs that print out messages that include the contents of variables,
- explain how the above programs work and the concepts involved.

Do the Interactive Notebooks

This unit comes with a series of interactive exercises in interactive notebooks. A link is in the. You may wish to do them before reading through these notes. Whether before or after, you MUST work through those exercises. These include

- **Interactive Notebook: Unit 2-1 Assignment**
- **Interactive Notebook: Unit 2-2 Types**
- **Interactive Notebook: Unit 2-3 Records**

Find the notebooks here: **https://jhub.eecs.qmul.ac.uk**

Non-technical Reading

The following booklets may help you understand concepts if struggling. They are available from the activities and reading section of QM+.

- **Chapter: 3 of Computing without Computers**
- **Chapter: 4 of Computing without Computers**

Deeper Reading

The following booklets go into the topics in this section in more detail. Read them if you find the concepts hard to understand. You can download them from the activities and reading section ot the QM+ site.

- **Booklet: What does it all mean**
- **Booklet: Box Variable**
- **Booklet: Types and Cast Operations**

Full Programs

In the interactive notebooks you are only writing grogram fragments. This is fine for learning the concepts and gaining confidence, but ultimately you need to be able to create actual Java programs that you can run on your note book.

Installing Java

- First you need to install Java
- It is free and works on any computer
- However, installing it can be tricky depending on the computer you have
- Don't give up if it goes wrong, get help either from another student who has done it or a lab demonstrator.
- Follow the instructions in the Unit 2 Interactive Notebook to install Java on your system.
- *If you do struggle to install Java, carry on doing exercises in the Interactive Notebooks while you sort it out, so you do not get left behind.*

Compiling a program

- It is very important that you understand the difference between COMPILING a program and RUNNING a program
- The interactive notebooks merge them together.
- Before a program can be run (or executed) it must be converted to an executable form
- The thing you think of as the program that you wrote is not in itself executable.
- You must translate it into an executable form
- This is done by a **compiler** and is called **compiling** the program
- The compiler first checks for simple mistakes in the program
  - it checks it does not contain **syntax errors** such as
    - spelling mistakes
    - missing punctuation
- Only if there are no mistakes does it convert it to a form that can be executed.

To create and run a program

You must go through the following steps. See the interactive notebook for more detail of this.

1. create the program using a text editor
2. save it as a file
3. compile the file (convert the instructions in the program to a form the computer can follow) . Type in a command prompt
   - javac hello.java
4. run the compiled program (tell the computer to follow the instructions in thecompiled program). Type in a command prompt
   - java hello
5. If you have problems with steps 3 or 4 go back to 1.

Some common pitfalls when working with full programs. . .

- name of file MUST be *<name of program>*.java
  - o Java puts the executable in <*name of program*>.class
  - o *not <name of file>*..class
- change but don't save
  - o If you don't save the file you recompile the old version
- change but don't recompile
  - o If you don't recompile, you run the old executable

- The program files are in a different directory/folder to where you are trying to compile or run it

Lecture Slides/Notes

A program fragment from last week:

You should be able to write program fragments in an interactive notebook like the following

```
public static void askForFact()
{
    String userfact =
                JOptionPane.showInputDialog
                     ("Go on tell me something you believe!");
    JOptionPane.showMessageDialog(null,
               "So... you think...″ + userfact + " do you");
    return;
} // END askForFact

askForFact();
```

A full program from last week:

The full program version of this is:

```
import javax.swing.*;  // import the swing library for I/O
class inputbox
{
    public static void main (String[] param)
    {
        askForFact();
        System.exit(0);
    } // END main

    public static void askForFact()
    {
       String userfact =
                JOptionPane.showInputDialog
                     ("Go on tell me something you believe!");
       JOptionPane.showMessageDialog(null,
               "So... you think...″ + userfact + " do you");
       return;
    } // END askForFact
} // END class inputbox
```

Types: Exercise

- Put the following into groups of similar things:

  1 true “hello” 5 ’p’ 17 false 6.2 ’+’ “12345” 3.14 ’a’ 2.3 ’z’ “afghj”
- What properties do they share?
- What operations can they be used for?

Types

Grouping values into different types with similar properties means:

- the computer can make sure there is appropriate storage space for them, and
- make sure the instructions only tell it to do sensible things with them.
- so you do not try to, for example, multiply two strings together (as it is meaningless).

Example Method

```
    public static void add3ages()
    {
       Scanner scanner = new Scanner(System.in);
       int age1;  // each will hold the age of a different child
       int age2;
       int age3;
       int totalAge; // the answer when the three ages are added
       int averageAge; //their average rounded as an integer
       // Set the ages to specific values
       //
       System.out.println("Give me an age");
       age1 = Integer.parseInt(scanner.nextLine());

       System.out.println("Give me an age");
       age2 = Integer.parseInt(scanner.nextLine());

       System.out.println("Give me an age");
       age3 = Integer.parseInt(scanner.nextLine());

       // Now do the calculation of the total age
       //
       totalAge = age1 + age2 + age3;
       averageAge = totalAge / 3;


       // Finally give the user the answer
       //
       System.out.println("The total age of the three children is "
                          + totalAge);
       System.out.println("and their average age is " + averageAge);
       return;

    } // END add3ages
```

Variables: Storing things for use later

- **Variables** are used by computer languages for storing information in a way that lets the program get at it again later
  - o ie commands can work on data collected earlier.
- A **variable** is like a named box that you can tell the computer to put some data in to.
- Different types of data need boxes of different shapes and sizes to store them
  - o integers, decimals, strings of characters, single characters, true or false values, output channels...

Variables

- If you want to use a box to store something in, you have to do two things:
  - get a box of the right kind/size
  - put something in it
- It’s the same with variables.
  - **declare** the variable (tells the computer to create the type of box you asked for)
  - **initialize** it (tells the computer to put something in it)

Declaring Variables

- You include a statement in the program introducing the variable:

  int X;

  String film;
- These introduce two variables called **X** and **film**.

- The computer is told that **X** holds an integer (a number) (**int**), and **film** holds a string of characters (**String**).
- These **declare** the variables **X** and **film**

Restrictions on Names

- There are some restrictions on what you can use as the name of a variable.
- They cannot start with a number for example.
- There are also some reserved "keywords" that already mean something else such as class and while.
- You cannot use a keyword as a variable (the computer would get confused!)

Exercise: What do the following do?

Give 2 example values that would be stored in each of the following declared variables:

```
String message;
int age;
char initial;
boolean finished;
```

Strings

- String values are in quotes.

  Eg "The Matrix"
- That is how you tell they are string values and not the names of variables like film
- An operation you can do on string values is to combine them using the operator +:
  "The Matrix" + " is cool!" creates the string "The Matrix is cool!"
  film + " is cool!" creates the string made of whatever is in variable film with the string " is cool!"

The Type: boolean

- We also have a type: boolean
- It has two possible values: true and false
- It is just like the other types: eg int, char, double, String.
- There are functions that return boolean.
- You can write your own functions to return booleans
- You can have boolean variables.

Assignment

- You store something in a variable by **assignment**
- You can assign the result of doing a calculation
  ```
  title = scanner.nextLine();
  ```
- stores the result of the method call in the variable called title.
  ```
  age = age+1;
  ```
- adds one to the value in variable age and stores it back in age.
  ```
  year = 2004 - age;
  ```
- sets the variable year to be 2004 minus the value in the variable age.

Putting values in Variables

- The commands:
  ```
  X = 1;
  film = "The Matrix";
  ```
- puts 1 in X and "The Matrix" in film
- "Box X **gets the number** 1"
- "Box film **gets the string** "The Matrix" "

All at once

- It’s bad manners (and dangerous) to leave variables around without values in them.
- So you will often want to declare a variable, and immediately store something in it.
- Most languages let you do this all at once:

```
int X = 1;
String film = "The Matrix";
```

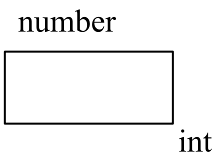
- These **declare** the variable and **intialise** it to contain an initial value.

Variables: the box metaphor

- You can think of a variable as a box in which data can be stored.

```
int number;
```

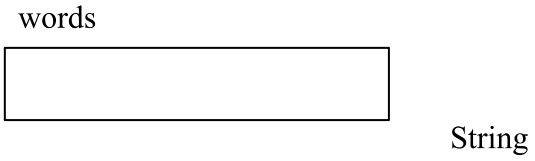
- creates a box called number which is the right size and shape to store integers (whole numbers):



Variables: the box metaphor

```
String words;
```

- creates a box called words which is the right size and shape to store a string: words
- String
- Since Strings can be any length, you have to think of a box for them as being a “stretchy” box



Other types in Java

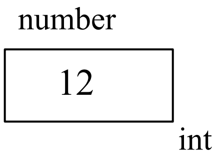
- Other things you can put in variables:
  - double: floating point numbers 3.14159
  - char: single characters ’a’, ’b’, ’c’, . . .
- Note that the character ’1’, is different to the integer 1 which is different to the floating point number 1.0
- They are different types, are stored differently and have different operations performed on them.

Assignment

- The operation for setting the contents of a variable is called **assignment**:

```
number = 3*4;
```

- puts 12 in the box number, obliterating whatever was there before:



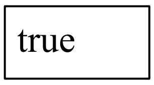
- The calculation on the right hand side is done first, then the value calculated stored

Boolean Assignment

```
boolean finished = true;
```

- creates a box called finished which is the right size and shape to store booleans and stores true in it:

finished



boolean

- We will see more on the use of booleans in the next lecture.

Assignment Copies Information

- You can copy information from one variable (box) to another:

```
        number = age;
```

- puts a copy of whatever was in the box age into box number , obliterating whatever was in number before but leaving age unchanged
- Suppose age held 42 the above would make number 42 too.

Assignment Exercise

- Draw before and after box diagrams that show what the following does assuming it started off holding the number 42

```
        number = number+1;
```

Integer Operations

- You can do the normal operations on integers as with a calculator.
- For example:

```
        f = (c * 9 / 5) + 32;
```

- multiplies c by 9, divides that by 5 then adds 32, putting the result in f.
- Arithmetic operations happen in the normal priority order (BODMAS)
- Eg multiplication before addition

Integer.parseInt

```
        int i = Integer.parseInt(textinput);
```

- This is a method that converts a String held in the variable textinput into an Integer (as you can't do calculations on strings!)
- Eg The string "123" of characters '1' then '2' then '3' is converted to the number 123 (one hundred and twenty three)
- Here the resulting integer is stored in variable i

Double.parseDouble

```
        double d = Double.parseDouble(textinput);
```

- Similarly this method converts a String held in the variable textinput into a double (ie a floating point / decimal number). Doubles give more accuracy than integers.

Expressions

- An expression is essentially just a fragment of a program that evaluates to give a value.
- The value it results in depends on the values of variables in it.
- If an
- For example.

  `(a + 1) *2` is an integer expression

  If a is 5 at the time it is executed it would evaluate to the value 12.
- Expressions are used in assignments to calculate the value to be assigned
- For example

```
        x = a + b + c;
```

- Here `a + b + c` is the expression as it is the part that evaluates to a value. It adds together the values of a, b and c. Only once the expression has been evaluated can the value calculated be stored in another variable by the assignment.

- The above examples are *integer expressions*. They evaluate to give an integer expression.
- You can have expressions of any type. For example,

  ```
  "Hello " + "World from" + name
  ```

  is a *string expression* as it evaluates to a string. If variable name holds Paul it would evaluate to the string `"Hello World from Paul"`
- Expressions can include method calls if the method computes a value. For example

  ```
  Integer.parseInt(textinput) + 25
  ```

is an integer expression that includes a call to the parseInt method. Whatever value it returns has 25 added to it to get the final value of the expression

Exercise

- Write a program that asks the user for 2 numbers and prints out their average (adding together and dividing by two).

Assignment

- General form:

  <variable> = <expression>;
- evaluates <expression>, (that is works out its value) and puts the result in <variable>, completely obliterating the previous contents.

Tracing: What does this do?

- Draw a series of box pictures with a box for each variable to work out what this code does...

  ```
  int x;
  int y;
  x=1;
  y=2;
  x=y;
  y=x;
  ```

Tracing: What does this do?

- Draw a series of box pictures to work out what this code does...

  ```
  int x;
  int y;
  int t;
  x=1;
  y=2;
  t=x;
  x=y;
  y=t;
  ```

Remember

Before you use a variable you need to do two things:

- Create the box (**declare the variable**):

  ```
  int count;
  ```
- Put something in the box(**initialise the variable**):

  ```
  count = 0;
  ```
- You can do both at the same time:

  ```
  int count = 0;
  ```

Some rules about variables

- Always call them by a name that tells you about their function: in the fahrenheit, celsius example a mathematician might have

  ```
  int f,c;
  . . .
  f = (c * 9 / 5) + 32;
  ```

- a computer scientist would more likely have
```
    int fahr,cel;
    . . .
    fahr = (cel * 9 / 5) + 32;
```
- Or better still write fahrenheit and celcius in full

Some rules about variables

- Always put your declarations in a sensible place, so you can find them later.
- For example:
  - do them in an initialisation phase for that part of the program.
  - Initialise variables immediately when possible.

Methods

- We previously saw how methods you define can return String values.
- They can return values of any other type in the same way.
- The header line (ie first line of the definition) for functions tells you what type of thing is returned, (and as we see later what types of things any arguments have to be).
```
public static int inputAge()
{
    int age;
    Scanner scanner = new Scanner(System.in);

    System.out.println("What's your age?");

    age = Integer.parseInt(scanner.nextLine());

    return age;
 }
```
- Whatever you put after the return statement (here age) must be the same type as stated in the header.
- You must then call the method in a context where a value of that type is needed
- If the method returns an int then it should be assigned to a variable of type int. eg for the method above
```
int yourage;
yourage = getAge();
```
- Functions and procedures have a lot in common, and Java treats them as the same kind of thing (a method): a procedure is a function that returns nothing ie has return type void.
```
    public static void askForFact()
```

Comments

- It is essential to add explanations ("comments") to programs explaining what they do and how they work at a high level.
- This is so that subsequent programmers including yourself can understand them.
- Comments are ignored by java - they are not computer instructions.
```
    /* This is a
       comment */
    // So is this
```

Testing your program a method at a time using a test method

- An advantage of using methods is that you can test them individually.
- An easy way to do this is to use the main method to call either a special **test method** or the actual method for the program:
- Comment it in or out depending on whether you want to run tests or the program itself
```
public static void main(String[] args)
{
    testmethod();

    // realprogram();
    System.exit(0);
 }
```

- The test method then runs a **test plan** - a series of calls to the methods that checks they work
- If your program was made of two methods called method1 and method2 (a function returning an integer), a simple test method would be something like:

```
public static void testmethod()
{
    method1();                    //check method 1 works
    int result = method2();      //check method 2 works
    System.out.println(result); //print its result to check
    int result = method2();
    System.out.println(result);
      // Call it several times
      // perhaps so you can give it different
    return;
}
```

Random Numbers

- Various Java libraries define a whole host of more complicated types than basic numbers and strings
- For example, suppose you want to create a virtual coin or dice that introduces randomness into your program, one way is to load the *Random* library

```
import java.util.Random;
```

- You can then create a variable that holds a random number generator.
  - o When you want one you ask it for a new random number like rolling a dice.
  - o One way to think of it is as an endless stream of random numbers where you can only ever see the next one in the sequence

```
Random dice = new Random();
```

- This declares a new variable we've called `dice` of type `Random`
  - o dice is now a thing with a special type `Random` that will give you random numbers.
- Now to get the next random number (throw the dice) you use the method nextInt. It takes an argument that gives the range of numbers involved

```
int dicethrow = dice.nextInt(6) + 1;
```

- Here `dice.nextInt(6)` says get me the next random number from dice . The argument 6 says make the number one of the 6 values 0,1,2,3,4 or 5.
- If we want an actual dice we want a number between 1 and 6 not 0 and 5 so we add 1 to the value.
- What we get is just an integer so we can store it (using an assignment) in an integer variable just like any other integer
  - o here we've called the integer variable `dicethrow`
- For example the following might be used to roll a 10 sided dice.

```
Random bigdice = new Random();
int diceroll = bigdice.nextInt(10) + 1;
System.out.println("You rolled " + diceroll +
                   " on a 10-sided dice");
```

Defining your own types: Record Types

- You can define your own new types from existing ones
- In particular you can create new records.

- They bind different values together in to a single combined value.
- We will see later how these can be packaged in a special way to create ‘abstract data types’ an extension of these simple records that is much more powerful.
- We will start simple though.

New class definitions

- So far we have put our programs inside a class definition

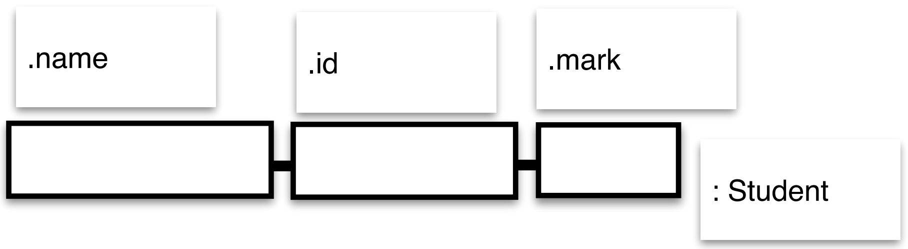
```
class myprog
{
   <the program code including main>
}
```

- We can create a new type by creating a new class outside the curly brackets of the main
- This creates records
- At its simplest, it is a way of packaging values together in to a single parcel that can be passed around.

```
class myprog
{
   <the program code including main>
}

class Student
{
   String name;
   String id;
   int mark;
}
```

- This creates a new type called Student that we can use inside the main class ie inside our program.
- You need a way to get at the separate values in a record.
- Here we invent name, id and mark as identifiers to do this.
- However, they are a different kind of thing to the variables we have seen so far, even though each line looks like a variable declaration.
- They are called ‘instance variables’ in Java but for now think of them as labels of internal compartments of a record.
- This class definition is not creating any variables itself.
- It is defining the blueprint for what a variable of type Student will look like.
- Linked variables are created when the program declares a variable of type Student.
- It says that it a variable of type Student will be a box, but one with a complicated structure. When you look inside the box you will find three compartments each a sub box. The first two are the right size and shape to hold Strings. The last one can only hold an integer.
- The three **fields** (compartments) are accessed by their names to get at the value inside:
- .name, .id and .mark



Defining a variable with type student

You define (ie create) variables of type Student with a variable definition like

```
        Student s = new Student();
```

- The first part creates a variable called s of type Student.
- The second part: new Student() creates the space to store records of this type and sets up the internal record structure with the fields as specified in the type definition.
- Once you have a variable that stores a record you access any field of it by giving the variable name AND the field name
    - eg s.name accesses the field called name of the record stored in variable s
- If used on the left hand side of an assignment this allows you to put a value in that box s.name = “Jo Bloggs”;
- This stores the string “Jo Bloggs” in field (ie compartment) name of variable s
- If used in an expression eg a print statement or right hand side of an assignment it accesses the value stored there System.out.println(s.name);
- This accesses field (ie compartment) name of variable s and prints the value found there

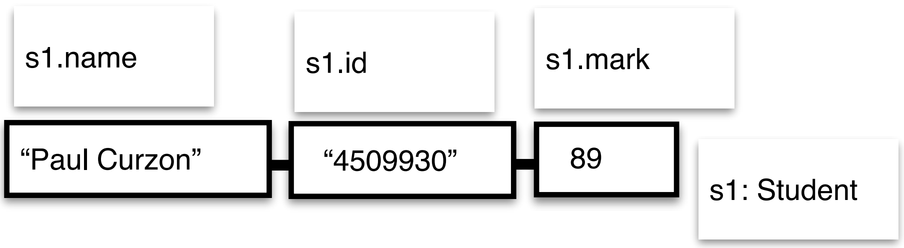
An example program

- For example, the following program uses the new type definition to create a single variable called s1 that has type Student.
- It then sets its value (which is made up of three parts).
- It then print them out.

```
class studentrecord
{
   public static void main(String []p)
   {
       Student s1 = new Student();

       s1.name = “Paul Curzon”;
       s1.id = “4509930”;
       s1.mark = 89;
       System.out.println(s1.id + “ “ + s1.name + “ “ + s1.mark);
   }
} //END class studentrecord

class Student
{
   String name;
   String id;
   int mark;
} // END class Student
```



A Java Program Template

All the programs you write will have the form below. The parts in bold will appear exactly as below. Parts starting //*** can be deleted as they just indicate what things need changing. Other parts not in bold need changing to suit the program (like giving it a sensible name. You do not need to type this in – you can download it from the ECS401 QM+ site.

```
/* ***************************************
   AUTHOR: Paul Curzon ***PUT YOUR NAME HERE
    ***WRITE A SHORT DESCRIPTION OF WHAT YOUR MODIFIED PROGRAM DOES HERE.

       This is not a just an outline of a program set up so you can fill
       in the gaps. Stuff in CAPITALS WITH STARS indicates what you need
       to change to create a program that work.  The rest is common to
       all programs you write.
   ****************************************/

class template // ***THE WORD AFTER CLASS MUST BE THE NAME OF THE FILE
               // ***(BEFORE the .java) YOU SAVE THIS IN
{
    /* ***************************************
     *    ***THE MAIN METHOD - WHERE THE PROGRAM STARTS

    public static void main (String[] param)
    {

         DOTHIS();  // ***REPLACE DOTHIS WITH THE NAME YOU USE BELOW
         NOWDOTHIS();  // ***REPLACE NOWDOTHIS WITH THE NAME YOU USE BELOW

         System.exit(0);

    } // END main

     /* ***************************************
     *    ***PUT A COMMENT HERE TO EXPLAIN WHAT THIS METHOD IS FOR
     */
    public static void DOTHIS ()   // ***REPLACE THE NAME 'DOTHIS'
                                   // ***WITH THE NAME YOU USED ABOVE
                                    // ***THE NAME SHOULD HELP THE READER
                                   // *** UNDERSTAND WHAT THIS CODE DOES
    {

        // *** DELETE THIS LINE. THE ACTUAL CODE GOES IN HERE
        // *** THESE ARE THE ACTUAL INSTRUCTIONS YOU WANT TO BE FOLLOWED
        return; // *** IT WILL END WITH A RETURN STATEMENT

    } // END DOTHIS
            // ****DONT FORGET TO CHANGE 'DOTHIS' HERE TO YOUR METHOD NAME

     /* ***************************************
     *    ***PUT A COMMENT HERE TO EXPLAIN WHAT THIS METHOD IS FOR
     */
    public static void NOWDOTHIS ()   // ***REPLACE THE NAME 'DOTHIS'
                                   // ***WITH THE NAME YOU USED ABOVE
                                    // ***THE NAME SHOULD HELP THE READER
                                   // *** UNDERSTAND WHAT THIS CODE DOES
    {

        // *** DELETE THIS LINE. THE ACTUAL CODE GOES IN HERE
       return; // *** IT WILL END WITH A RETURN STATEMENT

    } // END NOWDOTHIS
            // ****DONT FORGET TO CHANGE 'DOTHIS' HERE TO YOUR METHOD NAME

} // END class template
               // ***DONT FORGET TO CHANGE 'template' HERE
               // ***TO YOUR CLASS (IE PROGRAM) NAME FROM THE TOP
```

Unit 2 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the "week 1" unit. Do them before trying the next assessed exercise.

There are a series of full programs that you can compile and run on the ECS401 QM+ site – unit 2 area, **experiment with them modifying them to do slightly different things** until you understand how they work. Make them do longer calculations (add 4 numbers, do other operations like subtraction, etc). Programs there include:

- **add3.java** : Add a series of ages and print the result
- **add3input.java** : Add a series of ages user provides and print the result
- **add3input.java** : A better version of the above with methods
- **calculate.java** : Convert text to numbers and do more complex calculations
- **yearborn.java** : More calculations to do something a bit more interesting
- **daysminutes.java** : The program below calculating days into minutes
- **average3double.java:** Give an accurate rather than integer average using type double
- **studentrecord.java**: Create a new record type
- **simplerandom.java** : Roll a dice

Here is a typical simple program using expressions and assignment.

```
/* ****************************************
AUTHOR Paul Curzon
  Add the ages of three children (three integers)
   to work out the total age and average age
   Get the ages from the user.
***************************************** */

import java.util.Scanner; // Needed to make Scanner available

class add3input
{
    public static void main (String[] param)
    {
        add3ages();
        System.exit(0);
    } // END main

    /* ***************************************************
       This method adds the ages of three children
       getting the ages from the user
    */

    public static void add3ages()
    {
       Scanner scanner = new Scanner(System.in);
       int age1;  // each will hold the age of a different child
       int age2;
       int age3;
       int totalAge; // the answer when the three ages are added
       int averageAge; //their average rounded as an integer
       //  Get the ages of three people

       System.out.println("Give me an age");
       age1 = Integer.parseInt(scanner.nextLine());

       System.out.println("Give me an age");
       age2 = Integer.parseInt(scanner.nextLine());
```

```
        System.out.println("Give me an age");
        age3 = Integer.parseInt(scanner.nextLine());

        // Now do the calculation of the total age
        //
        totalAge = age1 + age2 + age3;
        averageAge = totalAge / 3;


        // Finally give the user the answer
        //
        System.out.println("The total age of the three children is " + totalAge);
        System.out.println("and their average age is " + averageAge);
        return;
    } // END add3ages

} // END class add3input
```

1. Number input
Write a program that reads in a whole number n, and prints it out again. Modify your code so that it prints out n+1 (ie one more than the number typed in).
2. Fahrenheit to Celsius
Write a program that reads a temperature in degrees Fahrenheit and prints out the corresponding temperature in degrees Celsius (C = (F - 32) * 5 / 9).
Make the program 'user friendly' (this applies to all subsequent programs you write - ever!). The messages to the user should be clear and easy to understand what is expected of them.
3. Celsius to Fahrenheit
Write a program that reads a temperature in degrees Celsius and prints out the corresponding temperature in degrees Fahrenheit (F = (C * 9 / 5) + 32).
4. Year Born (again)
Modify the year born program from above so that it asks the user for the current year first (so won't get the answer wrong in future.
5. Area of a rectangle
Write a program that reads the length and breadth of a rectangle in cm and computes its area.
6. Rectangle record
Write a program that creates a new record type Rectangle. Records of type Rectangle should hold two values: an integer length and an integer breadth. The program should ask the user for the length and breadth, store them in a variable of type Rectangle and then print the area of the rectangle.
7**. Compiler Messages**
Take your completed working program that calculates the area of a rectangle, and make a series of single changes that introduce errors (for example delete single lines, change the names of variables in one place only as for a typing error, delete brackets, etc). After each change made, compile the program, make a note of the error message and its cause, work out what the error message means, correct the program and recompile it to ensure it has been fixed, then insert the next error.
8**. Bonuses**
Write a program that given an amount of company profit, divides it equally between 5 employees of a firm as a bonus.
Does your program give an accurate answer? If not you may wish to consider how might this be done (HINT decimal numbers are a different type to integers - use type keyword double (for double length floating point number)) See the example programs.
9**. 12-sided dice**

Write a program that rolls a 12-sided dice printing the result of the roll.

Once you have done these, try the next assessed exercise. The assessed exercises must be done on your own. If you have problems with them then go back to the non-assessed exercises.

The following are additional, harder exercises. The more programs you write this week the easier you will find next week's work and the assessments and the quicker you will learn to program. If you complete these exercises easily, you may wish to read ahead and start thinking about next week's exercises.

9. Giving change

Write a program that works out the best way for a cashier to give change when receiving payment for some goods. There will be two inputs, the amount due and the amount of cash received. Calculate the difference and then the pounds, 50p's, 20p's, 10p's etc that the customer should receive in return. Hint: transform the difference into a whole number of pence. Then calculate the whole number of pounds due in the change. Subtract this from the original balance of change. Then calculate the number of 50p's needed...and so on.

10. Funny 6-sided dice

Write a program that rolls a 6-sided dice that has numbers 5, 6, 7, 8, 9, 10 on its faces, printing the result of the roll.

11. Odd 6-sided dice

Write a program that rolls a 6-sided dice that has odd numbers 1, 3,5,7,9,11 on its faces, printing the result of the roll.

Avoiding Bugs

It’s always better to avoid a bug in the first place, rather than cure it afterwards. That is where programming "style" comes in...Be a stylish programmer and make your programs easier to follow.

TIP 1: Always put all the variable declarations together at the start of the method - like an ingredient list in a recipe. It makes it easier to check you haven't missed anything. eg

```
int age1;  // each will hold the age of a different child
int age2;
int age3;
int totalAge; // the answer when the three ages are added
```

TIP 2: Write comments with variables that say what they will be used for (not just repeating their names and types - give information you can't tell just from the declarations themselves)

TIP 3: Use variable names that tell you what the variable does. A variable called t could do anything. A variable called temperature is presumably storing a temperature. It is even easier to work out what the variable called todaysTemperature will be used for! That way you won't get confused about your intentions for variables you declared.

TIP 4: Use blank lines to make it easier to see how related commands are grouped or not grouped - and comment each group of commands (not each line).

TIP 5: Step through the method you have written executing it (doing what each instruction says in your head or on paper) line by line as though you were the computer, checking it does what you expect.

TIP 6: If your program uses specific numbers (like 2006) then declare a final variable and set it to that number then use the variable elsewhere in the program in place of the number. final before a declaration just tells the compiler that this variable will not be assigned to in the program (so the compiler can tell you if it does because of a mistake by you.)

```
final int CURRENTYEAR = 2006;
    …
year = CURRENTYEAR - age;
```

There are two advantages to this. First the name used can tell you what the number means - like "its the current year" so avoiding confusion - perhaps it stands for something else. Secondly, if the number is used twice and you decide to change it, (eg to 2007) you only have to do the change in one place (less work and you won't miss any of the changes by mistake)

Catching Bugs

Everyone catches colds. Everyone who writes programs gets bugs. One of the skills of programming is to come up with ways that help you work out what you did wrong. Early in your programming career it will mainly be getting bits of the language wrong: misspelling things, getting the punctuation wrong,... Those things the compiler spots for you, but you have to learn how to work out what the compiler is telling you.

TIP 1: Double check that you have declared all the variables you have used at the start of each method. (eg by writing int age; if you used an integer called age)

TIP 2: Watch out for having mistyped 1 (one) instead of I (the letter i) or 0 (zero) instead of O (the letter O).

TIP 3: If you are using an editor that knows about Java it may use colours to emphasise part of the program - strings in one colour, variables in another etc. Use the colours to see where things have gone wrong. If the last part of the program has all turned pink it might mean for example you forgot a closing string quote for the string where the pinkness starts!

Some Common Compile-time errors

Undeclared Variables

```
pc$ javac add3.java
add3.java:52: cannot find symbol
symbol  : variable totalAge
location: class add3
    totalAge = age1 + age2 + age3;
    ^
```

Oops I forgot to declare the variable totalAge - it is telling me it "cannot find symbol" - that just means there is a name I'm using that it doesn't have a clue what I am talking about. It is guessing I meant it to be a variable but it can't be sure what type (is it a String or an Integer). I have to tell it by adding earlier the line: int totalAge;

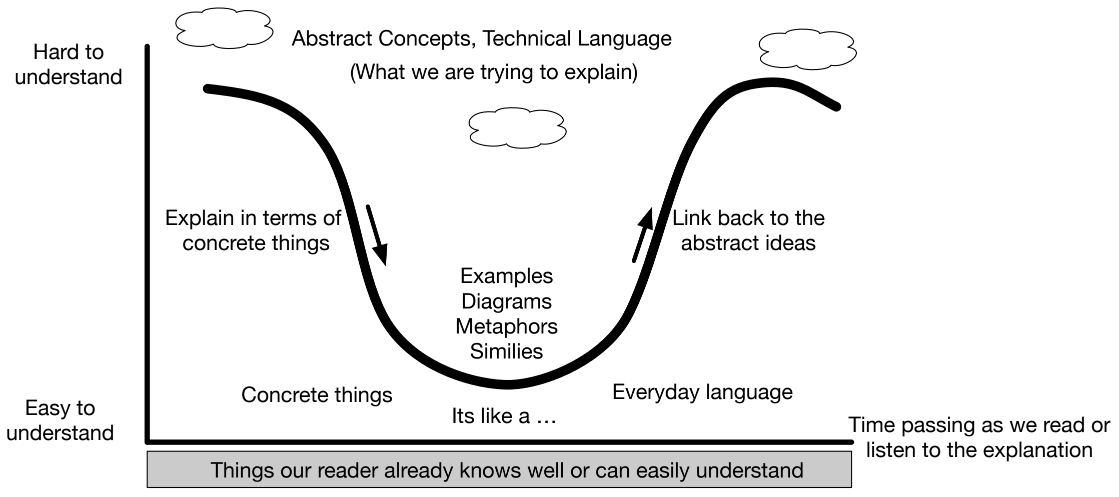
The same kind of message will be given if I had misspelled a variable I had declared (including putting in capitals by mistake) or just misremembered what I had called it.

```
pc$ javac yearborn.java
yearborn.java:61: ';' expected
            year + " or " (year - 1);
                         ^
```

This is an example where the compiler is confused. It thinks I've missed a semicolon when actually I've missed a +. It could just tell it needed something other than a bracket after quotes as a bracket made no sense: year + " or " + (year - 1);

Explaining concepts

A good explanation follows a wave pattern with three steps. It starts with the abstract, complex concepts (using the technical words) to be explained, then switches to more everyday language (words that people already understand) and concrete examples, diagrams, etc. Finally it links this everyday language and concrete things back to the technical concepts.



Hard to understand
Abstract Concepts, Technical Language
(What we are trying to explain)
Explain in terms of concrete things
Link back to the abstract ideas
Examples
Diagrams
Metaphors
Similies
Concrete things
Everyday language
Its like a …
Easy to understand
Time passing as we read or listen to the explanation
Things our reader already knows well or can easily understand

Here is a short example:

> *Types are a way that programming languages group values in to ones that have the same memory properties and operations applied to them.*
>
> *For example, think of the group of numbers like 1, 2,3 …,777, -5 etc that are all whole numbers that can be added, subtracted and so on.*
>
> *So all these whole numbers are all grouped as values of the type of integer (int in Java), stored in the same way in a single word of memory. They allows operations of +, (integer addition) - (integer subtraction) and so on.*

Highlight the technical terms and sentences in one colour and those that give examples and use ideas you were already familiar with n another.

You should by now be able to explain lots more concepts. Here are some questions to check. If you can't write detailed answers then do some more background reading but focus on being able to understand. Working out an answer for yourself will help you understand and so remember far better.
Make sure you can write answers in your own words without a book or computer in front of you. Remember there is no one right answer. Every ones answers should be different.

Explain what is meant by the following concepts, using Java programming examples to illustrate your answer:

a type	a String	assignment	a name	a boolean	an integer variable
a string variable		an integer assignment	a function	a procedure	an expression
an integer expression		a string expression	a boolean expression	a record	a field

You should be able to explain some of the concepts from week 1 better now too.

Concept maps: Draw concept maps that link the concepts you have come across this week.

Compare and contrasting concepts or code

Exam questions test your understanding in different ways and different kinds of answers are expected depending on the questions asked. So far we've looked at "**Explain**" questions. You can also be asked to **"Compare and contrast"** concepts (or just compare or just contrast them). A compare and contrast question is asking you to do more than just explain the concepts but to actively draw out the similarities and the differences.

For example you might be asked a question like:

1. *Compare and contrast variables and values.*

Notice how the following answer not only explains the concepts but also points out the similarities and differences. Bold phrases are used to either show a similarity or difference. Notice how concrete examples are given to illustrate the similarities and differences.

*Variables and values **are both** elements of programs. A variable is a location where data can be stored. It is has a name and a type. A value **on the other hand** refers to an actual piece of data. They are the things that can be stored in a variable. Values **also** have types **though** they do not have names. For example, in the code fragment*
n = 55;
n is a variable and 55 is a value. This particular instruction when executed stores the value 55 in the variable called n.

***Like** variables, values have types. For example in the above fragment **both** n and 55 are of type integer. For a variable the type refers to the kind of thing it can hold, **whereas** for a value it refers to the kind of thing it is. In Java **both** variables **and** values can be various types including integers, Strings and booleans. The only boolean values are true and false. String values are given in quotes. They superficially look like the names of variables and so are easily confused. **Whereas** s is a variable (no quotes), "s" is a String value and 's' is a character value. The assignment*
s = "s";
stores the string value consisting of a single character – the letter 's' into the variable with name s.

*Variables have to be declared in a program before they can be use **whereas** there is no equivalent of declaration for a (simple at least) value. For example before the above fragment, n would be declared – ie given its name and type as follows:*
int n;

*Variables **and** values are the basic building blocks of expressions. They are simple expressions themselves but can be built into more complex expressions using operators like + or *. For example*
*n +75 – k *2*
is an expression that is built from the variables n and k and the values 75 and 2.

Unit 2 Exam Style Question

Attempt the following questions in exam conditions. A model answer can be found on the ECS401 website. Compare your answer with the model answer. How good are your explanations? How good is your code?

Question 1 [25 marks]
This question concerns assignment and expressions

a. [6 marks]
State whether each of the following are good or bad choices for variable names, **justifying** your answer.
i) t
ii) class
iii) hair_colour
iv) hairColour1
v) 4cast
vi) timeOfDay?

b. [9 marks]
Compare and Contrast the following fragments of code
i)

```
t = a;
a = b;
b = t;
```

ii)

```
c = d;
d = c;
```

c. [10 marks]
Write a Java program that converts a length of time given in days to a length of time in minutes. For example 1 day would be converted to 1440 minutes (as 1x60x24=1440).

Unit 3: Making Decisions

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- read and write programs that make decisions
- explain the use of boolean expressions
- explain what is meant by if statements
- trace / dry run programs containing if statements
- read & write simple procedures containing if statements
- read & write methods that take arguments and return results
- read & write programs that use accessor methods
- explain concepts related to and the use of methods

From last week …

You should now be able to:

- read & write and run simple programs that manipulate data
- read & write programs that do calculations on values stored
- read & write programs that print out messages that include the results of calculations.
- read & write programs that store and manipulate different types of values.
- explain how your above programs work and the concepts involved.

Do the Interactive Notebooks

This unit comes with a series of interactive exercises in interactive notebooks. A link is in the. You may wish to do them before reading through these notes. Whether before or after, you MUST work through those exercises. These include

- **Interactive Notebook: Unit 3-1 If Statements**
- **Interactive Notebook: Unit 3-2 Boolean Connectives**

Find the notebooks here: **https://jhub.eecs.qmul.ac.uk**

Non-technical Reading

The following booklets may help you understand concepts if struggling. They are available from the activities and reading section of QM+.

- **Chapter: 5 of Computing without Computers**

Deeper Reading

The following booklets go into the topics in this section in more detail. Read them if you find the concepts hard to understand. You can download them from the QM+ site.

- **Booklet: Creating and Using Records**

Lecture Slides/Notes

A method from last week

You should be able to make methods like this one by now:

```
public static void average3ages()
    {
       int age1 = 7;
       int age2 = 13;
       int age3 = 10;
       int averageAge;
       String textinput;

       averageAge = (age1 + age2 + age3) / 3;

       System.out.println(
            "The average age of the three people is "
                  + averageAge);
       return;
    } // END average3ages
```

Decisions, Decisions

- Straight line programming isn't enough. You have to be able to do different things in different circumstances.

Examples:

Q: Which course do I take?

A: if you are a G400 student, then Computers and Society, but if you are a G452 student, then Intro to Multimedia, and if you are a . . .

- Or imagine a cash machine: "Do you want cash or the balance?"
- ...or your mobile phone with all those menus of options.
- …or the Noughts and Crosses AI that was all about decisions
- When you made a decision on what to type in the program has to make a decision on how to act in response…differently depending on what you did.

If statements

- Programming languages solve this by having **if statements**:
- We need a way in the language to spell out a decision…we need…
- Something that says what follows is a decision - we need a new keyword
- A way of asking a question - is this true or false?
- A way of giving the two alternative sequences of commands (mini-programs) depending on the answer.

if statement form

```
if ( <test> )
{
  <statement>
}
else
{
  <statement>
}
```

Example

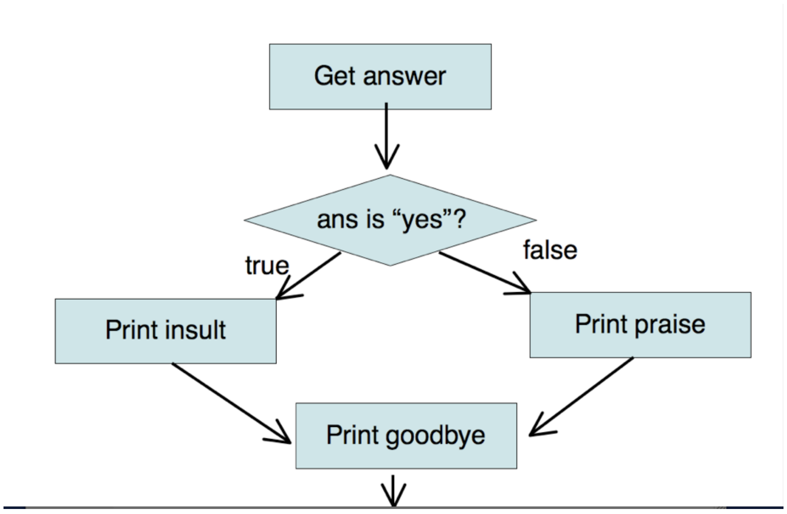
```
Scanner scanner = new Scanner(System.in);
String ans;
```

```
System.out.println("Can I insult you, please?");
ans = scanner.nextLine();

if (ans.equals("yes"))
{
    System.out.println("You smell!");
}
else
{
    System.out.println("You smell of roses!");
}

System.out.println("Thank you!");
```

Flow of Control



Exercise

- Write a method that asks the user to type in their name.
- If they are called Paul print "That's the name of a true genius".
- If it's not just print "Never mind".

Doing more than one thing

```
if (ans.equals("y"))
{
  System.out.println("You smell!");
  System.out.println("I mean it!");
}
else
{
  System.out.println("You smell of roses!");
  System.out.println("Wow!!");
}
System.out.println("Thank you!");
```

Blocks

- Simple statements can be grouped into a block using curly brackets

```
{
   System.out.println("You smell!");
   System.out.println("I mean it!");
}
```

- You can put as many statements as you like in a block and they will get executed together in sequence
- You can even put if statements inside a block

You can combine tests

```
if (ans.equals("yes”))
{
  System.out.println("You smell!");
}
else if (ans.equals("no”))
{
  System.out.println("You smell of roses!");
}
else
{
  System.out.println("yes or no!");
}
```

- You can easily build these up to the point where your code is very opaque: Don’t!
- Always adding final “catch all” else cases is a simple but important part of **input validation** - catching unexpected situations in the input provided - and so **defensive programming**.

Exercise

- Write a method that asks the user to type in the kind of pet they own if any. Print out a suitable message like “Ahh, a dog they keep you fit” or “Goldfish look after themselves” etc depending on the kind of pet they reply. If the program doesn’t recognize the name. Print “Not heard of that one”.

The test

- The test has to be something that has two answers - one for each possible follow-on command
- The answer can either by true or it can be false
- Integer expressions like n+1 evaluate to numbers
- For tests we need **boolean expressions** - that evaluate to booleans (true or false)
- ans is equal to “yes” ….TRUE

Some example boolean expressions

```
letter == ′y′
```

- letter is equal to ‘y’ ? True or false?

```
X <= 34
```

- X is less than or equal to 34 ?

```
X*Y > 0
```

- X times Y is greater than 0 ?

```
ans.equals(″yes")
```

- ans is equal to the string “yes” ?
- They all are either true or false depending on the values in the variables

The Type: boolean

- We have a type: boolean for true or false things!
- It has two elements: true and false
- It is just like the other types(int, char, double, String).
- There are functions that return boolean.
- You can have boolean variables.
- So you can assign to them using boolean expressions just like with integer variables and integer expressions

Constructing boolean expressions

standard comparison operators:

- equal: ==,

- not equal: !=,
- less than: <,
- greater than: >,
- less than or equal: <=,
- greater than or equal: >=

Exercise

- Assume x is 0 and y is 1, which of these boolean expressions evaluate to true and which to false?
```
      x > y
      y < 1
      y <= x+1
      x == y-1
```

Beware .equals() = and ==

- USE == (for comparing ints and chars)
- USE .equals() (for comparing strings)
```
      pet.equals("dog")
```

- DO NOT USE = (assignment)
- In JAVA it is a command saying move data from one place to another NOT a question with a true false answer
- Read = as "gets the value of " NOT "equals".

Constructing boolean expressions

- combinations through boolean operators (called **connectives**):
- and: & or &&
- or: | or ||
- not: !
- (cf. Logic and Proof course)

Examples:
```
ans=='y' | ans =='Y'
X>0 & Y>0
```

Equals for strings

- Java has two types of objects, simple and complicated (not technically called that).
- = = only works for simple objects. For complicated objects they only tell you whether the objects are represented internally in Java by the same bit of memory.
- Strings are complicated. This means:

  USE
```
      mystring.equals("Hello World!")
```
  DON'T USE
```
      mystring == "Hello World!"
```

Tracing if statements

- You can trace if statements using boxes as variables just as with straight line code.
- As you trace you just make sure that you jump to the correct lines of code according to the test (following the flowchart)
- Just trace those lines missing the ones on the branch of the if not executed.
- It's a good idea to have extra question boxes where you write the answers to the tests.

Tracing If statements

- Trace the following code 3 times for values input of 3, 10 and11.
```
int price = 0;
```

```
String weighttext;
int weight;

System.out.println("Weight?");
weighttext = scanner.nextLine();
weight = Integer.parseInt(weighttext);

if (weight <= 5)
    {price = 2;}
else if (weight <= 10)
    {price = 4;}
else
    {price = 6;}
System.out.println("The cost is " + price);
```

Systematic testing

- You're writing programs whose behaviour is more complicated now.
- Systematic testing is **very important**.
- In industry more effort goes into testing code than writing it.
- That is why we've put you into test pairs in the labs
- It is impossible to test programs exhaustively (every possible input), so "representative" cases have to be picked.

Systematic testing

- The idea is to test each possible program **behaviour**, not each possible **input**.
- For complex program behaviours there are still lots of cases, and the testing has to be done on an automated test-bed.
- Complex programs have to be broken down into bits, and each bit tested separately. Microsoft employs **more** people to **test** code, than to **write** it.
- You have to take testing seriously too.
- There is more advice on the website on how to test.

Programming style and layout

- The point of having a high-level language is that it can be read and understood by people.
- This point is lost if the code is written down in a way that is hard for people to understand.
- Most industrial coding is maintenance. This means *you* will have to read, understand and change code written by other people.
- *Worse:* This means *you* will have to read, understand and change code written by *yourself*!
- You will need all the help you can get!

Programming style and layout, cont

- There are some simple rules you can follow to make your program more easily readable.
- See the web and example programs and copy their style.
- These boil down to:
- choose informative names for variables (and later other "things")
- use layout to make the grammatical structure of the program clear (indent at **{** go back in at **}**)
- **Explain** what blocks of your code do with useful comments

Further Extensions

- If you really want to be a good programmer, once you have mastered if statements explore the variations
    - Switch statements
    - Assigning boolean expressions to boolean variables

Random Numbers

Combining random numbers with if –then-else we print results other than numbers

```
      Random coin = new Random();
      int cointoss = coin.nextInt(2);
      if (cointoss == 0)
          System.out.println("You tossed Heads");
      else
          System.out.println("You tossed Tails");
```

Defining methods – procedures

Defining methods

- One of the most important things to start to do is split your programs into methods.
- As programs get larger it becomes harder to read them, find mistakes and to change.
- It is better to split them into small bits
- Each method should do a clear self-contained job (a bit like a recipe in a recipe book)
- You have been writing methods from the start.

An example from earlier

Look at the following code that we were playing around with fragments of earlier:

- It uses two methods – main and insultme.
- methods like this that just do things are called procedures (more on that later)
- insultme is getting a little complicated
- we have also changed the messages several times in our different variations earlier
- can we make it easier to see what it does overall
- make it so that we can change parts

```
import java.util.*;

class insultornot
{
    public static void main (String[] param)
    {
        insultme();
        System.exit(0);
    } // END main

/* ********************************
    A method that asks to insult you
*/
    public static void insultme()
    {
      Scanner scanner = new Scanner(System.in);
      String ans;
      System.out.println("Can I insult you, please?");
      ans = scanner.nextLine();

      if (ans.equals("y"))
      {
        System.out.println("You smell!");
        System.out.println("I mean it!");
      }
      else
      {
        System.out.println(”You smell of roses!");
        System.out.println(”Wow!!");
      }

      System.out.println(”Bye!");
      return;
    } // END insultme
} // END class insultornot
```

We can take out chunks of code that print a message replacing them by calls to new methods

```
      if (ans.equals(“y”))
      { printInsultingMessage();
      }
      else
      { printNiceMessage();
      }
```

- We have just introduced two new methods called `printInsultingMessage` and `printInsultingMessage`
- Each is intended to do a clear well-defined task.
- We can see from the above cascade what the fragment is intended to do – either print an insulting or nice message
    - o without seeing the details of the new methods
- In the above fragment we **call** the new methods.
    - o the program says go and find some code with that name and when you've finished come back and carry on.
- We don't need to worry what they do precisely while writing this bit
- We then need to define exactly what they do do – what message does each print.
- Here is what we might write for the insulting message

```
/* ********************************
    Print an insulting message
*/
    public static void printInsultingMessage()
    {
        JOptionPane.showMessageDialog (null,”You smell!");
        JOptionPane.showMessageDialog (null,”I mean it!”);
        return;
    } // END printInsultingMessage
```

- It is a self contained method.
- We have given it a name in the first line: `printInsultingMessage`
- We have also now said what it does precisely
- Someone who does want to see the detail of the insulting message just looks here.

return

- Notice that the last instruction is return.
- That says “Now we have finished here, go back to where you came from.”
- All methods should execute a return statement as the last thing they do
- main is an exception as it uses System.exit for a similar purpose.
    - o it says quit the program and return to the operating system
    - o rather than just quit this method

All together

- The new program with it all put together is below
- It consists of 4 methods and the program jumps around between them:
- The program starts executing main. It says go execute insultme
- It jumps to insultme which asks for the person to make a choice.
- If they want to be insulted the program jumps to printInsultingMessage & starts to follow the commands there. When done it returns back to insultme. It carries on where it left of.
- insultme next prints Bye and then hits its return statement
- so it jumps back to main which carries on where it left off.
- The next thing its instructions say to do is exit so it does
- What would happen if the person asked not to be insulted? Which methods would be called then?

```
import java.util.*;
class insultornot
{
    public static void main (String[] param)
    {
        insultme();
        System.exit(0);
    } // END main

/* *********************************
    A method that asks to insult you
*/
    public static void insultme()
    {
      String ans;
      Scanner scanner = new Scanner(System.in);

      System.out.println(Can I insult you, please?);
      ans = scanner.nextLine();

      if (ans.equals(“y”))
      {
         printInsultingMessage();
      }
      else
      {
         printNiceMessage();
      }

      System.out.println(”Bye!");
      return;
    } // END insultme

/* *********************************
    Print an insulting message
*/
    public static void printInsultingMessage()
    {
        System.out.println(”You smell!");
        System.out.println(”I mean it!");

        return;
    } // END printInsultingMessage

/* *********************************
    Print a nicemessage
*/
    public static void printNiceMessage()
    {
        System.out.println(”You smell of roses!");
        System.out.println(”Wow!!");

        return;
    } // END printNiceMessage

} // END class insultornot
```

Always split your programs into methods!

Methods that return results

- Procedures just do something like print messages and then returned.
- Their purpose is not to compute a result to be used elsewhere.
- That is for example what is done by existing methods like
  `JOptionPane.showInputDialog`
  `scanner.nextLine`
- It is used in the following way

```
textinput = JOptionPane.showInputDialog("Give me a value");
ans = scanner.nextLine();
```

- The method doesn't just print something to the screen and let you type stuff in, it delivers that value typed back to the program and allows it to store it in a variable.
- What if we want to do something like that.

Example: Averages

- Suppose for example that we want to repeatedly get the ages of husbands and wives from the user and take their average.
- We might sensibly decide to split it off as a method.
- The code to do this might be:

```
 int averageAge = calculateAverage();
 System.out.println(i + ": " + averageAge);
```

- We have invented the method `calculateAverage`
- We need to define it
- It is easy to define but we need a way to pass the answer we have calculated back.
- That is easy to we just give the thing to return after the keyword return

```
public static int calculateAverage()
{
   int value1; int value2; int average;
   Scanner scanner = new Scanner(System.in);
   String textinput;

   System.out.println("Give me the first");
   textinput = scanner.nextLine();
   value1 = Integer.parseInt(textinput);
   System.out.println("Give me the second");
   textinput = scanner.nextLine();
   value2 = Integer.parseInt(textinput);

   average = (value1 + value2) / 2;

   return average;
}
```

- Notice though that we have also changed the word void we've been putting in the header line.
- It is now **int**
- That says that this method returns an integer value
- Put different types there if you want to return Strings, floats, etc
- It must match the type of the thing after the keyword return
- If there is no value returned we use void as that means return nothing
- That's why we've used it so far as our methods have not been returning values.

Defining methods – arguments

Passing arguments

- So far our methods are really restricted in what they can do.
- We really want them to be able to do different things each time we call them
- For example it would be nice to have a method that dealt with all the details of inputing numbers:
- Writing the following out over and over with minor variations is a pain!

```
Scanner scanner = new Scanner(System.in);
String textinput;

System.out.println("Give me the first");
textinput = scanner.nextLine();
value = Integer.parseInt(textinput);
```

- The trouble is we want to print different messages each time.
- We need to pass the information to the method when we call it.
- This is a simple form of generalisation - making the same code written once work for multiple situations.
- The information (the message to print) then needs to be stored
    - o we use a variable to do that
    - o we declare it in the header line of the method
    - o it is automatically initialised when the method is called
- Suppose we call the variable to store the information **message** we get the method definition:

```
public static int inputInt(String message)
{
   Scanner scanner = new Scanner(System.in);
   String textinput;
   int value;

   System.out.println(message);
   textinput = scanner.nextLine();
   value = Integer.parseInt(textinput);

   return value;
}
```

- We call it as before but now putting the information to pass (and so print) in the brackets of the call:
  `result = inputInt("Give me the first");`

By creating a method to Input ints we can also improve it, and that single improvement will apply everywhere it is used - we don't need to make the change in lots of places just in the inputInt method body.
For example: What if the user types something that isn't a digit? The program crashes. Later you will see how to check if something is a number to prevent this, and so could write a better inputInt method. This is another example of **input validation.**

- Our CalculateAverage method can now use this new one (as can any other method needing to get an integer from the user).
- It makes it much easier to write and understand
- The information passed are called arguments of the method.

```
public static int calculateAverage()
{
   int value1; int value2; int average;

   value1 = inputInt("Give me the first");
   value2 = inputInt("Give me the second");

   average = (value1 + value2) / 2;
   return average;
}
```

- We could do a similar thing to create a method called average2 say that just does the average calculation.
- It takes two pieces of information:
    - the two values to average
- The variables are again declared in the header line
    - to show they will store the information passed

```
public static int average2(int value1, int value2)
{
   int average;

   average = (value1 + value2) / 2;

   return average;
}
```

- We can now call this method to do the actual calculation.

```
public static int calculateAverage()
{
   int value1; int value2; int average;

   value1 = inputInt("Give me the first");
   value2 = inputInt("Give me the second");

   average = average2(value1, value2);

   return average;
}
```

More on Records

Accessing the fields of records through accessor methods

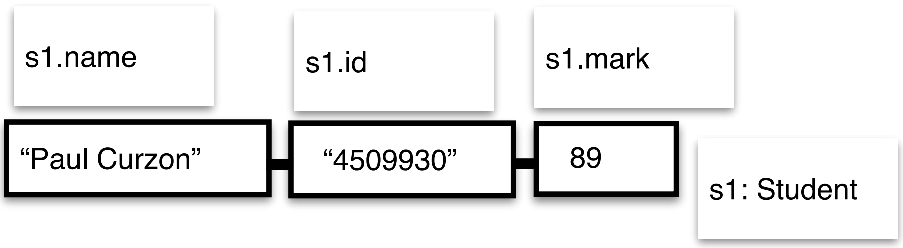
Accessor Methods

- It is good practice to access record fields ONLY using methods you define for the purpose, not directly.
- We will improve a variation of the earlier program below to illustrate:

```
class studentrecord2
{
   public static void main(String []p)
   {
       Student s1 = new Student();
       int mark1 = 40;
       int mark2 = 49;

       s1.name = "Paul Curzon";
       s1.id = ″4509930″;
       s1.mark = mark1 + mark2;
       System.out.println(s1.id + " " + s1.name + " " + s1.mark);
   }
} //END class studentrecord2

class Student
{
   String name;
   String id;
   int mark;
} // END class Student
```



Accessor Methods that extract information

- We need to define one method for each field in the record to allow us to **get** its value.
- We will first show a simple way of doing it that is possible in most languages.
- Later we will see a better way that Java allows you to do.
- We write methods with names of the form getFIELDNAME for each field of the record type.
- For the above record we would define the method getName as follows

```
public static String getName (Student s)
{
  return s.name;
}
```

- It takes a Student record as argument and extracts the value from the name field this value is of type String as that field is a string. It returns that String value: the name stored in the record.
- If we have defined this method in our program we can now use it to access the name field of any particular Student eg

```
System.out.println(getName(s1));
```

- This uses the getName method to get a name and then prints it.
- We define similar methods for each field:

```
public static String getId (Student s)
{
  return s.id;
}

public static int getMark (Student s)
{
  return s.mark;
}
```

- With these get accessor methods we can now write the main method above as follows:

```
public static void main(String []p)
{
     Student s1 = new Student();

     s1.name = "Paul Curzon";
     s1.id = "4509930";
     s1.mark = 89;
     System.out.println(getID(s1) + " " +
                        getName(s1) + " " +
                        getMark(s1));
}
```

- At first sight its not obvious why this is better, but as we will see later it makes a program much easier to change.
- We are using methods as a form of **abstraction:** hiding the detail of how the record is implemented.
- It also is the first step towards much more powerful ways of programming.
- With the detail hidden by the methods we don't need to think or know the detail, as long as we have the methods. We will see an example below

Accessor Methods that store information

- First, let's see a simple way to do a similar thing for setting the values in the record.
- We write one method for each field again.
- This time the methods have the name of the form setFIELD
- The method takes the record (s) as an argument but also the new value to put in it

- We return the newly modified record.

```
public static Student setName (Student s, String studentname)
{
  s.name = studentname;
  return s;
}
```

- We can do a similar thing for the other fields

```
public static Student setId (Student s, String studentid)
{
  s.id = studentid;
  return s;
}
```

- The total mark for each student stored in the record has been made up of two marks (coursework and exam).
- So we make its accessor set method take the two and calculate the total.
- . If we always use that method, we can be sure the total is always added the same way.
- Thats an advantage of using a method then used everywhere rather than do the addition in different places in the program

```
public static Student setMark2 (Student s, int smark1, int smark2)
{
     s.mark = smark1 + smark2;
     return s;
}
```

- Now by using these methods our main method becomes:

```
   public static void main(String []p)
   {
     Student s1 = new Student();

     s1 = setName(s1, "Paul Curzon");
     s1 = setId(s1, "4509930");
     s1 = setMark(s1, 40, 49);
     System.out.println(getID(s1) + " " + getName(s1) + " " + getMark(s1));
   }
```

- There is nothing now in the main method that says exactly what the internal structure of our Student record actually is. That is all specified in the accessor methods.
- The program as a whole is

```
class myprog
{
   public static void main(String []p)
   {
       Student s1 = new Student();
       int mark1 = 40;
       int mark2 = 49;

       s1 = setName(s1, "Paul Curzon");
       s1 = setId(s1, "4509930"");
       s1 = setMark(s1, mark1, mark2);
       System.out.println(getID(s1) + " " +
                          getName(s1) + " " +
                          getMark(s1));
   }

   // Get methods for Student record type
   public static String getName (Student s)
   {
     return s.name;
   }

   public static String getId (Student s)
   {
     return s.id;
   }
   public static int getMark (Student s)
   {
     return s.mark;
   }

   // Set methods for Student record type
   public static Student setId (Student s, String studentid)
   {
     s.id = studentid;
     return s;
   }

   public static Student setName (Student s, String studentname)
   {
     s.name = studentname;
     return s;
   }

   public static Student setMark2 (Student s, int smark1, int smark2)
   {
     s.mark = smark1 + smark2;
     return s;
   }

} //END class myprog

class Student
{
   String name;
   String id;
   int mark;
} // END class Student
```

Abstraction helps

- Now suppose we realise the program as a whole will need us to store both marks for each student not just the total. (Perhaps we might later need to add functionality to tell students both marks.)
- Because of our use of methods & abstraction we can make changes to a small part of the code
- We first add new fields to the record

```
class Student
{
   String name;
   String id;
   int mark1;
   int mark2
} // END class Student
```

- We then change the mark get method to do the addition

```
public static int getMark (Student s)
   {
     return (s.mark1 + s.mark2);
   }
```

- We change the mark accessor set method to just store the two marks

```
public static Student setMark2 (Student s, int smark1, int smark2)
{
     s.mark1 = smark1;
     s.mark2 = smark2;
     return s;
}
```

- The key thing is we don’t need to change any other part of the program we call the same methods.
- Here the main method does not need to change
- Now image the whole program is a full student record program a million lines long .
- The change to the Student type still only needs changes to the accessor methods
- The rest still works.

Unit 3 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the previous unit.

1. If statements, starting to write methods, accessor methods.
There are a series of full programs that you can compile and run on the ECS401 QM+ site – unit 3 area, download, compile and run them, then **experiment with them modifying them to do slightly different things** until you understand how they work.

Make them ask for different information, make different decisions, or do different things depending on the result. Change the question to be the opposite of it (eg ask if the two things are not equal rather than equal and see what happens). Programs there include:

- **if_boolean.java** : A boolean test in an if switches between commands to execute
- **if_String.java** : You can do tests on strings Change the assignment to ans to see the effect. Note == is not used to compare strings
- **if_int.java** : You can do tests on numbers What happens if you type a number other than 1 or 2 when testing this program
- **if_connectives.java** : Explore how to write ifs when you want one of many or all conditions to be true to take an action..
- **if_char.java** : You can chain a whole series of tests together if there are lots of choices. What happens if you type in a full word instead of a character when this program asks?
- **if_char2.java** : You can put any command inside if statements: here assignments
- **insultornot.java** : You can start to split your programs into methods
- **moreinsults.java**: boolean variables, if-then-else staircase, sequenced ifs.
- **studentgettersetter.java** : using accessor methods to access a student record
- **studentgettersetter2.java** : accessor methods make changes easy localising what needs to be done

2. Question and response
Write a program that asks the user
How many beans make five?
If the user responds 5, print
Correct!
otherwise print
Wrong!
Now change the program so that it prints *About right!* if the user responds 4 or 6

3. Maximum of two integers
Write a program that asks the user to input two integers and then prints out the larger of the two.

4. Maximum of three integers
Write a program that asks the user to input three integers and then prints out the largest of the three.

5. Pass or fail
The Queen Mary grading scheme for BSc course units awards a letter grade based on the numerical score between 0 and 100 as follows:

less than 40	: F
40 to 44	: E
45 to 49	: D
50 to 59	: C

```
60 to 69      : B
70 and above  : A
```

Write a program that inputs a numerical score from the keyboard and outputs the appropriate letter grade.

6. Postage

Write a program that computes the cost of posting a parcel.
The cost is calculated using the following rates:

```
first kilogram......................£6
each additional kilo up to 5kg.......4
each additional kilo up to 10kg......3
each additional kilo.................2
```

There is a maximum weight limit of 20kg. The program should print a prompt:

```
Type Weight (kg):
```

When you have typed a weight, say 4, the program should respond with:

```
The cost is £18
```

unless the weight is over 20kg, in which case it should respond:

```
There is a maximum weight limit of 20kg.
```

7. The planets

Write a program that asks the user to input the name of a planet (Mercury, Venus, Mars etc.) and outputs its place in order from the sun.
So, if the user correctly gives the name of a planet (e.g. Mars) then the program should respond with

```
Yes. Mars is the 4th planet from the sun.
```

If the user does not respond with the name of a planet (e.g. Sun) then the program should reject this as follows:

```
No. Sun is not a planet.
```

The program will input the user's answer and store it in a String variable as follows:

```
String planet = JOptionPane.showInputDialog ("Give me the name of a planet");
```

If the user types Mars (and then presses return) the above method invocation (or call) has the same effect as

```
String planet = "Mars";
```

Hint: The program should first check whether the answer is "Mercury" (which is the planet closest to the sun):

```
if (planet.equals("Mercury"))
    System.out.println("Yes, Mercury is the 1st planet from the sun.");
else . . .
```

Once you have done these, try the next assessed exercise. The assessed exercises must be done on your own. If you have problems with them then go back to the non-assessed exercises.

8. Using methods

Redo your answers to the above using methods for subtasks that return results.

9. Modified Student records

Modify the program studentgettersetter2.java to print the coursework and exam marks as well as the total by writing extra getter methods.

10. Planet records

Create a new type of planet as a record type. A planet has a name, a distance from the sun and a position in the solar system. Store in a variable Earth the details of the Earth (3rd from the sun, 150 million distance km). Print those details out. Use accessor methods to access the record fields.

10. Planet records

Redo the if-then-else planets question above but now using your record type to store information about each planet that includes its distance from the sun. Use accessor methods to access the fields of each record.

Extras

Days in a month

Write a program that asks the user to input the number of a month (1,2,...,12) and outputs the number of days in that month. For February output "28 or 29". If the user does not input an integer in the range 1 to 12 then output a suitable message (e.g. "A month is denoted by an integer in the range 1 to 12.").

Leap year

Write a program that inputs an integer giving the year and outputs whether or not it is a leap year. For example,

```
Please input year: 1996
1996 is a leap year
```

A year is a leap year if it is exactly divisible by 4 (i.e. no remainder) and not exactly divisible by 100, unless it is divisible by 400. This is why the year 1900 was not a leap year but the year 2000 is. A number is exactly divisible by 4 if the remainder on integer-division is 0. To discover the remainder on integer-division use the Java operator % (5 % 3 is 2). The test x % 3 == 0 for example, checks if x is divisible by 3.

Pounds and pence

Write a program that inputs a price given as a number of pence, and prints out the price as pounds and pence. For example,

```
Enter price in pence: 1415
£14.15
```

You will need integer division by 100 (.../100) to get the pounds, and the remainder (...%100) to get the pence. You may need to consider two cases, or you will end up with

```
Enter price in pence: 1405
£14.5
```

Days in a month again

Write a program that asks the user to input the number of a month (1,2,...,12) and outputs the number of days in that month. This time use a switch statement. For February output "28 or 29". If the user does not input an integer in the range 1 to 12 then output a suitable message (e.g. "A month is denoted by an integer in the range 1 to 12.”).

Inland revenue

Write a program that reads in a person's annual income and then computes and outputs their income tax liability assuming the following:

- there is a £4200 tax free allowance
- the first £4500 of taxable income is taxed at the 'lower rate' of 20%
- the next £22,600 of taxable income is taxed at the 'basic rate' of 23%
- any further income is taxed at the 'higher rate' of 40%

Avoiding Bugs

It’s always better to avoid a bug in the first place, rather than cure it afterwards. That is where programming "style" comes in...Be a stylish programmer and make your programs easier to follow.

TIP 1: Start writing an if statement (and other constructs with brackets) by typing in its outline structure with all the brackets - then go back and fill in the gaps. That way you wont forget the brackets or the else statement.

```
if ( )
{
}
else
{
}
```

TIP 2: Indent your if statements. Every time you type an open curly go in 3 spaces for the following lines. Only go back when you hit the close curly. See how easy it is to see what is in each branch of the following, compared to the version after...with long programs it gets harder unless you indent. Its also easier to see you haven't missed close curly brackets.

```
if (result == 1)
{
   temperature = "hot";
   raining = false;
}
else
{
   temperature = "cold";
   raining = true;
}
System.out.println("That was the weather");
```

compare with:

```
if (result == 1){
temperature = "hot";
raining = false;}
else { temperature = "cold";
raining = true;}
System.out.println(
"That was the weather");
```

TIP 3: Put a blank line before the start of an if statement and after the end (its final curly bracket) so its easy to see where it starts and finishes.

TIP 4: Whenever you write an if statement that tests a value input by the user, check that what happens if they type in the wrong thing - make sure the program does a sensible thing. It can be a good idea to use a final else specifically just to catch these cases and print a warning.

Catching Bugs

Everyone catches colds. Everyone who writes programs gets bugs. One of the skills of programming is to come up with ways that help you work out what you did wrong. Early in your programming career it will mainly be getting bits of the language wrong: misspelling things, getting the punctuation wrong,... Those things the compiler spots for you, but you have to learn how to work out what the compiler is telling you.

+++++HOT+++++ TIP 1: If the program compiles but then does the wrong thing and it is totally baffling why (the normal state with runtime errors!) add in extra print statements so you can see which parts of the program get executed. Comment them out once you have found the bug. Why? Because it allows you to tell which commands are executed - to see into your code as it runs. It also helps you see how and at what line a program crashes (if it does).

```
if (result == 1)
{
   weather = "hot";
   System.out.println(
            "DEBUG: DID HOT"); // add this to debug
}
else
{
   weather = "cold";
   System.out.println(
            "DEBUG: DID COLD"); // add this to debug
}
```

+++++HOT+++++ TIP 2: When you add in debugging print statements, include printing the values of variables so you can see if they have the value you expect at that point.

```
if (result == 1)
{
   weather = "hot";
   System.out.println("DEBUG: DID HOT: weather is " + weather);
                                    // add this to debug
}
else
{
   weather = "cold";
   System.out.println("DEBUG: DID COLD: weather is " + weather);
                                    // add this to debug
}
```

TIP 3: Make sure you haven't added a semicolon after the test. It might not be spotted by the compiler (if you left out the else) but make the program behave really weirdly. If it does cause a compile error the compiler will appear to be complaining about something later.

```
if (result == 1); // THIS ; IS WRONG
{
   weather = "hot";
}
else
{
   weather = "cold";
}
```

TIP 4: Double check you haven't written = (assignment) instead of == (test for equality).

```
if (result = 1) // THIS IS WRONG SHOULD BE ==
{
   weather = "hot";
}
else
{
   weather = "cold";
}
```

Some Common Compile-time errors

Forgot to change to the right directory

```
/week2/programs pc$ javac if_boolean.java
error: cannot read: if_boolean.java
1 error
week2/programs pc$ cd ../week3
```

Oops I forgot to declare the change directory to the place where I'd saved the new program

```
pc$ java if_int
Exception in thread "main" java.lang.NumberFormatException: For input string: "a"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
        at java.lang.Integer.parseInt(Integer.java:447)
        at java.lang.Integer.parseInt(Integer.java:497)
        at if_int.chooseDirection(if_int.java:37)
        at if_int.main(if_int.java:18)
```

Notice this happened when I ran the program not when I compiled it (it compiled fine. It is therefore a run time error not a compile-time error. Things seem to work fine. I ran the program several times before this without problem. Then I typed in a character 'a' when I was asked for an integer. The program is complaining that at a point when it needed a string containing a number, it got something else. To fix this we would need to write code to check that the string entered did hold a number before we do the conversion. That is complicated to write yourself. Alternatively we would need to "catch the exception" using an "exception handler". We are not going to cover that here but you may want to read about it yourself in your text book. A simpler alternative for now would be to do the if test on strings directly if this was a real program.

Unit 3 Exam Style Question

Attempt the following questions in exam conditions. A model answer can be found on the ECS401 QM+ site. Compare your answer with the model answer. How good are your explanations? How good is your code?

Question 1 [25 marks]
This question concerns programs making decisions.
a. [6 marks]
Explain what is meant by each of the following **illustrating** your answer with examples.
i) a type
ii) a boolean expression

b. [9 marks]
Give initial values for variable x in each example below that would lead to "Hello" being printed **justifying** your answer.

i) [2 marks]
```
if (x==0)
{
  System.out.println("Hello");
}
```
ii) [2 marks]
```
if (x<5)
{
  System.out.println("Hello");
}
```

iii) [2 marks]
```
if (x>11)
{
  System.out.println("Goodbye");
}
else
{
  System.out.println("Hello");
}
```

iv) [3 marks]

```
if (x>11 & x<20)
{
  System.out.println("Goodbye");
}
else if (x>2)
{
  System.out.println("Hello");
}
else
{
  System.out.println("Goodbye");
}
```

c. [10 marks]
Write a Java program that is given 2 integers by the user: the first is a number from 1 to 7 representing a day of the week, with 1 for monday etc. The second is a number from 1 to 12 representing a month. The program should convert the numbers into a text equivalent. For example if 1 and 12 were supplied by the user, the program should print:
A monday in December

Question 2 [25 marks]
This question concerns programs defining and using records.

a. [4 marks]
Explain what is meant by the fields of a record **illustrating** your answer with examples.

b. [5 marks]
Explain why methods are an important form of abstraction that can help if a program is to be changed, **illustrating** your answer with an example.

c. [6 marks]
Compare and contrast methods that get and set record fields.

d. [10 marks]
Write a Java program about deadly animals. Information about each animal is stored. The user is asked to name an animal and information about that animal is printed out. Define and use a record to store details of a series of animals including their name, and its 'deadly' score (a number up to 10). Your program should know about at least 5 animals (you may make up their deadly scores). It should use accessor methods.

Unit 4: Counter-controlled Loops (Bounded For Loops)

Learning Outcomes

Once you have done the reading and the exercises you should be able to:

- read & write programs that follow instructions a fixed number of times
- explain what is meant by a for loop
- trace / dry run the execution of programs containing for loops
- read & write methods that return results

From last week …

You should now be able to:

- read & write programs that make decisions
- explain the use of boolean expressions
- explain what is meant by if statements
- trace / dry run programs containing if statements

Do the Interactive Notebooks

This unit comes with a series of interactive exercises in interactive notebooks. A link is in the. You may wish to do them before reading through these notes. Whether before or after, you MUST work through those exercises. These include

- **Interactive Notebook: Unit 4-1 Counter-controlled (For) Loops**
- **Interactive Notebook: Unit 4-2 If statements inside loops**

Find the notebooks here: **https://jhub.eecs.qmul.ac.uk**

Non-technical Reading

The following booklets may help you understand concepts if struggling. They are available from the activities and reading section of QM+.

- **Chapter: 6 of Computing without Computers**

Deeper Reading

The following booklets go into the topics in this section in more detail. Read them if you find the concepts hard to understand. You can download them from the QM+ site.

- **Booklet: Rolling the Dice: Understanding Methods I**
- **Booklet: Global Variables are bad**

Lecture Slides/Notes

Code from last week

You should be able to write programs like the following before moving on.

```
String ans= "no";
ans = inputString("Can I insult you, please?");

if (ans.equals("y"))
{
   System.out.println("You smell!");
   System.out.println("I mean it!");
}
else
{
   System.out.println("You smell of roses!");
   System.out.println("Wow!!");
}

System.out.println("Thank you!");
```

Exercise

- Write a program that prints out a punishment given to you of writing the following line 10 times: "I must not be late for class"

Doing things repeatedly

You can do this by writing a method that includes 10 print statements.

```
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
System.out.println("I must not be late for class");
```

- Very long winded...there must be a better way.

A better way

- It could be worse - I could have said do it 100 times…
- But Computer Scientists are lazy and look for better ways…Each line is the same:
  `System.out.println("I must not be late for class");`
- We need a Java command that says "follow that instruction 10 times".
- If we were the Java inventors, we would need
  - A new keyword to say we are looping, and
  - A way of indicating what command should be repeated, and
  - A way of saying how many times to do it

Inventing our own repetition command

- Suppose we decided on the key word "for" to say we are looping, and
- put the commands to be repeated in curly brackets to be consistent
- …we would need something like

```
for (10 times)
{
   System.out.println("I must not be late for class");
}
```

- Java is slightly more complicated than that (but not much) …

Counters

- What if we had been told to number each line:
```
System.out.println("1: I must not be late for class");
System.out.println("2: I must not be late for class");
System.out.println("3: I must not be late for class");
System.out.println("4: I must not be late for class");
System.out.println("5: I must not be late for class");
System.out.println("6: I must not be late for class");
System.out.println("7: I must not be late for class");
System.out.println("8: I must not be late for class");
System.out.println("9: I must not be late for class");
System.out.println("10:I must not be late for class");
```
- Now every line is different…are we stuffed...back to doing it by hand? NO

Counters

- Notice that every line is still *virtually* the same
```
System.out.println("????: I must not be late for class");
```
- The only thing that differs is where I have put the question marks.
- So all we need is a way of putting something in the place of the ???? that can hold a number that we can then change each time we do the instruction again
- But we can do that - we just need a variable!
```
System.out.println(i + ": I must not be late for class");
```
- Remember using the + sign is just to build a string up from parts
- In this case it combines the *contents* of variable i with the rest of the string given
- I just need a way to set i to 1 at the start, add 1 each time round the loop and stop when it gets to 10

Java for loop

- The Java for loop allows you to build in a counter that can start and end at any point:
```
for (int i = 1; i<=10; i=i+1)
{
    System.out.println
            (i + ": I must not be late for class");
}
```
- This says
- Start a counter variable called i at 1
- Stop when it gets to 10
- Change its value each time round the loop
- i++ is shorthand for the assignment i=i+1;
  so you can write `for (int i = 1; i<=10; i++)`
- Each time round the loop print the message with the current value of i

A flexible loop command

- This gives a really flexible mechanism for doing things repeatedly.
- Suppose your punishment was to write the 2 times table out. Example lines of code are …
```
System.out.println("2 times "+ 4 + " is "+ 2*4);
System.out.println("2 times "+ 5 + " is "+ 2*5);
```
- Taking the common parts we get:
```
System.out.println("2 times "+ ???? + " is "+ 2*????);
```
- Replace the ???? By a loop counter variable and put it in a loop…

The 2 times table

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

Exercise

- Write a program that prints out a series of squares up to 5 squared: ie it prints
  1 squared is 1
  2 squared is 4
  3 squared is 9
  4 squared is 16
  5 squared is 25

A terminating variable

- The user can supply the number of times the loop executes.
- Just use a variable for the finish value (that's what variables are for).

```
String text = inputString("What is n?");
int n = Integer.parseInt(text);

for (int i = 1; i<=n; i++)
{
System.out.println("2 times "+ i + " is "+ 2*i);
}
```

Accumulator variables

- A common thing we need to do is gradually build a result up a bit at a time in a variable
- Eg write a loop that adds the first 5 integers
- We can add the loop counter on to a running sum each time round the loop

```
int sum = 0;
for (int i = 1; i<=5; i++)
{
    sum = sum + i;
}

System.out.println("The answer is "+sum);
```

Exercise

- Write a program that multiplies the first 10 integers.

For loops in general

- The general form of a for loop is

```
for (int i= <start>; i<= <finish>; i++)
{
  <body>
}
```

- *<body>* is the thing you do repeatedly
- Work that out first by working out what is common and what changes about the instructions to be repeated.

For loops

```
        for (int i= <start>; i<= <finish>; i++)
        { <body> }
```

- means:

  for i changing from <start> to <finish>

  do the <body>instructions over and over

- (unless you change i in the <body>- DONT)
- In other words,
  execute <body> with i=<start>then
  execute <body> with i=<start>+1then
  execute <body> with i=<start>+2then
  . . . then finally
  execute<body> with i=<finish> then
  go on to the rest of the program immediately below the loop.

Tracing

You can put print statements in a program to see how the variables change as the program executes line by line.

You can also do this on paper. Step through the execution writing down at each stage which point of the program you are at, and what the different variables contain.

One way is to write down the program with boxes for the variable. As you execute the program, fill in the boxes.

Another is to write a table with line numbers.

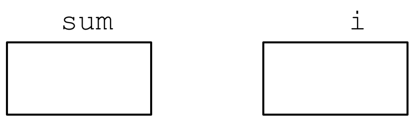
Dry Run/Trace of Loops

- You need a box to keep track of the value of each variable – including i
- i is initialized when the loop is first encountered and changed every time you get back to the top of the for loop again.

```
int sum=0;

for (int i=1; i<= 5; i++)
{

  sum = sum + i;
}
```



Tracing in a table

By labelling the lines you can also trace variables in a table.

Each line you encounter in the program has a row allocated where you fill in the values of the variables at that point

```
int sum =0;                //L1
for (int i=1; i<= 2; i++)  //L2
{
     sum = sum + i;        //L3
}
System.out.println(sum);   //L4
```

eg

LINE	sum	i	notes
L1	0		i doesn't exist yet
L2	0	1	Set i to 1 and test to check it is less than or equal to 2. It is so enter the loop body at line L3
L3	1	1	adding sum and i to get a new value for sum
L2	1	2	Add one to i then test of i<=2 is true so stay in the loop
L3	3	2	Adding 1 and 2 to get a new value for sum
L2	3	3	Add one to i. Now I is greater than 2 – the test is false so move on to line L4
L4	3	3	Prints 3 to the screen

Nested For loops

- Any commands can go into the body of a for loop: eg if statements or another for loop.
- Such"nested" for loops allow things to be done in two dimensions:

```
for (int i=1; i<=3; i++)
{
   for (int j=1; j<=5; j++)
   {
      System.out.println("*");
   }
   System.out.println();
}
```

Good idioms/patterns

- Rather than working out how to do a loop from scratch each time there are some common patterns that you will use over and over for particular tasks.
- They are common across lots of programming languages - not just Java

Idiom: Counter controlled loop

- How to do something exactly m times.

```
for (int i=1; i<=m; i++)
{
   <the thing to do m times>
}
```

Idiom: Accumulating a result

- How to build a result up in an accumulator a bit at a time over m times.
- This might be used to build up a string of output to then print in one go

```
for (int i=1; i<=m; i++)
{
  <preparation for calculation>
  acc = acc + <calculation>;
}
```

Idiom: Conditioned loops

- Process something m times but only if a condition is true for the element processed
- We will see this used later for searching for data.

```
for (int i=1; i<=m; i++)
{
  if( <test depending on i> )
  {
```

```
            <process i>
        }
    }
```

Idiom: Rectangular for loop

- Process n elements in some series m times
- We will see this used in an algorithm to sort data.

```
    for (int i=1; i<=m; i++)
    {
        for (int j=1; j<=n; j++)
        {
            <body>
        }
    }
```

Idiom: lower triangular for loop

- We will use this in a faster sort algorithm. The inner test end depends on the outer counter

```
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=i; j++)
        {
            <body>
        }
    }
```

Idiom: upper triangular for loop

- Notice how here the start point of the inner loop depends on the outer counter.

```
    for (int i=1; i<=n; i++)
    {
         for (int j=i; j<=n; j++)
         {
           <body>
         }
    }
```

Further Extensions

- Experiment with loops where you change the initialisation of the loop counter
- Use different kinds of tests for its termination and put different assignment commands in place of i++ (like i=i+2 or i=i-1)

Input Validation (again)

- A common issue when inputting numbers is that the program crashes if a non-number is input. We can now do better. People will mistakenly enter non-numbers.
- Lets look at checking positive integers only. We must check that each character is a digit.

```
public static boolean isPositiveInteger(String s)
 {
     for (int i = 0; i < s.length(); i++)
     {
         char c = s.charAt(i);

         if (!(isDigit(c)))
             return false;
     }
     return true;
 }
```

Exercises

- Write the isDigit method the above needs.
- Modify the above to check if the string is any legal integer (positive or negative)
- Write a new inputInt method that returns 0 if a non-number is input.

Unit 4 Example Programs and Programming Exercises

Only try these exercises once you feel confident with those from the previous week/unit.

1. For statements

There are a series of full programs that you can compile and run on the ECS401 QM+ site – unit 4 area, download, compile and run them, then **experiment with them modifying them to do slightly different things** until you understand how they work.

What happens when you type in different numbers? How about 0 or -1? What is the thing that is repeated each time? The programs include:

- **forname.java:** Print my name 3 times
- **for4.java** : Doing something a fixed number of times (here 4 times)
- **iloveyou.java** : Here is a simple use of a loop for lovers where the user gives the number of times to do it
- **for0.java** : Doing the same thing over and over with different values
- **for1.java** : Build an answer up a bit at a time in an accumulator variable. Each time round the loop adds more
- **exp_fact.java** : Here is an example of doing a calculation at each step building a calculation a line at a time
- **ages.java** : Returning results from a method
- **weeklypay.java:** more complex use of accumulator variables

Here is a typical for loop program

```
/* ****************************************
AUTHOR Paul Curzon

  A program to print the averages of a series of
  pairs of ages.

  Illustrates how methods can return values.

******************************************** */

import java.util.*;

class ages
{
    public static void main (String[] param)
    {
        averageAges();
        System.exit(0);

    } // END main


/* ********************************
    Ask for 10 pairs of ages for a husband and wife printing their average
*/
    public static void averageAges()
    {
       String resulttext = "";

       for (int i=1; i<=10; i++)
       {
          int averageAge;

          System.out.println("I need you to give me a pair of ages for a couple");
          averageAge = calculateAverage();
          resulttext = resulttext + "Couple " + i + ": " + averageAge + "\n";
        }

        System.out.println("Here are the average ages of the couples");
        System.out.println(resulttext);
        return;
    } // END averageAges


/* ********************************
    A method that asks for ages of a wife and husband and returns their average
*/
    public static int calculateAverage()
    {
       int husband;
       int wife;
       int average;

       husband = inputInt("Give me the husband's age");
       wife = inputInt("Give me the wife's age");

       average = average2(husband, wife);

       return average;
    } // END calculateAverage


/* ********************************
    Calculate the average of two given numbers
*/
    public static int average2(int v1, int v2)
    {
       int average;

       average = (v1 + v2) / 2;

       return average;
    } // END average2


    // A method to input ints
  public static int inputInt (String message)
  {
       Scanner scanner = new Scanner(System.in);
       int answer;

       System.out.println(message);
       answer = Integer.parseInt(scanner.nextLine());

       return answer;
  } // END inputInt

} // END class ages
```

2. Printing lines
Modify the forname.java program to do each of the following:
a) print YOUR name instead of mine 3 times.
b) print your name 6 times.
c) as above but print once before the list of names “Starting List” and after the list of names “Ending List”.
d) as above but store the number of times the name is to be printed in a variable.
e) as above but input the number of times the name is to be printed from the user.
3. Punishment made easy
Write a program that outputs a punishment message of your choice such as "Paul must not sleep in class" 20 times.
4**. Adding trace statements**
Add console print lines (System.out.println) to the above for1.java program to print the values of the variable i inside the loop to trace its execution - so you can see what happens to the variables in the loop each time as it executes. (See the catching bugs tips for more on this).
5. List the integers
Write a program that outputs the list of integers from 1 to 20.
Then write a program that inputs a non-negative integer n from the keyboard and outputs the list of the integers from 1 to n.
6. Add the integers
Write a program which inputs a non-negative integer n from the keyboard and outputs the sum of the integers from 1 to n. For example given the number 5 it adds the numbers from 1 to 5.
Once you have done these, try the next assessed exercise. The assessed exercises must be done on your own. If you have problems with them then go back to the non-assessed exercises.
7. Nested for loops
The following exercises concern nested for loops - loops inside loops and the difference between this and to loops that follow loops.
Compile and run the following programs, experiment with them modifying them to do slightly different things. What happens when you type in different numbers? How about 0 or -1? What is the thing that is repeated each time?

- You can put for loops one after the other
- or one inside another

8**. Squares**
Write a program that inputs a positive integer n from the keyboard and outputs an n by n square of 'P' characters to the console. HINT Write one for loop first that prints a single row of n 'P's. Get that working then put a loop round it to turn it into a square.
9**. Rectangles**
Write a program which inputs two positive integers m and n from the keyboard and outputs an m by n rectangle of '*' characters.
10**. Tables**
Write a program which outputs a multiplication table to 12; i.e.

```
1   2   3   4   5   6   7   8   9  10  11  12
2   4   6   8  10  12  14  16  18  20  22  24
...
```

11. Printing triangles
Write a program which inputs a positive integer n and outputs an n line high triangle of '*' characters whose right-angle is in the bottom left corner. For example, if 5 is input the output should be

```
*
**
***
****
*****
```

11. Repeated dice throws
Write a program that throws a 6 sided dice 20 times printing the result of the throw each time. Extend the program to print out how many times each number came up.

Additional Exercises
The following are additional, harder exercises. The more programs you write this week the easier you will find next week's work and the assessments and the quicker you will learn to program. If you complete these exercises easily, you may wish to read ahead and start thinking about next week's exercises.

Clockwork
Write a program that behaves like a 24-hour digital watch. The program should ask the user to input the day of the week (Mo, Tu, ...) and the hour, minute and seconds. If the user inputs a valid day of the week and a valid time the clock advances the time by one second and then displays the new day and time. For example,

```
   Input day     : Sa
   Input hour    : 23
   Input minute  : 59
   Input second  : 59
```

then your program will output

```
   Su 0 0 0
```

Now put your clock into the body of a for statement and make it 'tick' (advance by 1 second) 100 times.

Avoiding Bugs

It's always better to avoid a bug in the first place, rather than cure it afterwards. That is where programming "style" comes in...Be a stylish programmer and make your programs easier to follow.

TIP 1: Indentation is very important with loops – do it in a similar way as for if statements. Open { mean indent, } mean undo the indentation (see below)

Catching Bugs

TIP 1: ADD TRACE STATEMENTS If a program with a loop does something really strange and doesn't seem to do the loop, you need to see what it's doing step by step. Add some console print statements that show you where the program got to. Add one before the loop, one inside and one after the loop. For example if trying debug the following loop:

```
            for (int i=1; i<=n; i++)
            {
               resulttext = resulttext + "*";
```

```
        }
```

Add the following print statements:

```
    System.out.println("Entering the loop");
        for (int i=1; i<=n; i++)
        {
           System.out.println("In the loop");
           resulttext = resulttext + "*";
        }
    System.out.println("Left the loop");
```

You will then be able to see if the problem is because the loop is never entered, or is entered but doesn’t go round the correct number of times or does the loop properly and leaves - so the problem is somewhere else such as how the results are later printed.

TIP 2: ADD PRINT STATEMENTS THAT SHOW THE VALUES OF VARIABLES If after following tip 1, you know the loop isn't working properly but still can't see why, the next thing is to add some trace statements to show the values of the variables. The values inside the loop as the loop goes round and round are most important. It is always a good idea to print out the loop counter's value (i below) and any other variables whose values change in the loop. For example the above for loop might be modified as follows:

```
    System.out.println("Entering the loop");
    System.out.println("n has value " + n);
    System.out.println("resulttext has value " + resulttext);
        for (int i=1; i<=n; i++)
        {
           System.out.println("In the loop");
          System.out.println("i has value " + n);
          System.out.println("n has value " + n);
          System.out.println("resulttext has value " + resulttext);

           resulttext = resulttext + "*";
        }
    System.out.println("Left the loop");
    System.out.println("i has value " + n);
    System.out.println("n has value " + n);
    System.out.println("resulttext has value " + resulttext);
```

Remember only to try and print the values of variables after they have been declared. For example in the above you would get a compile error if you tried to print the value of i before the loop as it doesn’t exist then.

TIP 3: DON’T FORGET: REMOVE ALL DEBUGGING LINES ONCE IT WORKS!

TIP 4: OVER RUNNING LOOPS A common problem with loops is they loop one time to many or too few. Check explicitly every time you write a loop that you got the end test right.

The following loop goes round 5 times, with the loop counter i counting 1,2,3,4,5 and then stopping.

```
        for (int i=1; i<=5; i++)
        {
          resulttext = resulttext + "*";
        }
```

If you put a less than sign instead of a less than or equal to sign as below it would only count 1,2,3,4 and then stop.

```
        for (int i=1; i<=4; i++)
        {
          resulttext = resulttext + "*";
        }
```

Some Common Compile-time errors

Did not initialise a variable that is used to build up a result in a loop

```
pc$ javac for1.java
for1.java:52: variable resulttext might not have been initialized
               resulttext = resulttext + "*";
```

When running loops its easy to forget to initialise the variables changed in the loop. Here the line of code with the problem is inside a loop:

```
        for (int i=1; i<=n; i++)
        {
          resulttext = resulttext + "*";
        }
```

The first time round the loop, resulttest has no value set at all. I have to give it a value before entering the loop to add things to.

```
        resulttext ="";
        for (int i=1; i<=n; i++)
        {
          resulttext = resulttext + "*";
        }
```

It is easy to use the wrong variable in the loop!

```
pc$ javac iloveyou.java
iloveyou.java:57: cannot find symbol
symbol  : variable j
location: class iloveyou
              System.out.println(j+": I love you!");
                                                   ^
1 error
```

Here we used a loop counter of i but then used j by mistake in the loop - the compiler is just saying "j - what is j. You didn't tell me about a j?"

```
  for (int i=1; i<=n; i++)
        {
          System.out.println(j+": I love you!");
        }
```

Better still it should be done as part of the declaration of the variable

```
pc$ javac expFactTable.java
error: cannot read: expFactTable.java
1 error
```

Oops the file didn’t exist - I forgot to add the .java on the name of the file when I saved it. A similar message would have occurred if I got the capitals wrong or had saved the file in a different directory to this one.

Run time errors

A run time error that is easy to make is to get the formatting of the thing printed wrong in a loop - forgetting to add the new line character (\n) in so everything goes on one line.

Take care as well that you dont put things in loops that you only want done once or vice versa.

It is also easy to get the control character slashes the wrong way round - it is \n that goes to a new line - not /n

Unit 4 Exam Style Question

Question 1 [25 marks] This question concerns programs acting repeatedly.

a. [9 marks]

For each of the following statements **state** how many times the letter 'p' will be printed **justifying** your answer.

i) [2 marks]

```
for(int i = 1; i < 4; i++)
{
  System.out.println(”p”);
}
```

ii) [2 marks]

```
for(int i = 1; i < 1; i++)
{
  System.out.println("p");
}
```

iii) [2 marks]

```
for(int i = 1; i <= 4; i++)
{
  for(int j = 1; j <= 4; j++)
  {
         System.out.println("p");
  }
}
```

iv) [3 marks]

```
for(int i = 1; i <= 4; i++)
{
  for(int j = 1; j <= i; j++)
  {
       System.out.println("p");
  }
}
```

b. [6 marks]

The following code fragments contain errors. For each **state** what the error is and whether the compiler would detect it **justifying** your answer.

i) [3 marks]

```
/* write Hello 10 times */
for(int i=0; i < 10; i++);
{
   System.out.println("Hello");
}
```

ii) [3 marks]

```
/* write numbers from 1 to 10 */
for(int i=1; i < 10; i++)
{
   System.out.println(i);
}
```

c. [10 marks]

Write a Java program that uses a for loop to print the n times table, where n is a number supplied by the user. An example run might be:

```
Which times table should I print? 2
1x2=2   2x2=4   3x2=6   4x2=8   5x2=10
```