ECS510

“ADSOOF”
An explanation of the principles and practices of the module

Matthew Huntbach
matthew.huntbach@qmul.ac.uk

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

If your answer is something like "It sets up a variable called `i`, with initial value `0`, uses that in a for loop to index the arraylist `a`, each time round the loop calls the method `check` on `ch` with the element at position `i` in `a` as its argument, …" you have not yet learnt how to program.

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

If your answer is something like "It sets up a variable called `i`, with initial value `0`, uses that in a for loop to index the arraylist `a`, each time round the loop calls the method `check` on `ch` with the element at position `i` in `a` as its argument, …" you have not yet learnt how to program properly.

Learning to write

- Learning to write English is more than learning the meaning of individual words and how to put them together to make a grammatically correct sentence
- The key aspect of learning any human language is being able to use it to express yourself
- Also being able to understand what others mean when they say things without having to go through a detailed grammatical analysis
- A good programmer needs to be able to look at a simple piece of code and be able to tell right away what its underlying meaning is without having to go through a detailed step by step analysis
- A good programmer needs to be able to write small pieces of code in a similar instinctive manner

How do I get to be a good programmer?

?

How do I get to be a good programmer?

- Practice

How do I get to be a good programmer?

- Practice
- Practice

How do I get to be a good programmer?

- Practice
- Practice
- Practice

How do I get to be a good programmer?

- Practice
- Practice
- Practice
- Study

How do I get to be a good programmer?

- Practice
- Practice
- Practice
- Study
- Practice

How do I get to be a good programmer?

- There is no way round this, like many other practical skills you have to spend a lot of time actually doing it in order to reach the point where it comes naturally
- You cannot do it just by reading about it, and if you try to do it that way it will seem very hard and complex
- It will come easier to some than to others, so have patience and carry on practicing even if you are finding it hard
- The theory behind it makes much more sense when you have had some practice
- So mix your practice with reading
- When you have read something, try it out in practice

Playing the scales

- If you are learning to play a musical instrument, you spend a lot of time “playing the scales”
- This means playing the basic sound patterns, and doing it repeatedly
- It can seem boring and repetitive, and you want to move on to playing more complex and interesting tunes
- Playing the scales until making those sound patterns comes so naturally that you don’t have to think about it is essential to becoming a skilled musician
- Skilled music teachers will devise scales exercises that cover the important major aspects of use of the instrument
- Novice players may not realise what these important underlying aspects are

Peter Norvig: Teach yourself programming

- Peter Norvig is Director of Research at Google, and before that was a lecturer in Computer Science at the University of California, and co-author of the most widely used text-book on Artificial Intelligence: *"AI: a Modern Approach"*
- See his website: `norvig.com`
- On this website you will find many interesting comments and links, but perhaps the most famous is called

  "Teach Yourself Programming in Ten Years"
- Norvig was critical of books which claim to teach you how to program in 21 days or 24 hours
- My advice: if you have the book *"SAMS Teach Yourself Java in 24 hours"*, burn it

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

?

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

It takes an arraylist of elements of a particular type, a checker of elements of a particular type, and a "backup" element of that type. It returns the lowest indexed element from the arraylist which satisfies the checker. If none of the elements in the arraylist satisfy the checker, it returns the backup element.

Well structured code

- As mentioned previously, a good programmer should be able to say right away what a simple piece of code will do in terms of its underlying meaning
- Well-structured code means keeping all code simple
- But real-life software systems are large and complex
- Keeping code simple means as far as possible having it divided into small pieces, each with its own purpose and understandable on its own
- Limiting the possible interactions between pieces of code is an important aspect of this
- Being able to see code in terms of the services it provides to other code rather than what is inside it to make those services work is another important aspect

Generalisation

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

Generalisation is another important aspect of well-structured code. It means writing code in a way that is general, covering just the basic aspects, which means the same piece of code can be re-used many times in many different circumstances. This `findOk` method is an example.

Specialisation

```
class GreaterThan implements Checker<Integer>
{
 private int level;

 public GreaterThan(int lev)
 {
  level=lev;
 }

 public boolean check(Integer n)
 {
  return n>=level;
 }
}
```

Specialisation

```
class AllPassed implements Checker<Student>
{
 private int level;

 public AllPassed(int lev)
 {
  level=lev;
 }

 public boolean check(Student stud)
 {
  for(Module m : stud.modules())
     if(m.mark()<level)
        return false;
  return true;
 }
}
```

Re-use

- If we have

```
ArrayList<Integer> numbers = …;
int fallback= …;
Checker<Integer> gt = new GreaterThan(100);
Checker<Student> p = new AllPassed(40);
ArrayList<Student> students = …;
Student fred = …;
```

- Then

```
int best=findOk(numbers,gt,fallback);
Student Sample=findOk(students,p,fred);
```

  both use the same `findOk` method to do what seem to be different things, but are actually two versions of the same thing

- This is better than writing separate code to perform what is really the same task, for `Integers` and `Students`

Re-use

Or for different checks on `Integers` and `Students`:

```
class FemaleChecker implements Checker<Student>
{
 public boolean check(Student stud)
 {
  return stud.gender().equals("female");
 }
}
```

So

```
Checker<Student> fcheck = new FemaleChecker();
Student grep=findOk(students,fcheck,fred);
```

sets `grep` to refer to the lowest-indexed `Student` object which represents a female student from the list `students`, or to the `Student` object also referred to by the variable `fred` if none of the objects in the list represents a female student.

Don’t Repeat Yourself (DRY)

- See in “*97 things every programmer should know*”:
  `programmer.97things.oreilly.com/wiki/index.php/Don't_Repeat_Yourself` (contribution no. 30)
- “Of all the principles of programming, Don’t Repeat Yourself (DRY) is perhaps one of the most fundamental”
- An algorithm or data structure should be in one piece of code, used by calling that code with appropriate parameters or extension by inheritance
- If instead the code itself is repeated (“cut and paste”), then if you want to change the algorithm or data structure you have to find every place where it is repeated and change each of them

Meaningful Variable Names

- Consider:

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> b, T c)
{
 for(int d=0; d<a.size(); d++) {
    T e=a.get(d);
    if(b.check(e))
       return e;
    }
 return c;
}
```

- To the computer, this is just the same as the previous version of `findOk`. To the skilled programmer, it is harder to understand.
- Method and type names should be meaningful as well.
- Conventions for short names should be followed

More Generalisation

```
public static <T> T findOk(Iterable<T> a,
                           Checker<T> ch, T backup)
{
 for(T t : a)
    {
     if(ch.check(t))
        return t;
    }
 return backup;
}
```

This is a more generalised form of the method `findOk`. This version will take an `ArrayList<T>` argument, but it could also take an argument of another collection type, or even an array. It does not rely on elements of the collection having a position accessible through a numerical index.

Common patterns

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       return t;
    }
 return backup;
}
```

A `return` statement always terminates a message call. So if it is inside a loop, since it terminates the whole method call, it terminates the loop inside that method call. This means that the `findOk` method does not always check every element in the arraylist. The final `return` is reached only if the inner `return` is never reached.

Unnecessary complexity

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 boolean found=false;
 T retvalue=backup;
 for(int i=0; !found&&i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       {
        retvalue=t;
        found=true;
       }
    }
 return retvalue;
}
```

This code is correct, but harder to understand at a glance.

Layout

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup) {
 boolean found=false;
 T retvalue=backup;
 for(int i=0; !found&&i<a.size(); i++) {
 T t=a.get(i);
 if(ch.check(t)) {
 retvalue=t;
 found=true; } }
 return retvalue;
}
```

This code is correct, but almost impossible to understand at a glance.

To the computer, however, it is exactly the same as the previous code.

Software Engineering

- “Software engineering” means the practice of developing computer programs for use in the “real world”
- It covers all aspects from those to do with the actual code, through to human issues such as interacting with customers to find what they want from a program and managing the teams of programmers needed to write large scale programs
- Real world programs are MUCH larger than the sort of programs you write when you are learning to program
- Real world programs are subject to frequent modification as new versions are developed to add features and meet changes in customer requirements
- It is important to write code in a way that enables it to be modified easily, so structure and understandability are extremely important

Inefficiency

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 T retvalue=backup;
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       retvalue=t;
   }
 return retvalue;
}
```

This code is inefficient and will not always return the same value as the previous code.

Inefficiency

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 T retvalue=backup;
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       retvalue=t;
   }
 return retvalue;
}
```

It will return the highest indexed element which satisfies the check rather than the lowest.

It also unnecessarily goes through the whole arraylist. In general do not continue with computation when further work makes no difference to what is returned.

Efficiency

- There are two aspects to the study of efficiency, one is informal “common sense” efficiency issues such as this
- The other aspect is the formal study of time and space use in algorithms, which we will cover briefly
- Choosing an efficient algorithm and avoiding “common sense” inefficiencies are an important aspect of good programming
- Making code more complex and hard to understand because you just think it might improve efficiency is poor practice
- Writing code in a well-structured way means the algorithms and data structures used are isolated in separate pieces of code, and so can be easily modified to improve efficiency without having to change the rest of the program

Incorrectness

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 T retvalue=null;
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       retvalue=t;
    else
       retvalue=backup;
   }
 return retvalue;
}
```

This code is incorrect.

Incorrectness

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 T retvalue=null;
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       retvalue=t;
    else
       retvalue=backup;
   }
 return retvalue;
}
```

This code is incorrect. It can only return either the last element in the arraylist or the `backup` value.

Incorrectness

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 T retvalue=null;
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
       retvalue=t;
    else
       retvalue=backup;
   }
 return retvalue;
}
```

This code is incorrect. It can only return either the last element in the arraylist or the `backup` value, or `null` if the list is empty.

Testing and Correctness

- Getting a feel for code so that you avoid “obvious” mistakes is something you learn through experience
- With experience you become used to particular patterns of code
- With experience you get used to how code works, so you can work out in your head what will happen when a particular small piece of code is executed
- However, it is still very easy to miss problems, you should ALWAYS test code, never just assume it works as you suppose
- Testing is an important part of programming, it is not enough to say “it works on one example so it must be correct”, there must be careful consideration of test data to cover all possibilities
- Formal proof of algorithms is another approach, which we will not cover, apart from the concept of “loop invariant”

Specification

- When the code was written, was it to meet the requirement "Return an element which satisfies the checker, or the backup element if none of them do"?
- This is an "underdetermined" specification, meaning more than one solution would answer it
- So if it is not specified, it does not matter whether it is the lowest or highest indexed element which meets the specification that is returned, or any other
- The algorithm chosen to meet the specification determined which is the actual element returned
- However, the programmer should point out the underdetermined nature of the specification, and note how the algorithm deals with the underdetermined aspect

Undetermined and Non-deterministic

- An underdetermined specification will generally be met with a deterministic implementation
- That means whenever you use it with particular arguments it will always deliver the same result even though other results would meet the specification
- A non-deterministic implementation is one where the actual code could return different results when called with the same arguments
- This would have to be actually programmed in, such as finding all the elements that satisfy the checker and picking one at random
- There is no point in doing this unless it is specifically asked for

Partial Specification

- What if the specification were “Take an arraylist of elements of a particular type and a checker of that type and return the lowest indexed element from the arraylist which satisfies the checker”?
- This specification does not tell you what to do if none of the elements in the arraylist satisfies the checker
- It can be hard to write a specification which covers every possible aspects, a specification in which some possible arguments are not covered is termed “partial”
- A good programmer should recognise when a specification in partial, and deal with it appropriately
- Asking the person who wrote the specification to clarify is the best way of dealing with it

Making a Partial Specification Total

- A “total” specification is one which says what to do for every possible argument which fits into the types of the parameters
- Often a partial specification is deliberate, it means it does not matter what is done in cases which are not specified
- What is done, however, should be safe and consistent with the general use of the method
- Returning `null` is not safe, because if it happens and is not specifically checked for, the `null` gets passed on in program execution and can cause problems elsewhere
- Throwing an exception is a good way of dealing with a partial specification, as it means the problem is detected right away, and if it is a checked exception code has to be written to deal with it
- How partial specifications are dealt with should always be documented by the programmer

Ambiguity

- An ambiguous specification is one where there are more than one possible interpretations, and they are conflicting
- An ambiguous specification is a mistake, not deliberate, it comes about because human language has ambiguity
- For example, the word “it” is shorthand for something mentioned previously, but what if more than one things were mentioned previously? Sometimes it is not sure what “it” means.
- It may be that the writer of the specification assumed one interpretation and did not realise the other was possible
- Writing specifications in formal logic rather than human language is one way round that, but it has its own problems as formal logic is itself an artificial language
- Giving examples with the specification often helps resolve ambiguties

Static and Non-static

- In Java, a method declared as `static` is one that is not called on an object, it works only with the arguments it is given
- A method which is not declared as `static` must be called on an object, it works with the arguments it is given and also the contents of the object it is called on
- A method which is not declared as `static` is termed an "instance method" as it is called on an object which is an "instance" of the class the method is in
- So what a call to such a method does for particular arguments may change if what is inside the object it is called on changes
- This is not what "non-deterministic" means, however

Side-effects

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=a.size()-1; i>=0; i--) {
    T t=a.get(i);
    if(ch.check(t))
       return t;
   }
 return backup;
}
```

This code efficiently returns the the highest indexed element which satisfies the check.

However, if the method `check` had a side-effect on its argument, and you wanted that side-effect to be applied to every element in the arraylist, you would need the other version.

Side-effects

```
class GradeSetter implements Checker<Student>
{
 public boolean check(Student stud)
 {
  int m=stud.getMark();
  if(m<40)
     stud.setGrade("fail");
  else if(m>=70)
     stud.setGrade("distinction");
  else
     stud.setGrade("pass");
  return m>=60;
 }
}
```

This is poor code, in general a method which does one thing should not also do something else.

“Uncle Bob” Martin: The SOLID Design Principles

See: `www.davesquared.net/2009/01/`

Or the notes on object oriented analysis and design for ECS414

- Single Responsibility Principle
  A class should have only one reason to change
- Open Closed Principle
  You should be able to extend a class’s behaviour without modifying it
- Liskov Substitution Principle
  Derived classes must be substitutable for their base classes
- Interface Segregation Principle
  Make fine grained interfaces that are client specific
- Dependency Inversion Principle
  Depend on abstractions not on concretions

What does this method do?

```
public static <T> T findOk(ArrayList<T> a,
                           Checker<T> ch, T backup)
{
 for(int i=0; i<a.size(); i++) {
    T t=a.get(i);
    if(ch.check(t))
      return t;
    }
 return backup;
}
```

"It goes through the elements of the arraylist one at a time in position order, checking each one. When it finds one which satisfies the check, it returns it and so does not check the remaining elements."

This is a description as an algorithm rather than as a specification of its end result. It says how it does it, not just what it does, but it is a generalised description which avoids code details.

Users

- In computer programming, the word “user” means the human being who interacts with the software system
- ADSOOF code has no users in this sense
- The human user of a software system or “application” has no interest in the details of the code underneath, s/he is only interested in it performing the task it is meant to do
- ADSOOF code is part of the internal structure which makes the application perform the task
- ADSOOF code interacts with other code, not with users
- Interaction is through methods being called and returning values or having side effects, not through reading and writing things
- Interaction with users is indirect

Abstract Data Types

- A data structure and the algorithms which manipulate it will generally be inside a class, with objects of that class interacting with other objects only through the methods called on them
- An Abstract Data Type is a type defined by the definition of the methods that can be called on objects of that type, and not by the algorithms and data structures which makes those methods work
- The key principle here is to make programs well structured by establishing a clear separation between the implementation code and the code which interacts with it indirectly through the abstract data type methods
- This key principle is often missed in older textbooks and in introductory teaching of programming

Introductory Programming

- Introductory programming tends to be focused on the mechanics of the programming language rather than on the abstractions it is being used to implement
- Introductory programming tends to be about direct interaction with a human typing things and reading things rather than interaction between code
- Introductory programming tends not to be about breaking code into separate well-defined parts
- This is perhaps necessary to start you off writing code, but to progress you need to break out of some of the habits and ways of thinking that come from introductory programming
- Not doing this leads to complex and messy code
- Often it is those who are good at programming who find it hard to break away from this because they can cope with complex and messy code on a small scale

Object-Oriented Programming

- In introductory programming in Java we think of the `main` method as the main method
- In more realistic programming in Java, the `main` method exists just to set things up
- In more realistic programming in Java, most of the code is in separate classes
- Classes can define objects, but also hold collections of related static methods
- The idea of object-oriented programming is that you see programs in terms of objects which interact by calling methods on each other
- You can use an object without knowing the code in the class which defines it, all you need to know is its public methods
- This fits with the idea of abstract data types

“Remedial” Programming (recap)

- Some aspects of ADSOOF overlap with what you covered in the 1st year
- If you are a confident programmer, don’t be tempted to “slack off”, it’s easy to miss the point where it takes you further
- If you are a less confident programmer, take the opportunity to revise and gain a deeper understanding of topics you were uncertain about first time round

MOST IMPORTANT! (recap)

- You can use a piece of code (class, static method) without knowing the code, so long as you know its specification and it works according to its specification
- You can write a piece of code (class, static method) without knowing how it is going to be used, so long as you know its specification and it works according to its specification

Defining your own language

- Another way of thinking about this is that instead of programming in a pre-defined language, you write your own language which fits your needs and then use it
- Or you write code in your own language, and then write the code which implements that language
- You define and write classes to give the types you need
- You define and write methods to give the operations you need
- You use your own classes and methods as if they were pre-defined
- Algorithms and data structures are about the code underneath which provides these classes and methods

Practice, practice, practice

- It takes practice to get used to programming in the object-oriented style
- It is not enough just to know the theory of object-oriented programming, or just how the object-oriented features work as an aspect of the programming language
- It has to be something which comes naturally to you
- But it can be hard to see the benefits of the additional complexity of object-oriented programming when you are only dealing with small programs
- It takes practice to use computer code naturally as a language for expressing what you want to do

“We have already done this in the 1st year”

- In order to be thoroughly familiar with standard features and techniques, you need plenty of practice, you have to use those features and techniques until they really come naturally
- There are many subtle aspects of basic programming features which you need to experience in actual code
- It is not enough to know something in theory, you also have to be familiar with it in practice
- There are many misunderstandings which people have about how aspects of code work which persist if you have not had enough experience to have discovered an example which shows up the misunderstanding

“It is too easy”
“It is too hard”

- For those who are good at programming, ADSOOF provides plenty of lab exercises to give more practice
- For those who find programming more challenging, ADSOOF provides material that helps you catch up with the basics
- In both cases you need to put the work in:
- Don’t be tempted to slack off because:
  - “I know this already, I don’t need to do this work”
  - “This is too hard, maybe I’ll catch up later
- The new aspects of ADSOOF arrive subtly, if taken with the right attitude your understanding of programming will gradually change

Recursion and Pointers

- There is little point in knowing the theory of advanced algorithms and data structures if you do not have a full understanding of and experience in more basic aspects
- You should avoid an approach to learning which supposes it is about knowing a large number of facts
- Two of the most important basic aspects of algorithms and data structures which many never properly pick up are recursion (an aspect of abstraction) and pointers (which in Java terms means particularly the implications of aliasing)
- Misunderstanding of both these topics often comes from misunderstanding of basic aspects of Java code which tend to be skated over in introductory programming

Lisp Lists

- You will be introduced to a simple abstract data type which I call `LispList`
- It is introduced to help:
  - Get across the concept of abstract data types, and distinguishing them from their implementation
  - Get across the concept of constructive operations on an immutable data structure as a different approach to destructive operations on mutable data structures
  - Give more practical experience in recursion
  - Get across the concept of implementing a generic data type
- It is NOT introduced because it is something you would use in future

Peter Landin: Lisp

- Lisp was one of the earliest programming languages, but it worked on a different basis than other programming languages: a mathematical model of computing (lambda calculus) rather than a model based on computer hardware
- Peter Landin was a computer scientist who developed this idea, and proposed it as the basis for computer programming languages in general
- See his famous paper "The Next 700 Programming Languages":

  `www.cs.cmu.edu/~crary/819-f09/` (September 16)
- Peter Landin later became a professor at Queen Mary, the building where the Computer Science offices are is named the "Peter Landin Building" in his honour

Rich Hickey: Clojure

- Lisp was the first in the family of "functional" languages, there has been a recent revival of interest in this form of programming
- A modern version of Lisp called "Clojure" was developed a few years ago, and is now in widespread use
- Clojure was developed by Rich Hickey, see his talk:

`www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey`

for some of the thinking behind his advocacy of it, in particular the relevance of immutability to multi-core architecture

- Part of the reason for the success of Clojure its implementation in Java Virtual Machine (JVM) code
- Functional style syntax ("lambdas") was introduced as an extra feature in Java in version 8 of the language, released in 2014
- Scala is another functional style JVM language now in common use in professional software development. See `recworks.co.uk/jobs/`

Sorting

- You will be introduced to several sorting algorithms
- They are introduced to help:
  - Get across the concept of an algorithm as distinct from the code which implements it
  - Get across the concept of there being more than one algorithm to solve a particular problem
  - Introduce the concept of formal analysis of efficiency
  - Get across the concept of generalised algorithms with incidental details provided by inheritance and parameterisation
- They are NOT introduced because you would need to write actual sorting code yourself in future

Java's API Code

- Java provides an extensive code library, including the Collections Framework which provides the data structures you need for standard programming
- Java's code library also provides sorting methods
- There is no need to write your own code if what you require is provided in a code library
- In more advanced programming you would be writing your own code libraries to supplement what is provided as a standard or available from third parties
- A "Java programming module" would concentrate on how to use its code library rather than the underlying principles in its implementation

ADSOOF Labs

- There are weekly lab exercises in ADSOOF
- They are set because you learn by doing
- Sometimes you learn best by encountering a problem in practice and learning the theory of how to deal with it later
- The exercises have a small mark which contributes to the module mark but is mainly to encourage attendance
- Do as much as you can of each week's exercise, but set a time limit (no more than 4 hours)
- Get the balance right between thinking through the exercise yourself and asking for help or further explanation
- Do the work YOURSELF, otherwise you learn nothing

Follow the pace

- Each lab exercise is given in one week and assessed in the following week
- Assessment is done by Demonstrators and is meant to be quick, based on demonstration and short discussion
- The idea is to keep you moving at the right pace through the module material
- The mark for each exercise is SMALL (0.0375% of your degree which is 1/2667th), so don't get worked up about it
- Do the work on the lab exercise in the week provided for it, do not leave it until the day it is assessed
- While waiting for assessment, start the new lab exercise

Asking questions

- I will be present in the labs for most of the allocated time
- Do not apologise for “disturbing me” by asking me question when I am there, that is what I am there for
- If I do not give a full answer to your question, it’s not because I’m being rude, it’s because I want to help you understand, so I may guide you to what you need to find out the full answer yourself
- Use the notes and experiment by writing code to get answers to questions if you can
- If you are stuck and cannot progress, then is the time to ask
- If you are unclear about what a question is asking, ask for clarification
- Use the module forum and email to ask questions before the assessment

Term-time tests

- There are two exam-style tests in the module, one in mid-term, one at the end of term, each counting 8% of the module mark
- These are NOT “exams” because the word “exam” means a formal exam timetabled and monitored by Queen Mary’s central administration. There is one actual exam for the module, it is in May.
- An important aspect of the tests is that they provide two-way feedback: I provide feedback to you on your attempts, but they provide feedback to me on how well you are understanding the material
- Pay careful consideration to the feedback given, it is one of the most important parts of your learning experience

Written exams and exam-style tests

- The exam and tests will be a mixture of written explanation questions and questions requiring hand-written code
- Having to write code by hand is a good test of deep understanding, which is why companies often use it to choose applicants in recruitment ("the code interview")
- No coding question will ever require an answer with a pattern markedly different from examples given in the notes and the lab exercises
- No explanation question will ever require material beyond what was taught in this module
- Diagrams and "bullet points" are fine in answers to explanation questions. An "essay" style is usually inappropriate and waffle should be avoided.

Learning v. Memorisation (recap)

- The idea that education is about memorising then "re-gurgitating" in exams is common
- It doesn't work in this subject
- If it has worked for you so far, now may be the point where it stops working
- It is better to understand the principles and from this reconstruct the details rather than try to memorise the details without understanding the principles

Jeff Atwood: Why can't programmers .. program?

- Jeff Atwood is one of the most well-known programming bloggers, and perhaps his most famous article is

`blog.codinghorror.com/why-cant-programmers-program/`

  where he comments on an observation that *"199 out of 200 applicants for every programming job can't program at all"*
- It was observations like this which led to the "code interview" becoming an essential part of recruitment of software developers
- It also led to scepticism about Computer Science degrees
- Supposedly knowing lots of facts, but being unable to apply them in practice is of little use
- Being able to program is more about basic skills than knowledge of particular programming languages or more advanced aspects
- Employers are also concerned about lack of "soft skills" in graduates

Mini-project

- Details of the mini-project will be released about a month before the end of term, and submission will be in the last week of term. It counts for 14% of the total module mark.
- A good habit is to plan your time, and make sure you have work in a state that could be submitted well before the actual deadline, giving you time to check it over at the end.
- The project will be a simple coding exercise, with marks equally for the code and the presentation.
- Presentation means well-written explanation, which shows an understanding of algorithmic principles, and also appropriate diagrams, and tables or graphs.
- Running experiments and collecting and presenting data, in this case timing figures, is an important aspect of general science, although it tends to feature less in Computer Science

Further Data Structures

- The mini-project topic will be chosen so that those who are confident with programming can do their own research on data structures beyond those covered in the module's teaching in order to obtain an efficient solution
- For those who are less confident with programming, solutions using simpler data structures which are taught directly can be used, and a good mark obtained for good presentation
- Doing your own research into data structures and implementing them yourself is a good way of learning them
- In the mini-project you are asked NOT to use Java API classes, the point is to gain an understanding of the principle of using a data structure to implement an abstract data type
- This is a learning exercise, otherwise when developing software it is usually better to use library code if it provides what you need (the mini-project topic would be trivial if solved with Java API classes)

Soft Skills

- "Soft skills" means ability in things like presentation, communication, team-work, problem-solving, time management, inter-personal relationships, critical analysis, etc
- That is, things that cannot be taught directly as academic or technical subjects, but can be developed indirectly through academic work, and are learnt through practice and experience
- ADSOOF does not cover team-work and inter-personal relationships, but they are a major aspect of your Software Engineering module
- Good use of English (including grammar and spelling) is an aspect of presentation skills that ALL graduates should have, graduates in technical subjects are wrong if they think it doesn't matter for them
- It is the other way round for basic problem-solving skills, and a logical way of thinking with basic mathematics skills

If you don't want to be a programmer …

- Although the emphasis in this presentation is on ADSOOF providing the skills needed for moving on to a job in software development, the soft skills it develops are applicable anywhere
- A logical approach to problem-solving, confidence with abstraction, the ability to read and write clear instructions applies not just to programming
- ADSOOF provides an understanding of what programming involves which is necessary in the many jobs that are not directly about writing code, but which involve interaction with people who write code
- An important aspect of ADSOOF for those who are coming to the conclusion that programming is not for them is perseverance
- If you find it challenging, do not give up, but carry on to get over the barriers of understanding, a pass is better than a fail
- Ask questions, use feedback, read around the subject, conduct your own experiments to develop a better understanding

So …

- This set of slides was put together in response to comments made on this module by students in previous years
- Many of these comments revealed a misunderstanding of the aims and objectives of ADSOOF
- It is a tough module, but the end results are worth it
- The emphasis is on the development of practical skills, and assessment which shows you really have those skills
- Feedback from employers is that a good grade in ADSOOF indicates someone who will perform well in a software development job
- Feedback from past ADSOOF students is that the relevance of ADSOOF became more apparent after they got jobs