A proposed new structure for teaching programming at A-level

**v0.1**  - 17th June 2022

Author’s note:

This is the first attempt at designing a fairly radical new approach to teaching programming at A-level. At this stage, the emphasis is not on *how* it should be taught, but on identifying what the pupil should *understand* and *know how to do*, and the approximate order in which they should acquire that understanding and skill.

The idea is that a pupil would be taught the whole syllabus through *one* of the standard multi-paradigm languages; Python (3+), C#, VB, Java, or Delphi (10+). They would not be expected to be able to read or write code in any other language, but they would be expected to have some minimal familiarity with the existence of other languages and how capabilities are similar and different across those. For example they should be taught the *difference* between static and dynamic typing, the pros/cons of each, and which their own chosen language is.

The most radical aspect is that they should:

* First, learn how to create and use *pure* functions as the fundamental building block, with no procedural coding at all and no input output, but including higher order functions and recursion, to the stage where they can write powerful functions as a single expression. Note that this section is named ‘Programming with Functions’ and not ‘Functional Programming’ – though the latter term is introduced as the final sub-heading, serving also to introduce the need for …
* Secondly, learn (or, for many, re-learn) how to use *procedures* as building blocks, primarily for the purpose of handling input/output and other system capabilities, but also showing where and how sequential statements may be used in pure functions where the language does not have some of the capabilities of a pure FP language such as Haskell (e.g. ‘let…in’)
* Thirdly, learn how to use objects, combined with both functions and procedures, *initially as custom data types/structures*, and only subsequently with the idea of encapsulated methods. [Five years ago I could not have imagined myself ever writing the previous sentence!]. When encapsulation is introduced the major benefit is portrayed as ‘polymorphism’ and the closely related idea of ‘type equivalence’ – one of the most important principles of programming that is barely covered today.
* The current approach (and content) of Data structures and algorithms, though the two are more closely related. There is much more emphasis on understanding these through programming – both from the ‘external’ (interface and use) perspective and an internal (implementation) perspectives.
* The final section on Programming Method & Technique, places far more emphasis than at present on the idea of *minimalist* iterations, rather than on the all-too-common idea that agile development as a series of smaller waterfalls. Emphasis is also placed on the difference between ‘good programming is anything that works’ and ‘good programming is a work of art’.

Contents

[Programming with functions 1](#_Toc106368088)

[Fundamentals: Expressions and types 1](#_Toc106368089)

[User-defined functions 4](#_Toc106368090)

[Collections, higher order functions & recursion 6](#_Toc106368091)

[The pure ‘functional programming’ paradigm 9](#_Toc106368092)

[Programming with procedures 11](#_Toc106368093)

[Fundamentals: statements and sequence 11](#_Toc106368094)

[Flow of control structures 12](#_Toc106368095)

[Functions vs. subroutines vs. procedures 12](#_Toc106368096)

[Role of procedural programming in the modern world 13](#_Toc106368097)

[Input/Output 13](#_Toc106368098)

[Programming with objects 15](#_Toc106368099)

[Fundamentals: used defined types, classes, and instances 15](#_Toc106368100)

[Properties and associations 15](#_Toc106368101)

[Methods 16](#_Toc106368102)

[Type equivalence and polymorphism 18](#_Toc106368103)

[The pure object-oriented programming paradigm 19](#_Toc106368104)

[Well-known data structures and algorithms 20](#_Toc106368105)

[Fundamentals 20](#_Toc106368106)

[Collections, sorting & searching 22](#_Toc106368107)

[Stacks and queues 23](#_Toc106368108)

[Graphs, trees, and associated algorithms 24](#_Toc106368109)

[Hashing, Hash tables, and dictionaries 25](#_Toc106368110)

[Programming method & technique 26](#_Toc106368111)

[Aspects of the software development process 26](#_Toc106368112)

[Agile development 27](#_Toc106368113)

[Best practices and principles 28](#_Toc106368114)

## Programming with functions

### Fundamentals: Expressions and types

#### Expressions

**Understand…**

* Expressions are made up of operations and values, which may be ‘literal’ or named ‘variables’, or, (in some languages) ‘constants’
* Expressions are ‘evaluated’ to get a result
* The basic arithmetical
* Brackets may be used to specify that a subexpression be evaluated first and its value then used in a larger expression.
* Operations also include pre-defined functions, which are named, and are called with one or more arguments to return a result.
* The idea of ‘pseudo random number’ generation in computers
* The result, and the values used may be of various different types including integer, real number, Boolean– if a function is given a value of an incompatible type this will produce an error.
* Values may sometimes be explicitly converted from one type to another, and that (in some languages) this conversation may sometimes occur automatically.

**Know how to…**

* Read and evaluate expressions manually
* Use standard mathematical functions, including: modulo, power, rounding, trigonometry functions, and generation of random numbers
* Construct and evaluate expressions in the chosen language using all ideas mentioned above

#### Different types

**Understand…**

* Instances of these types hold a single value
* The nature and applicability of the most common single-valued types: integer, real, Boolean, character
* That a string type is in some ways like a single valued type, but is actually an immutable reference type, and may in most cases be treated as a list of chars

**Know how to…**

* Use each of these types in the chosen language
* Convert data from one type to another where possible and appropriate
* Coerce a type where possible and appropriate

#### Logical & conditional expressions

**Understand…**

* The Boolean type
* Boolean operators: AND, OR and NOT
* Bitwise operators/functions (and how these differ from Boolean operators)
* Comparison operators (may work on other types but produce a Boolean result)
* Expressions may employ ‘selection’ (conditionality) through the use of an IF function (in some languages known as the ‘ternary operator’
* *Some* students may previously have learned to use IF *statements* (see Procedural programming). The primary difference of a conditional function to an IF statement is that the former always returns a value – and so may be used within an expression. Modern programming favours the use of conditional expressions over conditional statements wherever possible.)

**Know how to…**

* Use all the operators listed above in the chosen language
* Use the IF function (or ‘ternary operator’) in the chosen language, including nested conditions.

#### String expressions

**Understand…**

* A string represents a piece of text, may be thought of as a list of individual characters and in most programming languages may be treated that way
* Strings are an example of an immutable type – so even functions that might appear to mutate a string (such as converting a string to upper case) always return a new string that is a modified version of the input string.
* Regular expressions provide a succinct and powerful mechanism of searching strings for potential matches
* Regular expressions are defined in a purpose-defined language – known as a ‘regular language’ (see 2.5.1 Automata) – which may be thought of as a programming language but which is not ‘Turing complete’

**Know how to…**

* Use pre-defined string functions to change casing, extract a substring, or find the index of a matching substring
* Read and mentally simulate the function of a given simple regular expression
* Write a simple regular expression to match specified examples and test the execution of that regular expression within the chosen language.

#### Prefix, Infix & Postfix notations

**Understand…**

* That operators that require exactly two arguments may be written in ‘prefix’, ‘infix’ or ‘postfix’ notations, and that each has advantages and disadvantages both from the programmer’s and the computer’s perspective.
* That functions are typically represented in a prefix notation (name of function before arguments) but may sometimes be used in an ‘infix’ location using ‘dot syntax’: variable.functionName(arg).

**Know how to…**

* Convert expressions (that use only the simple arithmetic operators) from infix to postfix notation and back.

### User-defined functions

#### Defining and then using functions

**Understand…**

* The purpose of a function is to allow an expression to be easily re-used, with different values, and/or in multiple places within a program, or (if it is defined in a library) between different programs, without having to write the expression in full each time
* This is one of many examples of the ‘DRY (Don’t Repeat Yourself) principle’. Repetition of an expression, or of any piece of code, runs the risk that if it needs to be changed (due to an error, a change in requirement, or to improve performance, say) that the change might not be made in all places.
* A function is defined by a name, the values it must be provided with (its ‘parameters’), and what it returns, and the expression which uses those named parameter values to generate the result.
* The first three from the list above define the ‘signature’ of the function while the expression defines the ‘implementation’.
* Calling a function by its name, with arguments, which are copied into the parameters to be used by the expression.
* User defined functions may call other user-defined functions, just the same as pre-existing functions, and they may in turn be called by other functions.
* In this way functions may be thought of as building blocks in their own right and that programming consists of creating new building blocks of increasing sophistication but where the most complex ones are just as simple to *use* as the simplest.
* ‘Functional decomposition’ is this process working in reverse, where you start by representing the problem as a single function and then work out simpler functions that will make it easier to construct the topmost one, and so on.
* A function is said to be ‘overloaded’ if there are multiple functions with the same name, but each defining different parameters (number of parameters and/or their type)
* In some languages parameters may be defined as ‘optional’ meaning that the function may be called without supplying all the arguments – which is another way to achieve ‘overloading’. The implementation of the function must know how to handle these different situations.
* It is possible to return two (or more) values from a function using a ‘tuple’

**Know how to…**

* Define and use functions in the chosen language.
* Call those functions from within other code
* Decompose a requirement for a function into smaller functions that contribute part of the solution and are each somewhat simpler to design.
* Use tuples

#### Statically-typed vs. dynamically-typed languages

**Understand…**

* Difference when declaring and when calling functions
* Advantages and disadvantages
* Which of these the chosen programming language is.
* See also Type equivalence and polymorphism

#### Using and creating libraries

**Know how to**

* Make use of in-build libraries (including ‘importation’)
* Find and install publicly available libraries
* Turn functions that you have defined into a library that may be re-used in other programs.

#### Good practices in writing functions

**Know how to**

* Come up with appropriate names for functions, accurately, conveying their ‘intention’, and where a previous chosen name is not optimal, how to ‘safely’ rename a function (or parameter).
* ‘Refactor’ existing code such that repeated expressions or sub-expressions are moved into separate named functions.

### Collections, higher order functions & recursion

#### Collections

**Understand…**

* Hold multiple values that are in some way related
* Usually, it will be the case, these related values will be of the same type, though depending on the language this might or might not be enforced or enforceable.
* Concept of a collection. Historically these were two separate main types – an array and a ‘linked list’ - implemented very differently with different functional and performance advantages. In some modern languages they are still distinct types – though it is typically possible to convert data from one to the other – while in some languages a single type offers the functional features and performance advantages of both.
* When a collection is, *or behaves like*, an array the elements may be accessed by an index.
* Collections may be ‘static’ – where the size (number of elements) – is specified when the collection is declared, or ‘dynamic’ which can expand as needed. (Historically, arrays were static, and lists were dynamic, but modern programming languages do not necessarily make this distinction).
* It is possible to define a collection as having two or more ‘dimensions’. Depending upon the language this might be achieved by a multi-valued index (like coordinates in 2D or 3D space) and/or by defining a collection where each element in the collection is another collection.
* An Immutable list may conveniently be implemented as a recursive data structure, where a list consists of single element (the ‘head’) and an optional reference to another list (the ‘tail’).

**Know how to…**

* Use an array and a list in the chosen language, or – where they are not distinct – be able to retrieve/set/modify an element by its index and/or be able to append, delete and insert elements expanding/collapsing the size automatically.
* Implement and use a 2-dimensional collection

#### Higher order functions

**Understand…**

* If a programming language treats ‘functions as first-class objects’ then a function may be held in a variable, or may be passed into another function as an argument, or returned as the result of another function. A function that takes in another function, or returns a function is known as a ‘higher order function’ (HoF)
* A lambda may be thought of as a function that has no name (‘anonymous’) that is intended for use in a single context (not to be re-used in multiple places) and is defined ‘in-line’ where it is needed – most commonly as an argument passed into a HoF.
* The HoF pattern is one technique for making ‘generalising’ (some say ‘genericising’) a function: making it applicable to a wider range of situations. It means that you can write a function where a small but important element within the algorithm can be customised – supplied as a sub-function to be passed in when the function is called.
* One common use of HOFs is to process lists – providing a way to apply the function that is supplied as an argument to each member of the list. Common examples of such functions include map, filter, and fold (though these three generic types of HoF may have different names in different languages)
* A map HoF returns a new list where each member is the result of applying the function supplied as an argument to a member of the original list.
* A filter HoF returns a new list containing only the members of the original list that match criteria – which are specified by a function supplied as an argument
* A fold HoF returns a single object that is the result of combining each member of the original list in a manner specified by the function supplied as an example. Reduce may be applied to calculate a sum or average, or to concatenate elements into a single string.

**Know how to…**

* Use the map, filter, fold functions (or their equivalents in the chosen language) singly and in combination to manipulate lists
* Use the fold function to implement a mathematical function defined by a ‘Taylor series’.
* In each case, be able to construct a lambda, or to specify a re-usable function to pass to provide the function argument needed by those HoFs
* Be able to define and use a simple HoF of your own, delegating the implementation to one or more of map, filter, reduce. (See also, Collections, sorting & searching)

#### Recursive functions

**Understand…**

* If you need to implement an algorithm that involves ‘iteration’ and that cannot be handled (or not conveniently) by combining standard HoFs then recursion is the way to do this. (see )
* In a recursive function the expression (or ‘body’) of the function will involve at least one call to the function itself, but passing in arguments that differ in some way from the arguments passed in when the function was called
* A recursive function must always test for a termination condition which, if passed, halts the process of calling the same function recursively.
* Recursive functions that iterate through a list typically do so by processing the ‘head’ (first item in the list) and then calling the same function recursively with the ‘tail’ (a list made from the original list without the first element). The list may be implemented internally as a head and tail, or it may be implemented as a conventional list/array but provide functions for accessing the head and tail – returning a single element and another list respectively.
* The role and operation of the system stack in the execution of both nested function calls in general and recursive functions in particular, and why a missing (or incorrectly written) termination condition will lead to a ‘stack overflow error’
* While Head and Tail functions may exist for any kind of list, but it is also possible to design a form of list where the internal structure directly reflects the Head/Tail idea. This is another example of the principle of ‘separating the interface from the implementation’.

**Know how to…**

* Define and use a recursive function both to apply to the same data repeatedly until a result is achieved, or to apply to each member of a list.
* Obtain the head and tail of a list using the appropriate functions available in the chosen language.
* Observe the operation of the system stack in debugging mode and use this to debug an recursive function that is not working correctly

### The pure ‘functional programming’ paradigm

#### Nature and advantages of the pure FP paradigm

**Understand…**

* Pure FP means writing programs entirely using ‘pure’ functions.
* Languages, such as Haskell, purpose-designed for FP enforce the rules about pure functions. They usually offer very terse syntax also.

**Know how to…**

* In code written in a langage that isn’t pure FP, how to recognise functions that do not pass all the rules for a pure function.
* For simple examples, refactor an existing function so that as much as possible of its logic is contained in pure functions.

#### Some traps to avoid when writing pure functions

**Understand…**

* The difference between ‘value types’ (most single-valued types) and ‘reference types’ (most data structures) – and how they are passed as arguments into functions. Where a reference type is ‘mutable’ (this includes most lists/arrays, but not, typically, strings), then any change made to the contents *inside a function*, may be visible outside the function, for example if an external variable holds a reference to the same instance. This would be a ‘side effect’ in FP terms. Therefore care must be taken not to make such changed – but to always return a new instance included in the result.
* The risk of violating the above rule may be eliminated by using ‘immutable’ data structures
* Random numbers pose a specific challenge for FP because calling Next (or equivalent) in most languages mutates the state of the random number generator, which is a side effect. However, it is possible to implement random number generation in pure FP, with a function that generates the next number from ‘seed values’ passed in and returns a new set of seed values with the generated number.

#### The challenge of input/output in FP

**Understand…**

* FP requires that all functions are ‘side-effect free’, but any form of input/output is usually considered to be a side effect
* Pure FP languages such as Haskell get around this conundrum by a sophisticated mechanism whereby any function that needs I/O effectively takes in the ‘state of the world’ as an input and returns a state of the world as part of its result. The language enforces that any function calling such an I/O function (also known as an ‘action’) must also include the state of the world in its signature.
* Consequently, in a pure FP system, the ‘core’ of the system consists of pure functions, and the outer edges consist of actions. This pattern, an example of the design principle of ‘separation of concerns’ is considered a good practice even for systems that are not attempting to be pure FP.
* ‘Actions’ in Haskell may also implement a form of sequencing to specify interaction between a user and the system.
* In other words, the code implementing an *action* in a pure FP system may look rather like ‘procedural programming’, even if in strict mathematical terms it is still a single expression.
* When writing a system in most ‘mixed paradigm languages’ such as [the chosen language for this syllabus’, it is convenient, and in many cases necessary, to use procedural programming techniques for managing the input/output. This code then calls into the core functionality of the application, which may be, and where possible should be, implemented as pure functions.

## Programming with procedures

### Fundamentals: statements and sequence

**Understand…**

* Procedural programming is also known as ‘imperative programming’, whereas functional programming is also known as ‘declarative’ programming. In the latter, you declare *what* the computer needs to know in order to solve the problem, but do not spell out the order in which this knowledge must be applied. In imperative programming you spell out each step in the process as an explicit ‘sequence’.
* Each step will typically consist of a ‘statement’. The difference between a statement and an expression is that a statement does not ‘return’ anything – it is a self-contained operation that results in a change to the state of the system.
* An ‘assignment statement’ evaluates an expression and assigns the result to a new, or an existing, ‘variable’. (It does not *return* the value – it ‘consumes’ the result of evaluating the expression).
* Other types of statement include outputting a result to the system – the simplest example being to print or write to the console – or to read input from the user or another part of the computer.
* A sequence of statements that execute in a fixed order form a ‘block’

**Know how to…**

* Assign values to new and existing variables, specifying the type where needed (for a statically typed language)
* Split up a complex expression into a sequence of assignment statements and, conversely, to re-write a sequence of assignments into a single assignment statement using a more complex expression.

### Flow of control structures

**Understand…**

* ‘Flow of control’ structures change the default (sequential) order of execution of statements
* In procedural programming, ‘selection’ (conditional execution) is handled by branching, using if then else elseif structures.
* Be aware that some languages support a switch-case statements (or equivalent) as an alternative means for handle a multi-way branch.
* ‘Iteration’ is achieved through the use of loop structures, of which a language may offer several variants such as for, while, and repeat-until
* Anything that can be written using any of these loop structures *may also* be written as a recursive function, and many *may* be implement using higher-order functions that iterate over a list. However, if the loop contains input/output statements it will not be a pure function, so there is not necessarily an advantage to adopting either of those options.

**Know how to…**

* Read, trace through, and write ‘chained’ and ‘nested’ selection statements in the chosen language, including using the switch…case statement if available.
* Read, trace through, and write all of the principle loop structures offered in the chosen language.

### Functions vs. subroutines vs. procedures

**Understand…**

* Procedural programming may involve both calling pre-defined functions and writing new ones. However, generally, procedural programming adopts a wider, less rigorous, meaning to the word ‘function’ than in FP – for example not enforcing the requirement that a ‘function’ must return a result, nor that it takes one or more arguments, nor that it may not generate side effects nor depend on things not passed in as arguments.
* In some languages, these less-rigorous forms of ‘function’ are referred to as ‘subroutines’ or ‘procedures’, but there is no universal agreement concerning the strict definitions of those terms beyond the fact that each defines a capability that may be called from several different places and that upon completion, control is passed back to where the capability was called from.

### Role of procedural programming in the modern world

**Understand…**

* An entire program can be written procedurally. For many years this was the only available approach to programming. Pupils may previously have learned to program this way.
* Unless you are using a pure FP language, you will need to use procedural programming patterns to handle the input output.
* If you aren’t using a pure FP language (with ‘let … in’) then breaking out a repeated sub-expression into a separate assignment can make code more readable and efficient. As long as you are not re-assigning a variable, this is not breaking the purity of the function.
* You can even use loops/branches when writing a pure function – as long as none of variables that are being mutated or re-assigned in the code are visible outside the function.

### Input/Output

#### File access

**Understand…**

* Uses for files within programs and their limitations
* The difference between ‘text’ and ‘binary files’
* Limitations of file-based persistence and link to 3.2 Databases

**Know how to…**

* Write code to read from and write to a file both as binary and text form
* For text files, design and implement an appropriate format for saving structured data

#### Console interaction

TBD

#### GUI interaction

TBD

#### Exception handling

**Understand…**

* The system may generate ‘throw’ an exception when something does not execute as expected
* Exceptions are commonly thrown by input/output actions, for example when the specified file name or URL cannot be found, or where an input value does not match the expected or required format.
* Exceptions can also be thrown where there is no I/O when a piece of code cannot be executed, for example when a number is divided by zero, where a reference is null (see Difference between reference types vs value types) , or there is a ‘stack overflow error’
* By default a throw exception will cause the program to halt with an error message. It is usually possible to determine where the exception was thrown by examining the System stack
* Where a certain kind of exception might be expected to be thrown in normal execution, it may be explicitly ‘caught’ by special code and, where appropriate, the program then continues – for example after advising the use that the value they entered did not match the specified criteria and asking for a new value.
* The programmer may also explicitly throw an exception where a certain condition is recognised, either as an aid to debugging when the program halts, or with the intent that it be caught by whatever ‘called’ the code currently being executed.
* That it is good practice to write ‘safe’ code that catches and handles all exceptions that might *reasonably*  be expected to be thrown in the course of normal operation.
* That it is better, where possible, to write code that ‘guards’ against the possibility of exceptions (for example testing that the divisor is not zero, or that a reference type might be null) than to catch exceptions – and why.

**Know how to…**

* Write a catch statement to intercept an exception that may be thrown by an input/output action, test the type of the exception thrown and then continue execution in an appropriate manner.
* Throw an exception within code to help in debugging.
* Identify opportunities to guard against certain conditions instead of catching an exception.

## Programming with objects

### Fundamentals: used defined types, classes, and instances

**Understand…**

* Custom (or ‘user-defined’) data types give more expressive power to programmers by allowing them to bring together unlimited amounts of related data elements (of the same or different types) into a single instance that in many cases corresponds to a real thing or concept in the application domain
* The most common form of custom data type is called an object. The structure of an object is defined in a ‘class’ – which may be thought of as a template, from which ‘instances’ may be created by calling a class ‘constructor’. The constructor may be explicitly defined – and specifies the minimum mandatory set of data that must be provided to create an instance. Referring to the ‘type’ of an instance means the same thing as referring to its ‘class’.
* In modern programming languages many of the in-build types are themselves defined by classes – though that definition is not necessarily visible to the application programmer using the objects.
* In most modern languages, objects are ‘first class’ – meaning that they may be held in variables, passed as arguments into functions, or returned as the result of an expression or function.

### Properties and associations

* An object will typically have multiple named ‘properties’ each of which holds a value – which may be a single value type, a data structure, or another object (of the same or different type/class).
* Properties may be specified as ‘read only’ (the value must then be specified when constructing the instance) or also ‘writeable’. A class where all properties are read only therefore defines an ‘immutable type’.
* Where a property holds an instance of another object class, or a data structure (e.g. a List) of other instances, it is sometimes described as defining an ‘association’ between objects

### Methods

**Understand…**

* A class may also define functions that operate on a given instanceof that class. Functions defined in this way are sometimes called ‘methods’ to distinguish them from free-standing functions – but some sources use the terms ‘function’ and ‘method’ interchangeably. (Pupils should recognise both terms, but will not be penalised for using them interchangeably).
* These methods are said to be ‘encapsulated’ in the object, together with the values of its properties. Both properties and methods are said to be ‘members’ of the object. A useful way to look at it is that the properties together define the ‘state’ of the object, and the methods define its ‘behaviours’.
* A method might mutate (change) one or more of the properties of the instance – provided that the class definition permits some or all of its properties to be mutated (written to) – and/or it might return new values retrieved or derived from one or more of the properties of the instance. Ob
* The advantages of encapsulating the methods on the objects, rather than implementing them as standalone functions include (depending on the capabilities of the programming language):
  + *Distribution*. It is convenient for purposes of re-use to package the definition of the type with the functions (‘methods’) that apply to it. Note, however, that it may also be possible to define a class and *standalone* functions that relate to that type within one code-file, so this advantage of encapsulation is not huge.
  + *Readability*. It is possible to invoke an object’s encapsulated methods using ‘dot syntax’ (see Prefix, infix, and postfix notation for operations). Note, however, that some languages make it possible to define standalone ‘extension methods/functions’ which may be called using dot syntax on an object of the type defined for the first parameter. So again, this is not necessarily dependent upon encapsulation.
  + *Prompting*. For some languages and IDEs it is possible to type a dot after a variable name and be prompted with a drop-down list (refined by auto-completion) of the methods available for that object.
  + *Information hiding*. In some languages it is possible to define members that are visible to methods defined on the same class, but not to code defined outside the class. This makes it possible to enforce rules to ensure the integrity of the date. It is also another example of the design principle of ‘separating the interface from the implementation’. For instance of a DateTime class give the *appearance* (from their public interface) of having separate properties for year, month, day, hour, minute, and second, when *internally* there is just a single integer holding the number of ‘tick’' (hundredths of a second) from some fixed start date (‘epoch’) and the apparent properties are all derived from this single number.
  + Polymorphism. This is the most powerful advantage of encapsulation. See 1.2.5 Type equivalence and polymorphism.

**Know how to…**

* Define methods on a class, and invoke methods on an instance of a class, in the chosen language
* In an appropriate situation, and as permitted by the language, define methods that ‘hide’ the underlying properties, offering more value to the code using the object and enforcing rules.

### Type equivalence and polymorphism

**Understand…**

* Type equivalence means that instances of two or more different types may be passed in as a given argument to a given function, and work correctly.
* The closely-related idea of polymorphism is where two or more classes define a method with an identical signature – but where the implementation may be the same or different – such that external code may invoke the method without knowing the specific type of the instance it is being called on.
* Both concepts contribute to the re-usability of code by making it applicable to a range of different types.
* Type equivalence and polymorphism may be achieved through several different mechanisms, depending upon the language. For example:
  + With dynamic typing (see Statically-typed vs. dynamically-typed languages), type equivalence and polymorphism are *automatic*. You may call any method on any variable – but you will get an error at run-time if that type does not have a matching method. Similarly, you may pass an instance of any type as any argument to a function and the function will work correctly *unless the implementation of the function uses an operation or method that will not work for that type.*
  + With ‘static typing’ types must be explicitly defined as being equivalent, either by making them ‘inherit’ from a common superclass or ‘implement’ a common ‘interface’. The type of a parameter required by an external function may then be specified as that common superclass or interface. The superclass or interface should also define any common members that all its sub-classes (or, for an interface, its ‘implementations’) must provide.
  + A superclass, as well as defining common members for which sub-classes must provide their own implementation (known as ‘abstract members’), may also define concrete members that are automatically inherited by the sub-classes without re-defining them – though, in some languages it is also possible to specify that a subclass may ‘override’ this inherited capability and provide its own new or modified implementation, matching the same signature.

**Know how to…**

* Identify and explain actual use of type equivalence within the chosen language
* Define a method with the same signature but different implementations on two different types such that the method may be called polymorphically, and write external code that can use this
* Define a method such that its implementation is inherited by sub-classes
* Override the implementation of an inherited method

### The pure object-oriented programming paradigm

**Understand,,,**

* Programming in the pure ‘OOP paradigm’ implies ‘thinking entirely in objects’. Typically design starts by identifying the set of objects that best captures the entire state of the system, and implementing all ‘behaviour’ (functionality) as methods encapsulated on the most appropriate of the objects.
* The methods may be written using procedural patterns, but in a pure OOP system, there are no procedures *outside* of any object.
* Like the other paradigms, in theory any requirement could be met using the pure OOP approach. However, it seems best suited – and is certainly most popular – for applications that involve ‘modelling’ a real-world (or sometimes an imaginary-world) phenomenon. These include games, graphical user interface frameworks, drawing or design applications (including CAD0 , scientific, engineering, and financial simulations, and also business systems built around an existing ‘record based’ information model. In the latter case the object model may structurally resemble the entity-relationship model used for a relational database, and indeed in some cases one is derived entirely automatically from the other and/or linked together by a generic piece of systems software called an object-relational mapper.
* The advantage of the pure OOP approach for these kind of applications is that the structure and constructs of the code very closely resembles the way that people naturally think or talk about the real world problem. So a new requirement, or change can be more easily translated into the necessary code changes.
* In a pure OOP system, the user interface may also be ‘object-oriented’ – where the users interactions follow a ‘noun-verb’ style: selecting an icon, say, that represents a noun in the problem domain, and then enacting one of the verbs that is (or appears to be) a property of that noun – either by moving it, or moving one of the ‘handles’ that it offers when selected, or by selecting from a pop-up menu. This is most obvious in the ‘desktop’ view of modern computers, or when using drawing programs that present a menu of shapes to be dragged and dropped. By contrast, many traditional systems adopt a ‘verb-noun’ style of interaction where the user must first follow a series of menus to find the ‘action’ required and then specify the data that the action is applied to.

**Know how to…**

* Show experience of having thought through a whole problem using the pure OOP approach, and implementing this as working code.
* Identify whether a user interface is primarily ‘verb-noun’ or ‘noun-verb’ (object-oriented UI), whether or not it has been written using OOP.

## Well-known data structures and algorithms

### Fundamentals

#### Relationships between data structures and algorithms

**Understand…**

* Broadly speaking: ‘Programming = Data Structures + Algorithms’
* Data structures may be custom (see Fundamentals: used defined types, classes, and instances) or they may be general purpose. Where general purpose data structures are defined in terms not specific to any language/platform, and where many implementations exist, they are sometimes described as ‘abstract data structures’
* Any function or operation implements ‘an algorithm’ – even an addition or multiplication operation. The algorithm is the abstract representation of how to go about implementing the operation. Algorithms may be custom designed to solve a specific problem or they may be ‘well-known’ algorithms that can be applied to a broad range of problems
* Algorithms do not necessarily involve any data structures – they might apply to processing single values (for example finding the HCF of two numbers) – but many of the more interesting algorithms are applied to data structures and/or use data structures internally for their implementation.
* To solve a given problem there may often be a trade-off between the choice of data structure and algorithm: using a more sophisticated data structure may allow a simpler algorithm, or vice versa.

**Know how to…**

* Modify a simple example to use a more sophisticated data structure and hence reduce the complexity of the algorithm

#### Analysing algorithm complexity

**Understand…**

* The same function may be implemented by different algorithms (and/or different data structures), delivering the same result, but with different efficiency in terms of the processing time and/or memory usage.
* With modern computer performance much of the time the programmer need not be concerned with the efficiency of operations, but when handling problems of larger scale (processing large data items, large numbers of data items, and/or to very high precision) efficiency can be a concern.
* Efficiency may be compared using the ‘Big O’ function. The Big O function does not calculate the *absolute* processing time nor memory usage, but the relative rate at which that measure increases with the size of the problem.
* Common Big O results include: O(1) – time/memory is constant in relation to the problem size, O(n) – time/memory grows in proportioAn to the problem size, etc.
* The Big O may be determined by how the number of iterations of any loop, recursive calls, or invocation of a standard HoF on a list, varies with the size of the input to the operation.

**Know how to…**

* Determine the Big O function for a fairly simple algorithm represented in the chosen language
* Draw a not-to scale plot that shows the relative rate of growth of the most common Big O functions

### Collections, sorting & searching

#### External view – functions, performance, applicability

**Understand…**

* That accessing an element via an index is O(1) – it takes the same time to access any element, and irrespective of the size of the collection

**Know how to…**

* Access an element by index in the chosen language.

#### Internal view

**Understand…**

* *How* an indexed element is retrieved and why this is a O(1) operation

#### Related algorithms

**Understand…**

* Bubble sort is a simple algorithm to explain and implement, but is inefficient
* Merge sort is more efficient than bubble sort. It may be conveniently described using a recursive approach
* Linear search is a straightforward way to find the first matching item (or none) in a list/array and will work for any data
* Binary search is more efficient that linear search, but will work only on a list/array that is already sorted.
* A binary search of an ordered collection has similar performance to searching a ‘binary search tree (see Graphs, trees, and associated algorithms) – but the latter offers additional advantages.
* That any sort or search algorithm can be implemented as a HoF (see Higher order functions ), where the mechanism by which elements are compared is specified by the passed in function. Many languages have both sort (e.g. OrderBy) and search (e.g. FindBy) HoFs.

**Know how to…**

* Implement each of the specified algorithms in the chosen language
* Evaluate and explain the Big O function for each of these algorithms

### Stacks and queues

#### External view – functions, performance, applicability

**Understand…**

* A stack is a last-in-first-out (LIFO) mechanism.
* Most programming languages provide Stack class, with Push, Pop, and Peek methods encapsulated. Depending upon the language/library a stack may be fixed in size, or may be dynamically expandable.
* Many standard algorithms make convenient use of a stack, internally and temporarily, including for example: reversing a string, evaluating an expression (see Graphs, trees, and associated algorithms).
* Most computers use a dedicated ‘system stack’, implemented in hardware which is used by the operating system and by compiled programs; it is not usual to make direct use of the system stack for application purposes . This may be made visible to the user when debugging a program, but application programs
* A queue is a first-in-first-out (FIFO) mechanism.
* Most programming languages provide Queue class that may be instantiated, with Enqueue, Dequeue, and Peek methods encapsulated. Depending upon the language/library a stack may be fixed in size, or may be dynamically expandable.
* Many standard algorithms make convenient use of a queue, internally and temporarily (see Graphs, trees, and associated algorithms).
* The operating system on most computers makes use of queues to ‘buffer’ information being sent/received via a network or other peripheral devices – in order to allow for differences or variations in the speed at which data is processed by the other device.

**Know how to…**

* Instantiate and use Stack and a Queue objects in the chosen languages
* Make use of a stack or queue appropriately in implementing a simple specified algorithm

#### Internal view

**Understand…**

* In the absence of a ready-made class, both stack and queue may be implemented using an array, together with ‘pointers’ to locations within that array and functions. This may be defined as a class, with encapsulated methods, or as standalone functions. (If these are to be pure functions then the array and the pointers must be passed in an out of the function along with the data item being added or removed from the structure).

**Know how to…**

* Write a class that implements the concept of a stack or a queue, with properties for the underlying collection and the pointers, and methods for standard operations.

### Graphs, trees, and associated algorithms

#### External view – functions, performance, applicability

**Understand…**

* The concept of a graph, including nodes and edges and what types those might take
* What a graph might be used to represent
* Different categorizations of graph including connected/dis-connected, fully connected or partially connected, directional or non-directional.
* The relationship of a tree to a graph, and what a tree might be used to represent
* The additional constraints imposed by a binary tree, and a binary search tree, what these might be used to represent, and the advantages of using them.
* The role and nature of optimisation algorithms operating on a graph

**Know how to…**

* Instantiate and use the ready-built Graph and Tree classes in the chosen language.

#### Internal view

**Understand…**

* How a graph may be implemented from more basic constructs using an adjacency matrix or list
* How a tree may be implemented as a graph, but may also be implemented as a recursive data structure
* How items are added to or removed from a graph or tree

**Know how to…**

* Show experience of having implemented a Graph class using only more basic constructs.

#### Related algorithms

**Understand…**

* Simple tree-traversal algorithms
* Simple graph-traversal algorithms
* Dijkstra’s shortest path algorithm

**Know how to…**

* Perform the above algorithms manually on a simple graph/tree
* Have experience of having implemented at least one of these algorithms

### Hashing, Hash tables, and dictionaries

#### External view – functions, performance, applicability

**Understand…**

* A hash table (also known as a ‘hash set’ or simply a ‘set’ in some languages) is a collection that does not permit duplicate entries.
* It can be used to perform mathematical ‘set’ operations such as union and intersection.
* A hash table also offers a Contains function/method with O(1) performance, which has many useful applications, for example checking whether a typed word is a recognised spelling.
* A dictionary offers the ability to associate any value with any key and retrieve it from just the key. Unlike an array, the keys do not have to be integers, or, if they are integers, do not have to be contiguous.

**Know how to…**

#### Internal view

**Understand…**

* The role played by the hashing algorithm, how that is used by a hashtable to prevent duplicate entries, and how it enables O(1) performance for the Contains method
* The role played by the hashing algorithm in a dictionary
* For both hashtable and dictionary, what happens when two different values (for a hashtable; two keys for a dictionary) both hash to the same result – known as contention – and how this is handled.

#### Related algorithms

**Understand…**

* How a hashing algorithm works
* Other uses of hashing algorithms in CS, including secure storage of passwords, and testing equality of two objects

**Know how to…**

* Use a ready-made hashing algorithm such as SHA in the chosen language.

## Programming method & technique

### Aspects of the software development process

**Understand…**

* **Specifying what is required**. Sometimes known as ‘[problem] analysis’. For a business application, this may entail interviews with potential users, customers, or other ‘stakeholders’. For commercial products it might entail extensive research into existing available products that compete with, or may need to connect with, the intended application. Or it might be that the intended application is an original idea of the developer, in which case it is still good practice to start – as far as is possible – with a clear specification (both from the perspective of a user and from a technical perspective) what you are trying to develop.
* **Designing and coding the solution**. For some systems the process of translating requirements into working code can be fairly direct, for others there is a need to ‘design’ elements of the solution – perhaps the ‘entity model’ for the database and/or the ‘object model’ for the code, or major components of the system and the interfaces between them.
* **Testing**. Testing of the system by potential users –to check that it meets their requirements, works correctly, and is easy enough to learn and use – is only the *last* stage of testing. Testing should take place as early and as often as possible throughout development. Every single function or method, for example, can and should be tested in isolation (this is known as ‘unit’ testing). Broadly speaking, tests should cover three things:
  + Normal conditions, sometimes known as the ‘happy case’
  + ‘Edge’ or ‘boundary’ conditions – because most errors in coding occur at the boundaries, including the common ‘off by one errors’ that are often found in iteration, indexing, or comparisons.
  + Error conditions. Anticipating things that can and do go wrong in normal operation such as a user missing an entry, a conflict when updating a file or database, or the network being down.
* **Evaluation**. As with any project, it is important to evaluate, as objectively as possible, both the quality of what has been delivered and the efficiency and effectiveness of the process, both in order to feedback learnings into the next phase of development or the next development project.

**Know how to…**

* Read a short document describing an existing activity, and identify from that the requirements or rules that may be relevant to the system
* Translate such a requirement into code, identifying where it is appropriate to undertake some design activity prior to coding, and being able to undertake and capture that design separately from the code
* Write suitable test cases for an individual function, or for a system from the user’s perspective
* Be able to code and run automated unit tests for a function
* Be able to evaluate a simple system critically

### Agile development

**Understand…**

* Historically, systems were typically developed in a predominantly linear fashion, starting with problem analysis and requirements specification, followed by solution design, coding, and testing. Iteration within those phases was principally only to deal with errors arising. The project was complete when the specified requirements were successfully met, and any feedback resulting from the final evaluation was relevant only to the next project.
* Some systems are still, necessarily, built this way, for example large systems dominated by engineering considerations. For most business systems, however, this linear process is no longer considered acceptable: because business requirements change frequently; because users find it very difficult to fully specify the requirements of a system they have never used, and because a new capability often trigger brand new ideas for requirements that could not have been foreseen. This has given rise to a more ‘agile’ approach to systems development.
* There are many so-called ‘agile methodologies’ which each prescribes specific approaches or techniques, but all are committed to the idea of short iterations, and to willingness to change anything that has already been done if necessary or beneficial. Because of the latter, there is a high commitment to *automated*  testing, so there is an immediate alert whether any small change has broken things that were not intended to have been changed.
* To some people, ‘iterative development’ just means breaking up an intended project into smaller sub-projects. However, at its most extreme, an ‘iteration’ may be interpreted as ‘the *smallest* addition or change to the functionality of the system that is capable of being implemented to the point where the system works and may be evaluated by a user’.

**Know how to…**

* From a broader description of a requirement, identify the *smallest* iteration that would deliver something that would be recognised by the user as contributing value and be capable of testing and evaluation. Be able to implement and test that, and then identify the *next* smallest increment, and so on.
* Where a new requirement does not fit with what has been implemented already, or where it is recognised that *had the new requirement been known up front, the design would probably have been different*, know how to ‘refactor’ the design, and check that all existing tests still pass, such as to make the implementation of the new requirement more straightforward.

### Best practices and principles

**Understand…**

* The design principle of ‘separating the interface from the implementation’ and the way this principle may be observed/applied at multiple levels, including functions, objects, and ‘APIs’
* The design principle of ‘separation of concerns’ and in particular its application to separating code concerned with Input/Output (or other ‘systems’ capabilities) from code concerned with implementing application logic. Where possible, the Input/Output system should call the core functionality, not *vice versa*; where that is not possible, input output functions should be passed into core functions so that the core function does not depend on any specific implementation of I/O.
* The DRY principle – Don’t Repeat Yourself and the way that this can be applied at many different levels
* Keeping functions small. A pure function should be implemented as a single expression, unless this would force the use of repeated code. A procedural ‘function’ should never contain more lines of code than may be viewed easily on one screen and should not contain more than two levels of nesting (usually revealed by indentation) *within the body of the function.*
* The coding principle of ‘intentionality’ – writing code such that the *intention* behind the code (whether it is working correctly or not) is as clear as possible to someone else reading the code. The role of naming (functions, parameters, classes) in intentionality, and the principle of using comments *only* to explain things that cannot be made clear, intentional, and completely documented, by the code itself (including understanding the risks of writing anything else in a comment!)

**Know how to…**

* Recognise violations of the principles listed above and suggest ways of improving the design and/or code.
* **Use a code repository** [recommended] to commit your code at each iteration; to identify code ‘differences’ between commits, to discard changes made since the last commit (if not working out), or revert to a prior commit if necessary.