Topic: 4.1.1 Abstraction

What is Computational Thinking?

Computational thinking (CT) involves a set of problem-solving skills and techniques that software engineers use to write programs that underlie the computer applications you use such as search, email, and maps.

There are many different techniques today that software engineers use for CT such as:

- Decomposition: Breaking a task or problem into steps or parts.
- Pattern Recognition: Make predictions and models to test.
- Pattern Generalization and Abstraction: Discover the laws, or principles that cause these patterns.
- Algorithm Design: Develop the instructions to solve similar problems and repeat the process.

Decomposition

Part of being a computer scientist is breaking down a big problem into the smaller problems that make it up. If you can break down a big problem into smaller problems, then you can give them to a computer to solve. For example, if I gave you a cake and asked you to back me another one, you might struggle. But if you watched me making the cake and worked out the ingredients, then you'd stand a much better chance of replicating it. If you can look at a problem and work out the main steps of that problem, then you'll stand a much better chance of solving it.

Look at an example, the equation to work out the roots of a quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

On first look, it might appear a little scary, but if we decompose it, we should stand a better chance of solving it:

1. b^2
2. $4ac$
3. $b^2 - 4ac$
4. $\sqrt{b^2 - 4ac}$
5. $-b + \sqrt{b^2 - 4ac}$
6. $2a$
7. $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$
8. Repeat for $-b - \sqrt{b^2 - 4ac}$

By noting the steps down to solve a problem, we can often recognize patterns, and by giving a list of steps, we are one step closer to creating an algorithm.

Topic: 4.1.1 Abstraction

Pattern recognition

Often breaking down a problem into its components is a little harder than taking apart an algorithm, we are often given a set of raw data and then are asked to find the pattern behind it:

$$1, 4, 7, 10, 13, 16, 19, 22, 25, \ldots$$

This is pretty easy with number sets, the above pattern $A_n = n + 3$. But pattern recognition might also involve recognizing shapes, sounds or images. If your camera highlights faces when you point it at some friends, then it is recognizing the pattern of a face in a picture.

The face was automatically detected by pattern recognition

If your phone tells you the weather when you ask it "What is the weather like in Karachi?", then it has recognized the word "weather" and that "Karachi" is a big city in Pakistan. Linking them together, pattern recognition is the computing behind why you are given tailored adverts when you log into your mail account or social network, they have recognized the pattern of what someone like you wants to buy. Pattern recognition might predict the weather, but prediction may not always be perfect.

Pattern generalization and abstraction

Once we have recognized patterns, we need to put it in its simplest terms so that it can be used whenever we need to use it. For example, if you were studying the patterns of how people speak, we might notice that all proper English sentences have a subject and a predicate.

Algorithm design

Once we have our patterns and abstractions, we can start to write the steps that a computer can use to solve the problem. We do this by creating **Algorithms**. Algorithms aren't computer code, but are independent instructions that could be turned into computer code. We often write these independent instructions as **pseudo code**. Examples of algorithms could be to describe orbit of the moon, the steps involved in setting up a new online shopping account or the sequences of tasks involved for a robot to build a new car.

Topic: 4.1.1 Abstraction

Limits?

Nowadays computers are ubiquitous and some would argue that there are no problems out there that a computer, given enough time, couldn't solve. But it this true? Is every problem solvable by a machine and can we ever know if this is the case?

What is Abstraction?

Abstraction is a technique for managing complexity of computer systems. It works by establishing a level of complexity on which a person interacts with the system, suppressing the more complex details below the current level.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called the abstraction principle.

Topic: 4.1.1 Abstraction

Functions and Procedures

Subroutine

A subroutine is a self-contained section of program code that performs a specific task, as part of the main program.

Procedure

A procedure is a subroutine that performs a specific task without returning a value to the part of the program from which it was called.

Function

A function is a subroutine that performs a specific task and returns a value to the part of the program from which it was called.

Note that a function is 'called' by writing it on the right hand side of an assignment statement.

Parameter

A parameter is a value that is 'received' in a subroutine (procedure or function).

The subroutine uses the value of the parameter within its execution. The action of the subroutine will be different depending upon the parameters that it is passed.

Parameters are placed in parenthesis after the subroutine name. For example:

Square(5) 'passes the parameter 5 – returns 25

Square(8) 'passes the parameter 8 – returns 64

Square(x) 'passes the value of the variable x

Use subroutines to modularize the solution to a problem

Subroutine/sub-program

A subroutine is a self-contained section of program code which performs a specific task and is referenced by a name.

A subroutine resembles a standard program in that it will contain its own local variables, data types, labels and constant declarations.

There are two types of subroutine. These are procedures and functions.

- Procedures are subroutines that input, output or manipulate data in some way.
- Functions are subroutines that return a value to the main program.

Topic: 4.1.1 Abstraction

A subroutine is executed whenever its name is encountered in the executable part of the main program. The execution of a subroutine by referencing its name in the main program is termed **'calling'** the subroutine.

The benefits of using procedures and functions are that:

- The same lines of code are re-used whenever they are needed – they do not have to be repeated in different sections of the program.
- A procedure or function can be tested/improved/rewritten independently of other procedures or functions.
- It is easy to share procedures and functions with other programs – they can be incorporated into library files which are then 'linked' to the main program.
- A programmer can create their own routines that can be called in the same way as any built-in command.

For more details about functions and procedures, re-visit Section 2.3.6

Abstract Data Types (ADTs)

In computer science, an abstract data type (ADT) is a mathematical model for certain data types or structures. This doesn't mean that ADTs can't be programmed, but that we must first understand their functionality before we can implement them. Different ADTs have different operations and store data in different ways.

The data that we keep on our computers and on the web need to be organized in a logical manner. Computer science has developed a number of 'data structures' to easily manipulate and sort the related data. Some of these structures are called the LIST, STACK, QUEUE, and TREE.

Dynamic and Static data structures

DYNAMIC	STATIC
Memory is allocated to the data structure dynamically i.e. as the program executes.	Memory is allocated at compile time. Fixed size.
Disadvantage: Because the memory allocation is dynamic, it is possible for the structure to 'overflow' should it exceed its allowed limit. It can also 'underflow' should it become empty.	Advantage: The memory allocation is fixed and so there will be no problem with adding and removing data items.
Advantage: Makes the most efficient use of memory as the data structure only uses as much memory as it needs	Disadvantage: Can be very inefficient as the memory for the data structure has been set aside regardless of whether it is needed or not whilst the program is executing.
Disadvantage: Harder to program as the software needs to keep track of its size and data item locations at all times	Advantage: Easier to program as there is no need to check on data structure size at any point.

Topic: 4.1.1 Abstraction

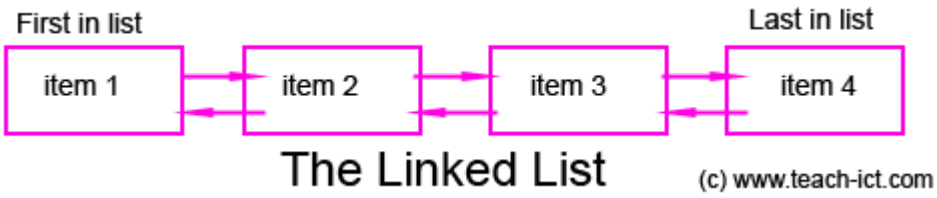
List

The basic **List** is a data structure having a number of items stored in the order that they were originally added. The 'List' can be allocated a fixed length - in which case it is a 'static data structure' on the other hand, if the list is allowed to grow or shrink then it is a 'dynamic data structure'.

An example of a simple list is the 'array' which can hold a number of data items or 'elements' as they are sometimes called. If the array is defined at compile time, then it is a 'static array'. If the array is allowed to vary in length then that is an example of a dynamic list.

A '**linked list**' is one where each data item points to its neighbors.

A linked list is excellent as a general storage structure because it is simple to insert and delete items and to find the first and last item.



Typical operations that can be carried out on a list are:

ADD (or INSERT)	Adds an item to the list
DELETE (or REMOVE)	Removes an item from the list
FIRST	Identifies the first item in the list
NEXT	Identifies the next item in the list
LAST	Identifies the end of the list
LOCATE	Identifies the location of a specific item within the list

Each of these operations has to be written as a 'function' or 'method' that acts upon the list.

Topic: 4.1.1 Abstraction

Stack

A LIST is a very general computer data structure. However, there are certain types of list that are so common, they are given their own name.

The 'STACK' is a **Last-In First-Out** (LIFO) List. Only the last item in the stack can be accessed directly.

If you had a number of items added to a stack in this order 1. 'Dog', 2. 'Cat', 3. 'Horse', the stack would look like this

If you had a number of items added to a stack in this order 1. 'Dog', 2. 'Cat', 3. 'Horse', the stack would look like this

Horse (last in)
Cat
Dog (first in)

The standard operations on a stack are

Operations on a stack	
PUSH	Adds an item to the top of the stack
POP	Removes an item from the top of the stack
TOP	Identifies the item at the top of the stack, but does not remove it

One of the most common uses for a stack in computer programming is to use it to control program flow within an application.

03-111-222-ZAK | OlevelComputer AlevelComputer | @zakonweb | zak@zakonweb.com | www.zakonweb.com

Topic: 4.1.1 Abstraction

Queue

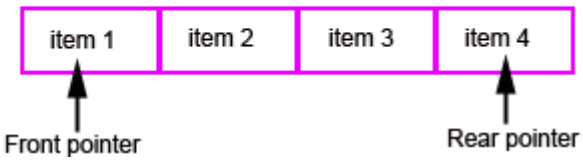
The QUEUE is another extremely common type of list data structure.

A queue is a **First-In, First-Out list.**

A queue maintains two pointers

1. A 'front of queue' pointer
2. An 'end of queue' pointer.

A Queue with its pointers



Uses of a queue

A queue data structure is used whenever there are a number of items waiting for a resource to become available. If you had three items added to a queue in this order 1. Dog 2.Cat 3. Horse the queue would look like this

Dog
Cat
Horse

The start pointer locates 'Dog' and the Rear pointer locates 'Horse'

The operations that are associated with a queue are

Operations on a QUEUE	
ADD	add an item to the back of the queue
REMOVE	removes the item at the front of the queue
FRONT	Identifies the item at the front of the queue, but does not remove it.

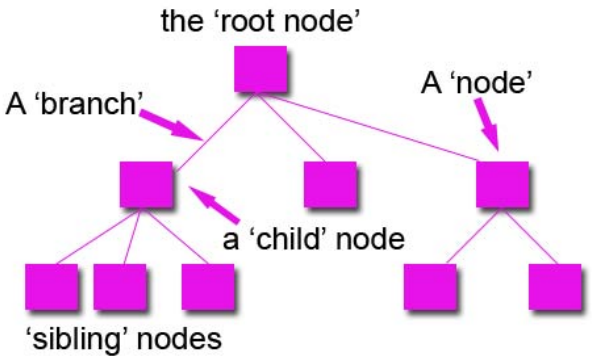
Topic: 4.1.1 Abstraction

Tree

The QUEUE and the STACK are linear lists. This means each data item only points to the one before it and after it. They have the idea of order built into them, such as 'last' or 'first'. But they do not imply there is any relationship between the data items themselves.

The TREE on the other hand, is designed to represent the relationship between data items.

Just like a family tree, a TREE data structure is illustrated below.



PARTS OF A TREE DATA STRUCTURE

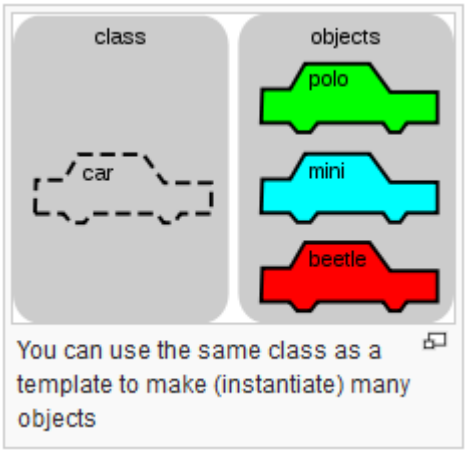
- Each data item within a tree is called a 'node'.
- The highest data item in the tree is called the 'root' or root node.
- Below the root lie a number of other 'nodes'. The root is the 'parent' of the nodes immediately linked to it and these are the 'children' of the parent node.
- If nodes share a common parent, then they are 'sibling' nodes, just like a family.
- The link joining one node to another is called the 'branch'.

Topic: 4.1.1 Abstraction

Classes

Object oriented programming is a type of programming paradigm based around programming classes and instances of classes called objects. These can be objects that appear on the screen (e.g., pictures, textboxes, etc.) or are part of the programming (e.g. actors, connections, particles, etc.).

Structures are very similar to Classes in that they collect data together. However, classes extend this idea and are made from two different things:



You can use the same class as a template to make (instantiate) many objects

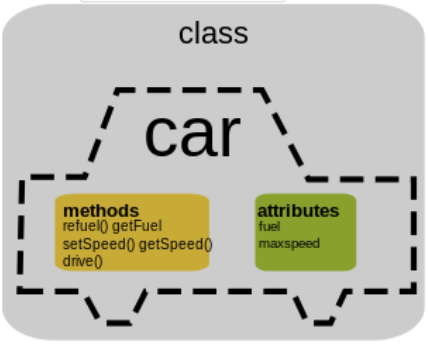
- **Attributes** - things that the object stores data in, generally variables.
- **Methods** - Functions and Procedures attached to an Object and allowing the object to perform actions

Let's take a look at the following example:

```
class car
  private maxSpeed as integer
  public fuel as integer
  public sub setSpeed(byVal s as integer)
    maxSpeed = s
  end sub
  public function getSpeed() as integer
    return maxSpeed
  end function
  public sub refuel(byVal x as integer)
    console.writeline("pumping gas!")
    fuel = fuel + x
  end sub
  public function getFuel() as integer
    return fuel
  end function
  public sub drive()
    fuel = fuel - 1
  end sub
end class
```

You can see that the class is called `car` and it has:

- two attributes: `maxSpeed, fuel`
- five methods
  - three procedures: `setSpeed, refuel, drive`
  - two functions: `getSpeed, getFuel`



Remember this is a class and therefore only a template, we need to 'create' it using an object

03-111-222-ZAK
OlevelComputer
AlevelComputer
@zakonweb
zak@zakonweb.com
www.zakonweb.com

Topic: 4.1.1 Abstraction

Attributes

These store information about the object. In the example above we store the fuel and maxSpeed. The attributes are attached to the classes, and if there are several instances (objects) of the classes then each will store its own version of these variables. Note that instead of the usual dim, there is the word private or public, we'll cover that in the encapsulation section.

Methods

Unlike structures, OOP allows you to attach functions and procedures to your code. This means that not only can you store details about your car (the attributes), you can also allow for sub routines such as drive() and refuel, which are attached to each class.

Facts and Rules

A declarative language is non-procedural and very high level. This means that the programmer specifies what needs to be done rather than how to do it.

The software will seek an answer to the question (goal) by interrogative a database containing **Facts** and **Rules**. It doesn't matter what order the facts and rules are arranged within the database – unlike procedural languages – the computer will find the best path towards the answer. There will either be a matching answer – for example, the question (goal) might be **'Who is David's wife?'** or a 'False' is returned where there was no answer to be found.

This type of language is geared more towards applications such as artificial intelligence and expert systems where inexact data has to be handled or general decisions have to be made.

PROLOG is a declarative language that was developed for artificial intelligence.

Example

Consider a PROLOG database containing the following facts and rules:

```
1. Fact: spouse (john, jane)
2. Fact: spouse (david, mary)
3. Fact: spouse(george, susan)
4. Fact: female (jane)
5. Fact: female (mary)
6. Fact: female (susan)
7. Fact: male (john)
8. Fact: male (david)
9. Fact: male (george)
10.   Rule: husband(A,B) IF spouse(A,B) AND male (A)
11.   Rule: wife (A,B) IF spouse(A,B) AND female(B)
```

Topic: 4.1.1 Abstraction

A query could be written:

```
wife(david,mary)?
```

The formal name for this statement is **'goal'**. The work of finding the answer is called **'satisfying the goal'**

Effectively, this is asking if Mary is the wife of David. In order to provide an answer, the following takes place:

1. Prolog will first of all scan the Rules looking for a match.
2. Rule 11 fits, where A and B can be any two names.

```
wife (A,B)
```

3. Then it applies the rule by looking to see if it can find a match to the spouse(A,B) condition and yes - Fact 2 fits.

```
spouse(david,mary)
```

4. Next it looks to see if the second condition female(B) has a match and yes Fact 5 is a match.

```
Fact: female (mary)
```

5. Both conditions have been met, so yes Mary is David's wife.

   An important feature of declarative languages is 'backtracking' where a search goes partially back on itself if it fails to find a complete match the first time around. This will be discussed in more detail later on.