Acknowledgements

Thanks are due to the members of Heriot-Watt University's SCHOLAR team who planned and created these materials, and to the many colleagues who reviewed the content.

We would like to acknowledge the assistance of the education authorities, colleges, teachers and students who contributed to the SCHOLAR programme and who evaluated these materials.

Grateful acknowledgement is made for permission to use the following material in the SCHOLAR programme:

The Scottish Qualifications Authority for permission to use Past Papers assessments.
The Scottish Government for financial support.
The content of this Study Guide is aligned to the Scottish Qualifications Authority (SQA) curriculum.

All brand names, product names, logos and related devices are used for identification purposes only and are trademarks, registered trademarks or service marks of their respective holders.
# Contents

1 Programming paradigms 1
   1.1 Revision .................................. 3
   1.2 Introduction ................................ 4
   1.3 Imperative Programming ...................... 5
   1.4 Object Oriented Programming .................. 7
   1.5 Concurrent Programming ...................... 17
   1.6 Conclusion ................................ 20
   1.7 End of topic test ........................... 21

2 Data types and structures 25
   2.1 Revision .................................. 26
   2.2 2-D Arrays .................................. 27
   2.3 Records and arrays of records .................. 32
   2.4 Linked Lists ................................ 33
   2.5 Stacks ..................................... 37
   2.6 Queues ..................................... 41
   2.7 Array of Objects ............................ 45
   2.8 Conclusion ................................ 47
   2.9 End of topic test ........................... 48

3 Development methodologies 51
   3.1 Revision .................................. 52
   3.2 Traditional versus contemporary methodologies .................................. 53
   3.3 Problem decomposition ........................ 53
   3.4 Iterative prototyping ........................ 55
   3.5 Contemporary development methodologies .................................. 55
   3.6 Conclusion ................................ 57
   3.7 End of topic test ........................... 58

4 Design notations 59
   4.1 Revision .................................. 60
   4.2 Pseudocode ................................ 61
   4.3 Unified Modelling Language (UML) ................. 65
   4.4 Wireframing ................................ 72
   4.5 Conclusion ................................ 74
   4.6 End of topic test ........................... 75

5 Standard algorithms 77
   5.1 Revision .................................. 79
   5.2 Introduction ................................ 81
5.3 Binary search ........................................ 81
5.4 Sorting algorithms ..................................... 88
5.5 Other sorting algorithms ............................... 101
5.6 Conclusion ........................................... 103
5.7 End of topic test ...................................... 104

6 Computational constructs 107
6.1 Recursion ............................................. 108
6.2 File Handling ......................................... 111
6.3 Conclusion ........................................... 114
6.4 End of topic test ...................................... 115

7 Testing and documenting solutions 117
7.1 Revision .............................................. 118
7.2 Introduction .......................................... 119
7.3 Component testing .................................... 119
7.4 Integration testing .................................... 120
7.5 Acceptance (beta) testing ............................ 121
7.6 Usability testing ...................................... 121
7.7 Accessibility testing .................................. 122
7.8 Summary ............................................. 123
7.9 End of topic test ...................................... 124

8 End of Unit 1 test ...................................... 127

Glossary 135

Answers to questions and activities 139
1 Programming paradigms ................................ 139
2 Data types and structures .............................. 143
3 Development methodologies ......................... 146
4 Design notations ....................................... 147
5 Standard algorithms ................................... 151
6 Computational constructs ............................ 157
7 Testing and documenting solutions .................. 161
8 End of Unit 1 test ...................................... 163
Topic 1

Programming paradigms

Contents

1.1 Revision ......................................................... 3
1.2 Introduction ................................................... 4
1.3 Imperative Programming ................................. 5
1.4 Object Oriented Programming ....................... 7
   1.4.1 Classes and Objects in Object Oriented Programming ............ 7
   1.4.2 Class diagrams ...................................... 8
   1.4.3 Classes and objects example: The music player .............. 11
   1.4.4 Music Player: the PlayList class ........................... 12
   1.4.5 Inheritance ........................................... 15
1.5 Concurrent Programming ............................... 17
1.6 Conclusion ................................................... 20
1.7 End of topic test ............................................ 21

Prerequisite knowledge

From your studies at Higher you should already know:

- the basic differences between Imperative, Declarative, Object-oriented and Domain-specific languages;
- that modern programming environments provide a variety of tools to write and debug programs;
- that all programming languages can be reduced to three basic control structures: sequence, selection and iteration;
- the difference between global and local variables;
- the difference between actual and formal parameters.

Learning objectives

By the end of this topic you will be able to:

- Understand the main features of:
  - Imperative Programming;
TOPIC 1. PROGRAMMING PARADIGMS

- Object Oriented Programming;
- Concurrent Programming.
1.1 Revision

Quiz: Revision

Q1: Which one of the following types of language would be most suitable for programming a knowledge base in an expert system?

a) Procedural  
b) Declarative  
c) Functional  
d) Object-oriented

Q2: Which type of programming language uses the concept of a class which describes the data and operations associated with that data as a single entity?

a) Procedural  
b) Declarative  
c) Domain-specific  
d) Object-oriented

Q3: Which type of programming language uses a knowledge base of facts and rules which is matched with a query to provide a solution to a problem?

a) Procedural  
b) Declarative  
c) Domain-specific  
d) Object-oriented

Q4: Which of these features would you expect to be provided by a high level language programming environment?

a) Keyword highlighting  
b) Automatic indentation  
c) Search and replace  
d) Debugging tools  
e) Spell check

Q5: Which of the statements below describes a class library?

a) A list of all of the classes used in a program.  
b) A file containing all the actual code used in a program.  
c) A set of pre-written classes that tests part of a program.  
d) A set of pre-tested classes which can be used in a program.
Q6:
This line of code is in a program to add the name "Fred" to an array of STRINGS:

```
addNameToList("Fred", names)
"Fred"
and
names
are:
```

a) Formal parameters  
b) Actual parameters  
c) Real parameters  
d) Reference parameters

Q7:

```
PROCEDURE addNameToList (STRING name, ARRAY OF STRING listOfNames)
In this procedure definition, name,
and
listOfNames
are:
```

a) Formal parameters  
b) Actual parameters  
c) Real parameters  
d) Reference parameters

1.2 Introduction

A **programming paradigm** is a specific style of computer programming. We are going
to look at three recognized programming paradigms: Imperative Programming, which is
the oldest paradigm, and reflects the sequential nature of the operation of the computer;
Object oriented programming which was developed to improve programmer efficiency
and to reduce errors as software became more complex; and Concurrent Programming
which was developed from programs written to make efficient use of multi processor
systems (where several processes are running and interacting simultaneously).
1.3 Imperative Programming

Learning objective

By the end of this section you will be able to:

- Understand the main features of Imperative Programming:
  - the three basic control structures: sequence, selection and iteration;
  - the use of variables;
  - the benefits of modularity.

In everyday English, an imperative is a command. In Imperative Programming a program consists of a set of commands. When and if these commands are executed will depend on how the three basic control structures have been used:

- **sequence** (one command is executed followed by another);
- **selection** (if a condition is true then one command is executed, else if that condition is false then a different command is executed);
- **iteration** (a command is executed a set number of times).

Although there are only three basic control structures, program code can be made more understandable if these control structures are combined to make more powerful computational constructs such as fixed loops, conditional loops, and simple and complex conditional statements. These constructs can be used as the components of algorithms, and as the building blocks of sub-programs (such as functions and procedures).

As the name suggests, a **variable** is something which can be changed. Variables in Imperative Programming are used to store values which can have their values changed within the program and which can be passed as parameters to procedures and functions. A variable can exist temporarily within a function or procedure (a **local** variable) or can exist while a program is running and have its value changed anywhere within that program (a **global** variable).

Imperative Programming is sometimes referred to as **procedural programming**. Procedures and functions are sets of commands which can be called by name within a program. These sets of commands can be used within a program as if they were a single entity, and can also have values (called parameters) passed to them which change their behaviour. Being able to combine commands into a single block of self-contained code is referred to as **modularity**. Modular programming has a number of advantages:

- Modules can be treated as separate sub-programs which can be tested independently of the main program;
- Modules can be re-used within a program. It is also possible to create a module library: a module or modules which are saved so that they can be used in other programmes. Ideally, these modules will be well documented (there will documentation of some sort which details what the module/modules do, the parameters that are required and the output(s) they generate). Also, module libraries won’t make use of global variables (this makes the code reusable because
it doesn't depend on a variable that may not be present). These module libraries can be used in projects to reduce development time (because the developer can reuse the code);

- Large programming projects will benefit from a modular approach because several people can work on them simultaneously, each one working on a different module;
- Modular programming makes code more readable because the main part of a program can be presented as a concise set of commands; each one calling a procedure or function.

The code below is a simple example of an input validation function which uses two formal parameters, \texttt{lowerLimit} and \texttt{upperLimit} in its definition. Because its code does not refer to any global variables, it can be treated as a separate self-contained block of code which can be tested independently. This function could be called within a program with a number of different actual parameters. Because it is self-contained, this function could be part of a module library and re-used within a different program without having to be re-written.

```fortran
FUNCTION getValidInteger(INTEGER lowerLimit, INTEGER upperLimit)RETURNS INTEGER

RECEIVE userInput FROM (INTEGER) KEYBOARD

WHILE userInput < lowerLimit OR userInput > upperLimit DO

SEND "Input must be between "$ lowerLimit " and "$ upperLimit] TO DISPLAY

RECEIVE userInput FROM (INTEGER) KEYBOARD

END WHILE

RETURN userInput

This function might be called within the main section of a program with a command such as:

```fortran
myValue = getValidInteger(1,100)
```

which would call the \texttt{getValidInteger} function with the actual parameters 1 and 100 and would result in assigning a value from 1 to 100 to the variable \texttt{myValue}.

The while loop in this function is constructed from all three basic control structures: The commands are executed in sequence, a selection is made depending on whether the input received is between the upper and lower limits, and the loop repeats while the input is outside those limits.
1.4 Object Oriented Programming

Learning objective
By the end of this section you will be able to:

- Understand the main features of Object Oriented Programming:
  - Classes
  - Objects
  - Methods
  - Encapsulation
  - Instance Variables
  - Inheritance.

1.4.1 Classes and Objects in Object Oriented Programming

As programs become more complex, it is important to limit how much one part of a program can manipulate the data in another section of the program. Object orientation allows a program to be separated into blocks of related data and the operations which apply to that data. Linking the data and its operations together in this way allows it to be treated as a single entity (an object) and means that the data can only be accessed directly by its associated operations. Restricting how an object's data can be accessed in this way is referred to as encapsulation.

A class is a blueprint for an object. Once a class has been defined, any number of objects can be created from that class. When you are defining a class, you need to think about what kind of objects will be created from that class. A class definition will describe what data it needs to use, known as its Instance Variables, (Sometimes known as Properties) and what it will be able to do, known as its Methods. The use of classes in Object Oriented Programming means that class libraries can be built up, saving project development time.

For example, a button has characteristics such as its colour, the text on its label, and its shape, so a Button class would need instance variables to include colour, label, and shape. Its methods might be ChangeColour, ChangeShape, OnHover, OnPress, OnRelease etc.

<table>
<thead>
<tr>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>color</td>
</tr>
<tr>
<td>label</td>
</tr>
<tr>
<td>shape</td>
</tr>
<tr>
<td>changeColour()</td>
</tr>
<tr>
<td>changeShape()</td>
</tr>
<tr>
<td>onHover()</td>
</tr>
<tr>
<td>onPress()</td>
</tr>
<tr>
<td>onRelease()</td>
</tr>
</tbody>
</table>
A music player application might contain a playList class. This class would need to store a list of music tracks, the maximum number of tracks it can contain, and its total playing time. This playList class would have instance variables: an array of tracks, playingTime and maxSize. It would have methods: AddTrack, DeleteTrack, FindTitle.

<table>
<thead>
<tr>
<th>PlayList</th>
</tr>
</thead>
<tbody>
<tr>
<td>tracks[], playingTime, maxSize</td>
</tr>
<tr>
<td>addTrack()</td>
</tr>
<tr>
<td>deleteTrack()</td>
</tr>
<tr>
<td>findTitle()</td>
</tr>
</tbody>
</table>

Object-oriented languages depend greatly on the concept of class libraries. These are sets of classes that can be used by developers as common building blocks for complex applications, similar to the module libraries that are used in procedural programming. A procedural language program is built from a number of procedures which are called from a main procedure. An object oriented program has a number of methods linked to the class within which they are defined, in keeping with the idea that classes and objects are self-contained.

1.4.2 Class diagrams

Classes can be described using a class diagram. When you create a class diagram you need to think about the instance variables and the methods which objects created from the class will need.

For example if we were writing a program where the user has to guess the value of a random number generated by the computer, we might construct a Player class. The Player class will need instance variables for the player's name and their guess and methods to make a guess and to check that the guess is valid.

<table>
<thead>
<tr>
<th>Player</th>
</tr>
</thead>
<tbody>
<tr>
<td>name, guess</td>
</tr>
<tr>
<td>play()</td>
</tr>
<tr>
<td>valid_Guess()</td>
</tr>
</tbody>
</table>

```
CLASS Player IS {STRING name, INTEGER guess}
METHODS
PROCEDURE play()
DECLARE THIS.guess INITIALLY 0
SEND "I’m thinking of a number, please enter your guess" TO DISPLAY
```
SET THIS.guess TO validGuess()
END PROCEDURE

FUNCTION valid_Guess() RETURNS INTEGER

DECLARE userInput INITIALLY 0
RECEIVE userInput FROM KEYBOARD

WHILE userInput < 0 OR userInput > 100 DO
    SEND "choose as value from 1 to 100" TO DISPLAY

    RECEIVE userInput FROM KEYBOARD

END WHILE

RETURN THIS.userInput

END FUNCTION

END CLASS

Note that in the Class definition here we are using the predefined name THIS to refer to the instance variables of the class.

To set up the game we could create a new player object from this class. Creating an object from a class is known as instantiation.

DECLARE player1 INITIALLY Player ("Fred", 0)

And they could be given an opportunity to make a guess using the command:

player1.play

This may not seem to have any benefits over the procedural programming paradigm, but if we wanted to build a multiplayer game, then we could easily use the Player class to create new players, each one with their own local name and guess instance variables.

DECLARE player2 INITIALLY Player ("Barney", 0)
DECLARE player3 INITIALLY Player ("Wilma", 0)

Further class diagram examples:

If we were creating a combat game where players and computer characters fight each other, then we might define a player class with instance variables to store their name and health and a method to return their health value.
New player objects could be created with these commands:

DECLARE player1 INITIAL Player ("Destroyer", 50)
DECLARE player2 INITIAL Player ("Avenger", 50)

Having defined a Player class, we could define a Team class which has as its instance variables, an array of players and the number of points held, with methods for players to join and leave the team, and one to return the points value.

CLASS Team IS {ARRAY OF Player players, INTEGER points}
METHODS
PROCEDURE join( Player newplayer)
  <Add player to players array>
END PROCEDURE

PROCEDURE leave(INTEGER index)
  <remove player from Players array identified by their index position in the array>
END PROCEDURE
New teams could be created with these commands:

DECLARE team1 INITIALLY ( [player1, player2], 0)
DECLARE team2 INITIALLY ( [player3, player4], 0)

1.4.3 Classes and objects example: The music player

In this example we will define two classes: a Track class and a PlayList class.

The Track class has three instance variables: title, artist and length, and one method: showTrack.

```
FUNCTION getPoints() RETURNS INTEGER
RETURNS THIS.points
END FUNCTION

END CLASS
```

The predefined name THIS is used to access the instance variables stored in the current method.

We can create a track object from the Track class like this:

DECLARE favouriteTrack INITIALLY Track ("Telstar", "Tornados", 3.4)

And the ShowTracks method can be used like this:

favouriteTrack.showTrack()
The result of this command would be to send:

**Telstar Tornados 3.4**

to the screen.

### 1.4.4 Music Player: the PlayList class

The PlayList class has four instance variables: `tracks` which is an array of `Track`, `next` which is used to store the next available place in the list, `max` which is used to store the maximum number of tracks, and `playTime` to store the total playing time.

```
CLASS PlayList IS {ARRAY OF Track tracks, INTEGER next, INTEGER maxLength, REAL playTime}

# Creating instance of a class is like creating an instance of a record.

METHODS

CONSTRUCTOR PlayList(INTEGER maxLength)

# the CONSTRUCTOR is used to give initial values to the instance variables

DECLARE THIS.maxLength INITIALLY maxLength
DECLARE THIS.tracks INITIALLY THIS.Track[maxLength]
DECLARE THIS.next INITIALLY 0
DECLARE THIS.playTime INITIALLY 0.0

# note the use of THIS to refer to the instance variables

END CONSTRUCTOR

PROCEDURE addTrack(Track newtrack)
```
# adds a new track to the playList
# if next is less than the maximum number allowed

IF THIS.next = THIS.maxLength THEN
  <error - too many tracks>
ELSE
  SET THIS.tracks[THIS.next] TO newTrack
  SET THIS.next TO THIS.next + 1
  SET THIS.playTime TO THIS.playTime + THIS.newTrack.length
END IF

END PROCEDURE

FUNCTION findTitle(STRING trackTitle) RETURNS INTEGER

# finds a track in the playList

DECLARE result INITIALLY -1

# will return -1 if title not found

FOR index FROM 0 TO maxLength - 1 DO
  IF THIS.tracks[index].title = trackTitle THEN
    SET result TO index
    END IF
END FOR
RETURN result

END FUNCTION

PROCEDURE deleteTrack(INTEGER index)

# deletes a track from the playList

IF index >= THIS.maxLength OR index < 0 THEN
  <error - invalid index>
ELSE
  SET THIS.playTime TO THIS.playtime - THIS.tracks[index].length
  FOR counter FROM index TO next-1 DO
    SET THIS.tracks[counter] TO THIS.tracks[counter + 1]
  END FOR SET
  THIS.next TO THIS.next-1
END IF
END PROCEDURE

PROCEDURE showTracks()

    # loops through the playList displaying each track
    FOR counter FROM 0 TO next - 1 DO
        THIS.tracks[counter].showTrack()
    END FOR

END PROCEDURE

END CLASS

The PlayList class is more complex than previous examples. The instance variables include an array of Track together with variables to store total playing time, maximum number of tracks, and the next available index position in the array if the playlist is not full.

The CONSTRUCTOR method for the PlayList class takes maxlen as a parameter and gives initial values to this and the other instance variables which are used whenever a new object is created. This saves having to give these variables initial values every time a new object is created.

The addTrack procedure uses maxLength and next to check to see if there is space in the playlist and adds newTrack to the array if space is available, incrementing next if successful.

The findTitle function takes trackTitle as a parameter and uses a linear search to return the index position of the track when found.

The deleteTrack procedure takes the index position of the track to be deleted as a parameter and if it is a valid index position, removes that track from the playlist and updates next so that an additional space is available.

The showTracks procedure uses a loop to send all the track details in the playlist to the screen.

Once we have these two classes, Track and PlayList defined, we can use them to create new objects and use their associated methods.

We can create a playList object named MyTracks from the PlayList class like this:

DECLARE myTracks INITIALLY PlayList(10) which sets up the myTracks playList with 10 tracks

We can now use the addTrack method from the PlayList class to add tracks to the playList:

    myTracks.addTrack(track("Hey Joe","Jimi Hendrix",5.47))
    myTracks.addTrack(track ("Smithsonian Institute Blues","Captain Beefheart",3.59))

and using the favouriteTrack object we created from the Track class we can add it to the playList as well.

myTracks.addTrack(favouriteTrack)
Activity: PlayList

Assuming that the commands above have been successfully executed, Write the pseudocode to do the following:

1. Set up a new playList called recentTracks.
2. Add "Rockaway Beach" by the Ramones 3.4 mins duration to the recentTracks playList.
3. Show all the tracks in the recentTracks playList.
4. Delete Hey Joe by Jimi Hendrix from the myTracks playList.

You can use the Haggis reference language or any other pseudocode you are comfortable with to complete this task.

Activity: Class

A bank statement class contains:

- an array of entries;
- a variable to store the maximum entries it can contain;
- a variable to store the next available place in the list;
- and a balance.

An entry is a record containing:

- a date;
- and an amount (deposit or withdrawal).

Create class diagrams for BankStatement and Entry with appropriate methods.

a) Write the pseudocode to define these two classes.
b) Write a method within the BankStatement class to add a new Entry.
c) Write a method within the BankStatement class to display the last 30 entries.
d) Write a method within the BankStatement class to display all the entries in the statement.

You can use the Haggis reference language or any other pseudocode you are comfortable with to complete this task.

1.4.5 Inheritance

Classes can be hierarchical. This means that a class can be created which contains instance variables and methods common to a number of different types of object, and
sub-classes which inherit the methods of the class above them (the class above is referred to as a **superclass**). A sub-class of the Button class might be ImageButton which changes its image when the mouse is moved over it. This ability to create a sub-class from a pre-defined class is called **inheritance**. The benefit of using this technique is that once a class has been defined and tested, sub-classes can then be created which share the main characteristics of the parent class, thus saving testing and development time.

A method in a sub-class can be written to override the method in the super class above it. This means that when the method is called the one in the sub-class is executed instead of the one in the super class. This ability to redefine methods in object oriented programming is called **polymorphism**.

If you need one of the sub classes to have a different method then that method can override these methods for its specific circumstances.

**Music player inheritance example: The AlbumTrack class**

We can define a new class which inherits all the instance variables and methods from the Track superclass and is extended with additional instance variables and methods.

```
CLASS AlbumTrack INHERITS Track WITH { STRING AlbumName}

METHODS

FUNCTION getAlbum() RETURNS STRING
    RETURN THIS.AlbumName
```
A value of a subclass can be created using the subclass name and all the data elements of the superclass followed by the elements of the subclass. In this example we can assign a track to a playList.

```
SET favouriteTrack2 TO AlbumTrack ("Telstar", "Tornados", 3.4, "SixtiesHits")
SEND "This track is in the " & favouriteTrack2.getAlbum" Album"
```

The three instance variables `Telstar`, `Tornados`, 3.4 are defined in the superclass `Track` and inherited by the sub-class `AlbumTrack`.

The instance variable `SixtiesHits` is defined in the `AlbumTrack` SubClass. Notice that the superclass instance variables appear first, then the sub-class instance variables when the new Object is created.

### 1.5 Concurrent Programming

**Learning objective**

By the end of this section you will be able to:

- understand the main features of Concurrent Programming.

In a concurrent program, several streams of operations may execute simultaneously.
This type of programming is particularly relevant to programs running on systems with more than one processor. In Concurrent Programming, each stream of operations executes as it would if it occurred in a sequential program except for the fact that streams can communicate and interfere with one another. Each stream or sequence of instructions is called a thread.

These threads are typically managed by a scheduler (part of the operating system) which allows access to shared resources and data (e.g., processor(s), variables etc.). Threads can be given levels of priority by the scheduler which means that more important threads get a greater amount of processor time - this can mean that low-priority threads only get the processor when all high-priority threads are waiting (or otherwise unable to run).

Concurrent programs are difficult to write and debug because it is difficult to predict when one thread will need information from another before it can continue executing. This situation is known as a Deadlock. Deadlock is when two threads are each waiting for the other to finish and as a result neither does. For example, in an operating system Thread "A" may enter a waiting state because the resource it needs is being used by Thread "B" but Thread "B" enter's a waiting state because the resource it needs to access is in use by Thread "A". Neither thread can make progress until the other releases its resource so deadlock occurs and neither thread ever completes its operation.

Similarly, if two threads both try to change the value of the same variable, the resulting value can be unpredictable unless some system of locking is in place. This situation, where two threads are potentially racing against each other to update or access a variable, is known as a Race Condition. Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". For example:

```
IF x = 5 THEN
  # This is the "check"
  SET y TU x * 2  # This is the "Act" #
  # If another thread changed x in between "if (x = 5)"
  and " SET y TU x * 2" above, #
  # y will not be equal to 10. #
END IF
```

In this case, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. There is no real way of knowing.

In order to prevent race conditions from occurring, a lock is placed on the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
# Obtain lock for x #
IF x = 5 THEN
  SET y TU x * 2
  # Now, nothing can change x until the lock is released, therefore y = 10 #
END IF
# Release lock for x
```
A third potential problem is **Resource Starvation** where the priority given to one or more threads means that they continuously access resources which are needed for another thread. Because this thread can never access the resources it requires, it will never manage to complete. This can be caused by poor scheduling, or the fact that higher priority threads always take precedence over lower priority ones.

In general, Concurrent Programming requires coordination, known as **concurrency control**, so that interactions between threads are correctly sequenced, and access to resources is coordinated so that they are shared among operations according to their priority.

The advantages of Concurrent Programming are:

- A user can interact with applications while tasks are running, such as stopping file download in a web browser.
- Long-running tasks need not delay short-running ones, for instance a web server can serve a web page while at the same time processing a query.
- Complex programs can make better use of multiple resources in new multi-core processor architectures.

Concurrent Programming approaches are a feature of declarative and Object Oriented Programming languages.

Java, Erlang and Salsa are examples of languages specifically designed with Concurrent Programming in mind.

Due to the complexity of writing concurrent programs, the role of setting up groups of program instructions as threads to run concurrently is typically managed by translation software. The programmer will focus on developing code to solve a particular problem but the translation software will deal with the threading of the translated code.
1.6 Conclusion

Summary

- The three basic control structures, sequence, selection and iteration underlie all programming languages.
- Variables are used to store values which can be changed within a program and which can be passed as actual parameters to procedures and functions.
- Formal parameters are those declared in the definition of a procedure or function.
- A module is a sub-program which uses local variables improves the readability and maintainability of code.
- Object Oriented Programming uses classes which define both the instance variables and the methods of the objects which are created from them.
- Object Oriented Programming uses encapsulation which makes code less prone to side effects since the instance variables and methods associated with an object are only accessible within that object.
- Inheritance in Object Oriented Programming is where a sub-class is created which inherits the methods and instance variables of the class above it, but which has additional methods and instance variables of its own.
- Concurrent Programming is where several streams of operations may execute concurrently, but simultaneous access to data needs to be strictly controlled making programming difficult.
1.7 End of topic test

End of topic test

Q8: Which of these statements are true?
   a) Variables are only used in Imperative Programming.
   b) Keeping variables local improves modularity of code.
   c) A variable is something whose value can be changed.
   d) Only variables can be passed as parameters to a sub-program.

Q9: Which of these statements are true of the three basic control structures, sequence, selection and iteration?
   a) They underlie all programming languages.
   b) They are not used in Object Oriented Programming.
   c) They can be combined to create more powerful computational constructs.
   d) They improve the modularity of code.

Q10: Which of these are benefits of modularity?
   a) Modularity improves the readability of code.
   b) Global variables improve modularity.
   c) All sub-programs must have parameters.
   d) Formal parameters are those declared in the definition of a sub-program.

Q11: In Object Oriented Programming ________ enables you to hide, inside an object, both the instance variables and the methods that act on them.
   a) Encapsulation
   b) Inheritance
   c) Recursion
   d) Iteration

Q12: In Object Oriented Programming ________ is an abstract idea that can be represented with instance variables and methods.
   a) A Class
   b) An Object
   c) A Function
   d) A Variable
Q13: In Object Oriented Programming ________ is the process of creating new classes, called sub-classes, from an existing class.

a) Encapsulation  
b) Inheritance  
c) Recursion  
d) Iteration

Q14: What instance variables are inherited by the three icon classes?

<table>
<thead>
<tr>
<th>Icon</th>
<th>FileIcon</th>
<th>EntityIcon</th>
<th>ShortcutIcon</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER x_pos</td>
<td>STRING imagePath</td>
<td>STRING imagePath</td>
<td>STRING imagePath</td>
</tr>
<tr>
<td>INTEGER y_pos</td>
<td>STRING filePath</td>
<td>STRING entityPath</td>
<td>STRING appPath</td>
</tr>
<tr>
<td>STRING name</td>
<td></td>
<td></td>
<td>MoveTo (xpos,ypos)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Delete()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Q15: To make the structure more efficient, a DesktopIcon subclass is created with three new subclasses: FileIcon, EntityIcon and ShortcutIcon:

<table>
<thead>
<tr>
<th>DesktopIcon</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING imagePath</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MoveTo (xpos,ypos)</td>
</tr>
<tr>
<td></td>
<td>Delete()</td>
</tr>
</tbody>
</table>

What Instance variables would the three new subclasses of the DesktopIcon have?

© Heriot-Watt University
Q16: In Concurrent Programming, a ____________ is where two threads access the same data simultaneously resulting in unpredictable results.

a) Deadlock  
b) Race Condition  
c) Priority Condition  
d) Resource Starvation

Q17: In Concurrent Programming, a ____________ where two threads are competing for the same resource resulting in neither completing its operation.

a) Deadlock  
b) Race Condition  
c) Priority Condition  
d) Resource Starvation
Topic 2

Data types and structures

Contents

2.1 Revision ........................................... 26
2.2 2-D Arrays ........................................ 27
2.3 Records and arrays of records .................. 32
2.4 Linked Lists ....................................... 33
   2.4.1 Linked Lists - extension material ........ 36
2.5 Stacks .............................................. 37
2.6 Queues ............................................. 41
2.7 Array of Objects ................................. 45
2.8 Conclusion ......................................... 47
2.9 End of topic test .................................. 48

Prerequisite knowledge

From your studies at Higher you should already know:

- the difference between a simple and a structured data type;
- that data types are available in your chosen programming language;
- how your programming language handles records.

Learning objectives

By the end of this topic you should be able to implement the following data structures in your chosen programming language:

- 2-D arrays;
- Records and arrays of records;
- linked lists;
- stacks;
- queues;
- arrays of objects.
2.1 Revision

Quiz: Revision

What data structure would you use to store:

Q1: The last 20 key-presses made while editing a document?
   a) INTEGER
   b) REAL
   c) CHARACTER
   d) BOOLEAN
   e) STRING
   f) ARRAY of INTEGER
   g) ARRAY of REAL
   h) ARRAY of CHARACTER
   i) ARRAY of BOOLEAN
   j) ARRAY of STRING
   k) ARRAY of RECORD

Q2: A list of Email addresses?
   a) INTEGER
   b) REAL
   c) CHARACTER
   d) BOOLEAN
   e) STRING
   f) ARRAY of INTEGER
   g) ARRAY of REAL
   h) ARRAY of CHARACTER
   i) ARRAY of BOOLEAN
   j) ARRAY of STRING
   k) ARRAY of RECORD

Q3: A list detailing whether or not the emails in a folder have been replied to?
   a) INTEGER
   b) REAL
   c) CHARACTER
   d) BOOLEAN
   e) STRING
   f) ARRAY of INTEGER
   g) ARRAY of REAL
   h) ARRAY of CHARACTER
   i) ARRAY of BOOLEAN
   j) ARRAY of STRING
   k) ARRAY of RECORD
Q4: A set of room descriptions and contents in an adventure game?

a) INTEGER
b) REAL
c) CHARACTER
d) BOOLEAN
e) STRING
f) ARRAY of INTEGER
g) ARRAY of REAL
h) ARRAY of CHARACTER
i) ARRAY of BOOLEAN
j) ARRAY of STRING
k) ARRAY of RECORD

Q5: A set of pupil names, dates of birth and test scores?

a) INTEGER
b) REAL
c) CHARACTER
d) BOOLEAN
e) STRING
f) ARRAY of INTEGER
g) ARRAY of REAL
h) ARRAY of CHARACTER
i) ARRAY of BOOLEAN
j) ARRAY of STRING
k) ARRAY of RECORD

2.2 2-D Arrays

Learning objective

By the end of this section you will be able to:

• implement a 2-D array in your chosen programming language.

A 1-D array is a data structure which contains a set of variables of the same data type whose position is stored using an integer index. You can think of 1-D as meaning 1 dimensional.

The statement:

DECLARE myArray INITIALLY [34, 6, 77, 0, 45, 23, 6, 27, 92, 5]

would set up an array of 10 integers indexed from 0 to 9.
In this example, the command:

```
SEND myArray [4] TO DISPLAY
```

would print the value 45 which is the element with an index of 4.

Much of the information in the real world which you may wish to store is best presented as a two dimensional grid rather than a one dimensional list. A spreadsheet for example is arranged using row and column values to store numerical information. If you want to store games like Sudoku, Go and Chess, a 2-D grid is the obvious solution.

A **2-D array** is just an array of arrays

```
DECLARE myArray INITIALY
[[23,6,78,0],
 [16,4,14,27],
 [18,8,102,34],
 [49,4,8,45],
 [5,7,93,2]]
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>23</td>
<td>6</td>
<td>78</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>4</td>
<td>14</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>8</td>
<td>102</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>4</td>
<td>8</td>
<td>45</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>93</td>
<td>2</td>
</tr>
</tbody>
</table>

In this example the command:

```
SEND myArray [1][3] TO DISPLAY
```

Would print the value 27 because this is the element at position 1, 3. When processing 2D arrays the first index is the number of steps down from the top left corner of the array and the second index represents the number of steps to the right - you go down and then along!
We are going to look at how we could represent a very simple 3 x 3 maze using a 2-D array.

The rooms in the maze are numbered 0 to 8

You can move in any of four directions, North, East, South and West. We will use the numbers 0 to 3 to identify these four directions.

The maze can be represented as a grid. The numbers in the grid represent the room you would enter if you moved in that direction.

A move in a direction which hits a wall can be represented by an integer out-with the
range 0 to 8. In this case the value -1 is going to be used.

So now we have a 9 X 4 grid which looks like this:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>7</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>7</td>
<td>-1</td>
<td>8</td>
<td>-1</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>7</td>
</tr>
</tbody>
</table>

This grid can be represented as a 2-D integer array:

```
DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALY []

SET maze TO [ [ ] ] * 9 # array with 9 elements, each an empty array
DECLARE maze AS ARRAY OF ARRAY OF INTEGER INITIALY []

#This loop fills the 2-D array with the value -1.

SET maze TO [ [ ] ] * 9 # array with 9 elements, each an empty array
FOR counter FROM 0 TO 8 DO
    maze[ counter ] = [ 0 ] * -1

    # Update each element to be a 4-element array of -1s
END FOR
```

Once a 2D array, or array of arrays has been initialised, if you wanted to print out its contents as a table it could be done with this code:

```
FOR column FROM 0 TO 8 DO
    FOR row FROM 0 TO 3 DO
        SEND maze[column][row] TO DISPLAY
          END FOR
    END FOR
<print new line>
END FOR
```
This computational construct is often referred to as a **nested loop**.

We can now set the cells which need to contain a room number with a set of statements.

```plaintext
SET maze[0][1] TO 1
SET maze[1][1] TO 2
SET maze[1][2] TO 4
SET maze[2][2] TO 5
SET maze[2][3] TO 1 .... etc.
```

Once this set of commands is complete, the result of moving from room to room can be coded using a procedure.

```plaintext
PROCEDURE changeRoom(INTEGER room, INTEGER direction)

DECLARE newRoom INITIALLY 0
IF maze[room][direction] = -1 THEN
    SEND "you have hit a wall" TO DISPLAY
ELSE
    SET newRoom TO maze[room][direction]
    SEND "You are now in room "& newRoom TO DISPLAY
END IF
SET room TO newRoom
END PROCEDURE
```

**Activity: 2-D Array**

Implement this maze in your chosen programming language. You will need to tell the user where they are when the program starts, (room 0) and inform them when they reach the end of the maze (room 8).

This is the basic structure of many text based adventure games. Room descriptions and contents could be added as parallel string arrays, with additional procedures to allow the player to interact with items or characters in the rooms.

..........................................

© HERIOT-WATT UNIVERSITY
2.3 Records and arrays of records

Learning objective
By the end of this section you will be able to:

• implement an array of records in your chosen programming language.

You will remember from the Higher course that a record structure is one which can contain variables of different data types, just as a record in a database can be made up of fields of different types.

For example, storing the product information for items on sale on a website might need the following information to be stored:

• productName
• stockNumber
• price

<table>
<thead>
<tr>
<th>productName</th>
<th>stockNumber</th>
<th>Price in £</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB Cable</td>
<td>624</td>
<td>1.74</td>
</tr>
<tr>
<td>HDMI Adaptor</td>
<td>523</td>
<td>5.00</td>
</tr>
<tr>
<td>DVD-RW pack</td>
<td>124</td>
<td>10.99</td>
</tr>
</tbody>
</table>

For each item this information could be stored as a record:

```
RECORD product IS{STRING productName, INTEGER stockNumber, REAL price}
```

Now that we are storing the information about each product as a single record, a product inventory could be created to store the information about these products as an array of records.

```
DECLARE productInventory AS ARRAY OF product INITIALLY []

SET productInventory[0] TO { productName = "USB cable", stockNumber = 624, price = 1.74 };
SET productInventory[1] TO { productName = "HDMI adaptor", stockNumber = 523, price = 5.00 };
```
This code would display the contents of the product inventory and the total value of stock held.

```
DECLARE totalValue INITIALLY 0

FOR counter FROM 0 TO 2 DO
    SEND productInventory[counter].productName TO DISPLAY
    SEND productInventory[counter].stockNumber TO DISPLAY
    SEND productInventory[counter].price TO DISPLAY
    SET totalValue TO totalValue + productInventory[counter].price
END FOR

SEND "Total value of stock held = £" & totalValue TO DISPLAY
```

**Activity: Records and arrays of records**

Implement the code above in your chosen programming language but with an additional Boolean field in the record structure indication whether the stock is to be reordered or not.

Your program should present each item in turn and record the user's choice. When this is complete it should present them with a list of items to be re-ordered.

```
..........................................
```

### 2.4 Linked Lists

**Learning objective**

By the end of this section you will be able to:

- implement a linked list in your chosen programming language.

A **linked list** is a set of items organised sequentially just like an array, but where each item includes a link to the next item in the list.

**Advantages:**

- a linked list is a dynamic data structure, which means that its size is not fixed, and can grow and shrink during execution;
- a linked list is very flexible as the order of items in a linked list can be changed without actually moving any data around, just the links between them;
- A linked list is much more memory efficient than an array because it only needs to be as large as the number of items to be stored, not as large as the total possible number of items to be stored.
Disadvantage:

- With a linked list structure it is not possible to identify a specific item directly using its index in the way that you can when storing data in an array. To identify the position of an item in a linked list you have to walk through the list from beginning to end.

An entry in a linked list will consist of a piece of data and a pointer to the next item.

![Linked List Diagram]

In the example above, **deleting** item B from the list would be achieved by changing the link from item A to point to item C. As a result the memory used by item B would then be marked as free space.

**Inserting** an item X between C and D would be achieved by changing the link of the pointer from C to point to the memory location of X and then linking the pointer from X to the memory location of D.

Similarly **adding** an item to the end of the list can be done by redirecting the link from D to the new item and linking the new item to NUL. Adding an item to the beginning of the list can be done by redirecting the Head of the list to the new item and pointing its link to the memory location of item A.

Your chosen programming language may well have a built in linked list data type.

Although some programming languages may provide the option of a dynamic array which can be re-sized during run time, this is a very inefficient use of memory because when an array is re-sized, what actually happens is that the original array has to be copied into a new block of memory to accommodate the extra items which have been added. During this process both the original array and the new copy must exist in memory. A linked list is much more memory efficient since it allows items to be added or removed simply by changing the pointers associated with at most two elements.

Linked lists are useful for implementing dynamic data structures like **queues** and **stacks**.
The following algorithm describes how items are inserted and deleted from a linked list:

```
PROCEDURE setupList()
    DECLARE data INTEGER INITIALLY 0
    DECLARE head POINTER INITIALLY NULL
    # POINTER is a link to a memory location#
END PROCEDURE

PROCEDURE newNode(INTEGER data, POINTER next)
    IF head = NULL THEN
        <point head to data>
        Next = NULL
    ELSE
        <check each node in turn until the one pointing to NULL is found>
        <point last node to data>
        next = NULL
    END IF
END PROCEDURE

PROCEDURE deleteNode(INTEGER data)
    IF head = NULL THEN
        SEND "List is empty" TO DISPLAY
    ELSE IF
        <head points to data>
        THEN
        <point head to next item in list>
    END IF

    DECLARE found INITIALLY FALSE

    REPEAT
        <loop through items>
        IF
            <current item = data>
            THEN
                <Point current item to next item in list>
                SET found TO true
            END IF
        UNTIL current item pointer = NULL
        IF found = false THEN
            SEND "not found" TO DISPLAY
        END IF

END PROCEDURE
```
Activity: Linked lists

Find out how your programming language deals with linked lists and whether or not they have a predefined linked list class.

2.4.1 Linked Lists - extension material

This code uses a programming technique known as recursion which is covered in the Computational Constructs Topic. You may wish to return to this section once you have completed the Computational constructs topic.

A recursive definition of a linked list would be:

```
RECORD List IS { INTEGER value, List next}
```

The following recursive functions and procedures can be used to manipulate linked lists.

```
# displaying a list to screen

PROCEDURE displayList(ARRAY OF List myList)
    IF myList = {} THEN
        SEND "List is empty" TO DISPLAY
    ELSE
        SEND myList.value & " " TO DISPLAY
displayList(myList.next)
    END IF
END PROCEDURE

# Deleting an item from a list

PROCEDURE delete(INTEGER position, List myList)
    DECLARE newPosition INITIALLY 0
    IF position = 1 THEN
        SET myList TO myList.next
    ELSE
        SET newPosition TO position -1
        Delete (newPosition, myList.next)
    END IF
END PROCEDURE

# Finding the length of a list

FUNCTION listLength(List myList) RETURNS INTEGER
    IF myList = {} THEN
        RETURN 0
    ELSE
        RETURN listLength(myList.next) + 1
    END FUNCTION
```
#Inserting an item at a specific position in a list

FUNCTION insert(INTEGER item, INTEGER position, List myList) RETURNS List

    IF myList = {} THEN
        RETURN {item,{}}
    ELSE
        IF position = 0 THEN
            RETURN {item, myList}
        ELSE
            RETURN insert(item, position -1, myList.next)
        END IF
    END IF
END FUNCTION

The record definition of a list defines a list recursively as consisting of the item at the head of the list (value) and the list which is the rest of the list (next) often called the tail of the list.

The displayList procedure checks to see if the list is empty, otherwise it prints the first item in the list (the head) then calls displayList with the rest of the list (the tail). Each time displayList is called the head of the remaining list is printed and the procedure is called again with what is left of the list (the tail). Eventually displayList is called with an empty list and the procedure terminates.

The delete procedure works in a similar way. If the item at the head of the list (position 1) is to be deleted then what is left is the tail of the list. If the item at any other position is to be deleted then the procedure is called recursively with smaller and smaller tails until the item to be deleted is at position 1 of that tail.

The listLength procedure uses the fact that an empty list has length 0, and the rule that length of any other list is 1 + the length of the tail of that list.

The insert function returns a list using the fact that if the list is empty then the item to be inserted will be at its head. Otherwise the function is called recursively with smaller and smaller tails until the item is inserted at the head of that tail.

2.5 Stacks

Learning objective

By the end of this section you will be able to:

- implement a stack in your chosen programming language.

A stack is a data structure which behaves like a vertical list where data elements can be added or deleted only from the top of the stack. The terms push and pop refer to adding and removing items respectively. A stack is often referred to as a "Last in First Out" structure.
The stack is useful where a situation calls for the most recent item to be accessed such as in programming where procedures are called. Each procedure call is placed on the stack pending processing, and removed as procedures have completed their operations. Stacks are also used to store the code produced during the compilation of high level languages and are used to store the intermediate results of calculations.

When a running program is interrupted by a higher priority task, the status of the program and the contents of all the registers are stored in memory and that memory location is placed on top of the stack. If the interrupt itself is interrupted by an even higher priority task, then exactly the same process is repeated: its data is stored and the memory location is placed on the stack. When the current running process is complete, the memory location at the top of the stack is read from the stack and the interrupted program can then retrieve its data and continue from where it was before it was interrupted.

A stack may be represented in computer memory as a 1-dimensional array or list with
the bottom at a fixed location and a variable stack pointer to the current top location which is movable.

**Stack errors**

A stack will have a limit to how many items it can contain. If you have ever encountered a "Stack Overflow" error then your program has attempted to place too many items on the stack. The use of recursion in programming imposes significant overheads on stack manipulation, in some cases leading to stack overflow. A corresponding "Stack Underflow" error occurs when there is nothing left on the stack to retrieve.

**Activity: Push and Pop within a stack**

**Q6:** Three variables: A, B and C are pushed on to a stack in the order given. Three pop operations are then carried out. State the variables in the order they would be retrieved.

.................................

**Q7:**

Consider the integers 16, 27, 8, 55 and 12. If the number 35 is to be added to the list then it is pushed onto the top of the stack. Draw the stack structure which includes the new number.

.................................
Activity: Object oriented stack

The following algorithm is an object oriented programming approach to implementing a stack.

```haskell
CLASS Stack IS ARRAY OF INTEGER stack,
    INTEGER stackPointer,
    INTEGER size

METHODS

CONSTRUCTOR (INTEGER size)
    DECLARE THIS.stack INITIALLY []*size
    DECLARE THIS.stackPointer INITIALLY 0
    DECLARE THIS.size INITIALLY size
END CONSTRUCTOR

PROCEDURE push(INTEGER Value)
    IF THIS.stackPointer=THIS.size THEN
        <stack overflow action>
    ELSE
        SET THIS.stack[THIS.stackPointer] TO value
        SET THIS.stackPointer TO THIS.stackPointer+1
    END IF
END PROCEDURE

FUNCTION pop() RETURNS INTEGER
    IF THIS.stackPointer=0 THEN
        <stack underflow action>
    ELSE
        SET THIS.stackPointer TO THIS.stackPointer -1
        RETURN THIS.stack[THIS.stackPointer]
    END IF
END FUNCTION

END CLASS
```

The constructor function runs when an object is created from a class. It creates the stack structure of the required size, and sets max to be maximum number of elements the stack can hold (from the size parameter passed as a parameter).

The push procedure detect trying to write to a full stack (a stack overflow error). Otherwise, if the stack isn't full we can push the value onto the stack and update the stackPointer to the new top of the stack.

The pop function detect if is the stack is empty, if it is then it catches the stack underflow error, otherwise if the stack has elements in it then it returns the value and moves the stackPointer to show there is one less item in the stack.
Implement the `pop` and `push` sub programs from this algorithm in your chosen programming language.

2.6 Queues

**Learning objective**

By the end of this section you will be able to:

- implement a queue in your chosen programming language.

A queue is also a 1-dimensional array or list, but data items are inserted and retrieved at different ends. A queue is a "First in First Out" structure. In a queue two pointers are required: one to point to the head of the queue and one to point to the end. An important aspect to realise here is that the data itself does not move but merely the pointers to the head and end of the queue. Queues are used when multiple printing jobs have to be processed and also during the scheduling of tasks in a multitasking environment. It is also an important structure in event-driven languages where events are placed in a queuing system to wait being processed.

1. 88 is added to the end of the queue;
2. Rear pointer = Rear pointer + 1;
3. 17 is removed from the queue;
4. Start pointer = Start pointer + 1;
5. 39 is removed from the queue;
6. Start pointer = Start pointer + 1;

Suppose now that 27 and 13 are now added to the queue. At this point the rear pointer is referencing the end of the queue. We now want to add the value 10.
The situation would be:

There are no free locations at the end of the array, but at the start of the array there are free locations. The value 10 can be added there, with pointers updated appropriately.

The array is now treated as a **circular queue**, with the first element following on from the last element in the array. A circular queue arrangement is such that when there is no space at the bottom of the queue you simply start again at the top. The rear pointer now indicates the item 10 which is now at the end of the queue, while the start pointer is unchanged.
Activity: Queue

The following algorithm is an object oriented programming approach to implementing a queue.

```java
CLASS Queue IS {INTEGER start, INTEGER rear, INTEGER currentSize, INTEGER maxSize, ARRAY OF INTEGER queue}

METHODS

CONSTRUCTOR Queue(INTEGER size)
    DECLARE THIS.start INITIALLY 0;
    DECLARE THIS.rear INITIALLY 0;
    DECLARE THIS.currentSize INITIALLY 0
    DECLARE THIS.maxSize INITIALLY size;
    DECLARE THIS.queue INITIALLY INTEGER[maxSize]
END CONSTRUCTOR

PROCEDURE join(INTEGER data)
    IF THIS.currentSize = THIS.MaxSize THEN
        SEND "Queue Overflow" TO DISPLAY
    ELSE
        SET THIS.queue[THIS.rear] TO data
        SET THIS.rear TO rear + 1
        SET THIS.currentSize TO THIS.currentSize + 1
    END IF
    IF THIS.rear > THIS.maxSize THEN
        SET THIS.rear TO 1
    END IF
END PROCEDURE

FUNCTION leave() RETURNS INTEGER
    IF THIS.currentSize = 0 THEN
        SEND "Queue Underflow" TO DISPLAY
        RETURN 0
    ELSE
        RETURN THIS.queue[THIS.start]
        SET THIS.queue[THIS.start] TO 0
        SET THIS.currentSize TO THIS.currentSize - 1
        SET THIS.start TO THIS.start + 1
    END IF
    IF THIS.start > THIS.maxSize THEN
        SET THIS.start TO 1
    END IF
END FUNCTION

END CLASS
```

The constructor function runs when an object is created from a class. It creates the queue structure of the required size with start and rear pointers.
The **join** procedure checks to ensure that the queue is not full, otherwise it adds the data passed as a parameter to the rear of the queue and updates the rear pointer and **currentSize**. If the rear position is greater than **maxSize**, then the rear pointer is set back to 1.

The **leave** function checks to ensure that the queue is not empty, otherwise it returns the item at the start position, updates **currentSize** and the start pointer. If the start position is greater than **maxSize**, then the start pointer is set back to 1.

Implement the Queue program from this algorithm in your chosen programming language.

..........................................

**Activity: Queue with one pointer**

This is a version of a Queue which uses a single pointer and shuffles the contents of the array as items leave and join instead of moving the pointers. It is less efficient, but behaves more like a real life queue.

```plaintext
CLASS Queue IS {ARRAY OF INTEGER queue, INTEGER queuepointer, INTEGER size} WITH

CONSTRUCTOR Queue(INTEGER queueSize)
    DECLARE THIS. size INITIALLY queueSize
    DECLARE THIS.queue INITIALLY [size]
    DECLARE THIS.queuepointer INITIALLY 0
END CONSTRUCTOR

PROCEDURE join(INTEGER value)
    IF THIS.queuepointer=THIS.size THEN

        <queue overflow action>

    ELSE
        SET THIS.queue[THIS.queuepointer] TO value
        SET THIS.queuepointer TO THIS.queuepointer+1
    END IF
END PROCEDURE
```
FUNCTION leave() RETURNS INTEGER
    DECLARE result INITIALY <whatever>
    IF THIS.queuepointer=0 THEN
        <queue underflow action>
    ELSE
        SET result TO THIS.queue[0]
        FOR i FROM 0 TO THIS.queuepointer-2 DO
            SET THIS.queue[i] TO THIS.queue[i+1]
        END FOR
        SET THIS.queuepointer TO THIS.queuepointer-1
    END IF
    RETURN result
END FUNCTION

END CLASS

If you prefer this version, implement the algorithm in your chosen programming language.

..........................................

2.7 Array of Objects

Learning objective

By the end of this section you will be able to:

- implement an array of objects in your chosen programming language.

An array of objects behaves just like an array of records would.

This example is similar to the Playlist example in Topic 1, but uses a simpler track class.

First we can declare a Track class:

CLASS Track IS {STRING title, STRING artist }

METHODS

PROCEDURE showTrack()
    SEND THIS.title & " " & THIS.artist TO DISPLAY
END PROCEDURE

END CLASS
Now we can use the track class to declare a Collection class which is an array of track objects:

```plaintext
CLASS Collection IS {ARRAY OF Track tracks}

METHODS

CONSTRUCTOR Collection (INTEGER size)
    DECLARE tracks INITIAL []
    DECLARE size INITIAL 10
END CONSTRUCTOR

PROCEDURE showTracks()
    FOR counter FROM 0 TO size DO
        tracks[counter].showTrack()
    END FOR
END PROCEDURE

END CLASS

We can build up a new collection array using the Collection class:

```plaintext
DECLARE sixtiesTracks INITIAL Collection(10)
```

```plaintext
SET sixtiesTracks[0] TO {"Sugar Sugar", "The Archies"}
SET sixtiesTracks[1] TO {"My boy lollipop", "Millie"}
SET sixtiesTracks[2] TO {"Walking back to Happiness", "Helen Shapiro"}

...

SET sixtiesTracks[9] TO {"Twist and Shout", "The Beatles"}
```

And then we could list the tracks using the showTracks procedure:

```plaintext
sixtiesTracks.showTracks
```

Implement this algorithm in your chosen programming language, using your own choice of music.
2.8 Conclusion

Summary

You should now know:

- A 2-D array is an array of arrays, but can also be implemented in some languages as an array with two indexes which can be used to store data which is best understood as a grid with two axes.

- A 2-D array can be accessed using two fixed loops, one inside the other. This is sometimes called a nested loop.

- A record is a data structure consisting of several variables of different types, and an array of records is an array containing a set of records.

- Linked lists are dynamic data structures where data items have a link to the next item in the list. This means that the order of data items can be changed just by moving links, not the data itself.

- A stack is an array of data items which can only be accessed from the top.

- A stack Overflow is an error caused when trying to push data onto a full stack.

- A stack Underflow is caused when trying to pop data off an already empty stack.

- A stack is a First In Last Out (LIFO) data structure.

- A queue is an array of data items with pointers which refer to the first and last items in the queue.

- A queue is a First In, First Out (FIFO) data structure.
2.9 End of topic test

End of topic test

**Q8:** Describe the three variables required for the implementation of a stack class.

**Q9:** The following diagram represents a queue which can hold a maximum of 7 items:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>45</td>
<td>7</td>
<td>23</td>
<td>94</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Front = 1  
Rear = 4  
What would the values of Front and Rear be after the following operations:

Join 3  
Leave  
Join 47

a) Front = 0, Rear = 4  
b) Front = 6, Rear = 3  
c) Front = 6, Rear = 4  
d) Front = 2, Rear = 6

**Q10:**  
What would the result of the next leave operation be after the following operations:

Join 56  
Join 35  
Leave

a) 23  
b) 47  
c) 3  
d) 7
Q11: What would the values of Front and Rear be after this leave operation?

a) Front = 3, Rear = 0
b) Front = 0, Rear = 6
c) Front = 2, Rear = 3
d) Front = 2, Rear = 4

Q12: A string array has been declared to hold passwords for rooms in an adventure game.

\[
\text{DECLARE stringArray AS ARRAY OF ARRAY OF STRING INITIALLY []}
\]
\[
\text{SET stringArray TO [ [ ] ] } \ast 7
\]

If we wanted to use this array to store a \( 5 \times 7 \) grid of text values, correct this pseudocode to fill it initially with the word "password" using variable names "row" and "column" to identify the indexes.

Q13: What command would you use to print out the third row in the grid?
Topic 3

Development methodologies

Contents

3.1 Revision ................................................................. 52
3.2 Traditional versus contemporary methodologies ................... 53
3.3 Problem decomposition ............................................. 53
3.4 Iterative prototyping ................................................... 55
3.5 Contemporary development methodologies ......................... 55
   3.5.1 Spiral development .............................................. 55
   3.5.2 Release early, release often .................................... 56
   3.5.3 SCRUM .......................................................... 56
   3.5.4 Extreme programming ........................................... 56
   3.5.5 Model-Driven Software Development (MDSD) .............. 56
   3.5.6 Other contemporary methodologies ......................... 57
3.6 Conclusion .................................................................. 57
3.7 End of topic test ....................................................... 58

Prerequisite knowledge

From your studies at Higher you should already know about:

- the seven stages in the traditional software development process: analysis, design, implementation, testing, documentation, evaluation, and maintenance;
- the Top-down Design / Stepwise Refinement approach to software development;
- Rapid Application Development (RAD) and Agile development methodologies.

Learning objectives

By the end of this topic you will have studied:

- problem decomposition;
- Iterative prototyping;
- a number of other contemporary development methodologies.
3.1 Revision

Quiz: Revision

Q1: During the software development process, who is responsible for converting the design into actual program code?
   a) Programmers
   b) System Analyst
   c) Independent Test Group
   d) Client

Q2: Software is evaluated according to a number of different criteria. Which one of the following would not be included in an evaluation report?
   a) Portability
   b) Efficiency
   c) Editability
   d) Maintainability

Q3: Which one of these is NOT an advantage of Agile software development?
   a) Reduced development time.
   b) Responsiveness to changed circumstances.
   c) Reduced costs.
   d) A reduction in the need for teams to communicate about their activities.

Q4: Which statement is the best description of pseudocode?
   a) A list of pre-written subroutines in the program.
   b) An informal list of testing instructions within the code.
   c) A list of statements in a high level language.
   d) A high-level description of how a computer program functions.

Q5: Which of these has the waterfall model stages in the right order?
   a) Analysis, design, implementation, documentation, testing, evaluation, maintenance.
   b) Analysis, design, implementation, testing, documentation, evaluation, maintenance.
   c) Analysis, design, evaluation, implementation, testing, documentation, maintenance.
   d) Analysis, design, implementation, documentation, evaluation, testing, maintenance.
Q6: Who is responsible for writing the software specification during the software development process?

a) Systems Analyst
b) Programming team
c) Client
d) Beta Testers

3.2 Traditional versus contemporary methodologies

The traditional software development model assumes a linear progression through the analysis, design, implementation, testing, documentation, evaluation, and maintenance stages. Using this methodology, the client would only be involved during analysis, testing and evaluation. Agile software development is the term used for a group of software development methods which focus on delivering working programs early in the development of a project and use iterative methods to constantly improve this software based on user feedback.

Many modern software development methodologies use a combination of both approaches and range across a spectrum from the traditional "waterfall" approach at one end to the so-called lightweight approaches such as Agile development and extreme programming at the other end. In the past, the kinds of project which these methodologies are associated with tended to follow the same pattern, with traditional methods being used in large scale projects and the more lightweight methodologies being used in smaller ones. There has however been an increased interest in Agile methods with many large government and technology companies now deploying Agile methods for all software projects.

3.3 Problem decomposition

Learning objective

By the end of this section you will be able to understand:

- Problem decomposition.

Decomposition is the process of simplifying a problem by working out what information it needs (the data structures) and breaking it down into smaller sub-problems which use a subset of that information in order to be solved. You will already be familiar with this approach to solving complex programming problems from your studies at Higher, known as top-down analysis. Decomposition is often the first approach to a complex problem as the sub-problems identified will hopefully look similar to previously encountered ones.

In a library borrowing system for example, the first approach is to identify the information needed to solve the problem: we need to store information about books, and information
about members.

To identify the sub problems we need to think about the various operations which are associated with this information: Books are borrowed, returned, requested, searched for, added to the library and removed from the library. Members join the library, change their details, leave the library (or die), borrow, and return books. Each of these operations will need a subset of the information required for the whole system.

Once the sub-problems have been identified, some of them will look familiar. You will have used a linear search algorithm in Higher computing, so `findMember()` and `findBook()` are tasks you have encountered before. Here are class diagrams for the Library, Member and Book classes.

![Class Diagrams](image)

**Activity: Sub tasks**

Consider a school which has teachers, students and courses. If you were asked to build a management system which allowed administrators to manage the information about who taught what and who attended which courses, how would you break this down into smaller sub tasks?
3.4 Iterative prototyping

Learning objective
By the end of this section you will be able to understand:

- Iterative prototyping.

Iterative prototyping is an approach which evolved from the Agile development community. Many software projects will be able to be broken down into separate parts which can be worked on independently for at least some of the lifetime of the development phase. These separate parts can be developed as prototypes which the client can see and comment upon. The developer can then adapt the prototypes according to the client's wishes and re-submit further versions until the client is happy with that part of the program. This is an iterative process in that changes are made to the prototypes in the light of comments or problems spotted with previous versions.

The advantages of Iterative Prototyping are:

- The client is involved throughout the development process. The finished project should not present them with any unwelcome surprises.
- It is easier to change parts of the project if client feedback is received early on in the development.
- Although the fate of a prototype is by definition something to be discarded once the finished article is produced, some prototypes may evolve to be part of the working system. The SCRUM management system uses a system where small working prototypes of the software are released at the start with only a few of the required features and gradually build into the complete program.

3.5 Contemporary development methodologies

Learning objective
By the end of this section you will be able to describe:

- a number of other contemporary development methodologies.

If you check Wikipedia you will find a long list of software development methodologies which fall along the heavyweight - lightweight spectrum. This section will cover a few of the more well-known ones.

3.5.1 Spiral development

This is a system which attempts to combine the best aspects of the waterfall model and iterative prototyping. Development of a product involves a cycle through the process, effectively creating a series of prototypes, but developing these prototypes using the waterfall model, hopefully avoiding mistakes being made and not spotted until late on in
the development process.

### 3.5.2 Release early, release often

As its name suggests this approach is at the lightweight end of the spectrum. This approach emphasizes the importance of early and frequent releases so that users can feedback frequently and developers are kept in touch with their needs. The obvious disadvantages are the difficulties of controlling the existence of older versions of the software and the likelihood that users will experience crashes and need to constantly update their software. The advantage is the users access to the developers and their influence on the final product.

### 3.5.3 SCRUM

SCRUM is a management system for an iterative development methodology with an emphasis on customer participation in the development process. Working software, called “releases” are produced regularly and meetings between client and developers are scheduled frequently. The client effectively becomes part of the development team. The Product Owner (who is typically a representative of the client) joins the development team to represent the clients/users views.

The first topic from Information System Design and Development goes into a lot of detail about SCRUM.

### 3.5.4 Extreme programming

Extreme programming is similar to SCRUM, but with shorter cycles of prototyping and client feedback. Extreme programming uses small groups of developers who both design and code.

### 3.5.5 Model-Driven Software Development (MDSD)

Model Driven Software Development (MDSD) is a style of software development seen as an alternative to the traditional style of programming. Model Driven Software Development concentrates on building models of a software system using a diagrammatic design notation using tools like the Unified Modelling Language (UML). We will look at UML in more detail in the Design notations topic. The idea is that you use these diagrams, to specify your system and they can then be used to generate code in a conventional programming language.
3.5.6 Other contemporary methodologies

- Cowboy coding
- Kanban
- Bottom up development
- Test-driven development.

You can see articles on these and many other software development methodologies here:

Activity: Contemporary software development methodology

Choose a contemporary software development methodology from the list above and research its characteristics. Summarise them to present to other members of your group.

3.6 Conclusion

Summary

- Decomposition is the process of breaking a complex problem down into smaller more easily solvable problems. Used in all software development methodologies.
- Modern software development methodologies use a combination of both traditional and Agile approaches.
- Many software projects will be able to be broken down into separate parts which can be worked on independently and developed as prototypes.
- Prototypes allow the client to be involved throughout the development process.
- Modern software development methodologies are more flexible and responsive to changes in circumstances than the traditional waterfall model.
3.7 End of topic test

End of topic test

Q7: Which of these software development methodologies is least likely to involve the client with the coding and testing stage of a project?

a) SCRUM  
b) Waterfall model  
c) Spiral development  
d) Cowboy coding

Q8: Which of these statements is the best definition of a prototype?

a) A completed project ready for testing.  
b) A partially completed project.  
c) A working model designed for evaluation.  
d) A scale model.

Q9: Which of these statements best describes iterative software development?

a) A development methodology which progresses in a series of stages, each one dependent on the successful completion of the previous one.  
b) A development methodology which uses prototyping.  
c) A development methodology which involves the client at all stages in the process.  
d) A development methodology which combines a number of different approaches.

Q10: Which one of these is NOT a benefit of iterative prototyping?

a) The client is involved throughout the development process.  
b) Some prototypes may evolve to be part of the working system.  
c) It is easier to change parts of the project early on in the development.  
d) Testing of the finished project can be outsourced.
Topic 4

Design notations

Contents

4.1 Revision .......................................................... 60
4.2 Pseudocode ...................................................... 61
4.3 Unified Modelling Language (UML) .......................... 65
  4.3.1 Use Case diagrams ........................................ 65
  4.3.2 Class diagrams ............................................ 67
  4.3.3 Sequence diagrams ...................................... 71
4.4 Wireframing ...................................................... 72
4.5 Conclusion ....................................................... 74
4.6 End of topic test ................................................ 75

Prerequisite knowledge

From your studies at Higher you should already know:

- how design notations can help the software development process;
- about a variety of graphical program design notations including structure diagrams;
- data flow diagrams and wireframes;
- the relationship between pseudocode and source code;
- how to interpret examples of pseudocode.

Learning objectives

By the end of this topic you should be able to use the following design notations associated with programming paradigms:

- Pseudocode (imperative programming);
- Unified Modelling Language (object-oriented programming);
- Wireframing (interface design).
4.1 Revision

Quiz: Revision

Q1: The design approach of breaking a large and complex problem into smaller, more manageable sub-problems is known as:
   a) Process refinement
   b) Top-down refinement
   c) Bottom-up design
   d) Top-down design

Q2: Which of these is NOT a graphical design notation?
   a) Wireframe
   b) Structure diagram
   c) Pseudocode
   d) Data flow diagram

Q3: Which design notation needs the data structures to be decided on before it can be drawn?
   a) Wireframe
   b) Structure diagram
   c) Pseudocode
   d) Data flow diagram

Q4: Which design notation would you use to design a user interface?
   a) Wireframe
   b) Structure diagram
   c) Pseudocode
   d) Data flow diagram

Q5: Which design notation is the easiest to create source code from?
   a) Wireframe
   b) Structure diagram
   c) Pseudocode
   d) Data flow diagram
Q6: Stepwise refinement is?

a) creating pseudocode from a structure diagram and data flow diagram.
b) breaking a large and complex problem into smaller, more manageable sub-problems until they become trivial.
c) writing source code.
d) creating a wireframe interface design.

4.2 Pseudocode

Learning objective

By the end of this section you should be able to use the following design notation associated with programming paradigms:

- Pseudocode (imperative programming).

Pseudocode is a method of describing an algorithm in a way which makes it easy to understand and ultimately convert into source code in whatever language the programmer wishes to use. Pseudocode is often described as an informal description of an algorithm, and can be written in a wide variety of styles. We have standardised the pseudocode we use in this course on the Haggis specification because that is the style used in SQA assessment materials. Because Haggis pseudocode has a formal specification it is possible to build a syntax checker for any algorithm written using it. This is not normally an option for pseudocode. Because of its formal nature, Haggis is usually described as a reference language. If you are interested you can see a syntax checker for the Haggis reference language here: http://haggis4sqa.appspot.com/haggis Parser.html
You will have already seen many examples of pseudocode in the Higher course. This example describes the linear search algorithm to find an item in an array of numbers and give its index position.

PROCEDURE linearSearch(ARRAY OF INTEGER numbers, INTEGER highestIndex)

DECLARE itemToFind INITIALLY getvalidItem(1,100)

DECLARE found AS BOOLEAN INITIALLY false
DECLARE counter INITIALLY 0

REPEAT
    SET found TO numbers[counter] = itemToFind
    SET counter TO counter + 1
UNTIL found OR counter > highestIndex

IF found THEN

    SEND itemToFind & " found at position" & (counter-1) TO DISPLAY
ELSE SEND "Item not found" TO DISPLAY
END IF

END PROCEDURE
You will have seen a similar style of pseudocode used in this course where the object-oriented programming paradigm is used. This is an example of the same algorithm in that style.

```
CLASS Linear IS {ARRAY OF INTEGER numbers, INTEGER length}

METHODS

FUNCTION getValidItem(lowerLimit, upperLimit)RETURNS INTEGER
    RECEIVE userInput FROM (INTEGER) KEYBOARD
    WHILE userInput < lowerLimit OR userInput > upperLimit DO
        SEND ["Input must be between " & lowerLimit & " and " & upperLimit] TO DISPLAY
        RECEIVE userInput FROM (INTEGER) KEYBOARD
    END WHILE
    RETURN userInput
END FUNCTION

PROCEDURE linearSearch()
    DECLARE itemToFind INITIALLY getValidItem(1,100)
    DECLARE found AS BOOLEAN INITIALLY false
    DECLARE counter INITIALLY 0
    REPEAT
        SET found TO THIS.numbers[counter] = itemToFind
        SET counter TO counter + 1
    UNTIL found OR counter > THIS.length
    IF found THEN
        SEND itemToFind & " found at position" & (counter-1) TO DISPLAY
    ELSE SEND "Item not found" TO DISPLAY
    END IF
END PROCEDURE

END CLASS
```

Pseudocode is different from source code because code specific details are not usually included:

- the syntax for input from the keyboard and output to screen will vary from one language to another;
- variables do not necessarily need to be declared in advance of their use. Some languages require this, in other languages, it is optional;
- some languages require all commands to end with a specific symbol like a semi-colon. In others white space has a syntactic value;
- the syntax for assigning a value to a variable varies from one language to another, and in some cases assigning a value determines the data type for that variable;
- control structures like If ... Then ... Else, Repeat ... Until, While ... Do will vary in syntax from language to language.
- pseudocode may include natural language text describing a process which needs to be expanded. This is known as elision. In the Haggis reference language elision is identified by appearing between the < and > symbols.

**Activity: Pseudocode**

Use the linear search pseudocode above to implement the algorithm in two different procedural programming languages. Compare your two programs with the pseudocode in terms of their syntax. The semantics in each case should be identical.

Pseudocode can be used at different levels of detail in the top-down analysis of a problem. Sub problems can be identified but not expanded in the top level algorithm. In this example from the Development methodologies topic, the sub-problem of asking the customer for their purchase choice has not been expanded whereas the Setup and Calculate procedures have been fully set out.

```plaintext
PROCEDURE main()
  Setup
  <Ask customer for choice>
  Calculate
END PROCEDURE

RECORD Product IS{STRING productName, INTEGER stockNumber,
  REAL price, BOOLEAN purchased}

PROCEDURE setup()
  DECLARE shoppingBasket AS ARRAY OF Product INITIALLY []
  SET shoppingBasket[0] TO {productName = "USB cable",
    stockNumber = 624, price = 1.74, purchased = false}
  SET shoppingBasket[1] TO {productName = "HDMI adaptor",
    stockNumber = 523, price = 5.00, purchased = false}
  SET shoppingBasket[2] TO {productName = "DVD-RW pack",
    stockNumber = 124, price = 10.99, purchased = false}
END PROCEDURE

PROCEDURE calculate()
  DECLARE total INITIALLY 0
  FOR counter FROM 0 TO 2 DO
    IF shoppingBasket[counter].purchased = true THEN
      SET total TO total + shoppingBasket[counter].price
    END IF
  END FOR
  SEND "Total cost of items = £" & total TO DISPLAY
END PROCEDURE
```

**Activity: Pseudocode 2**

Expand the <ask customer for Choice> pseudocode code in the algorithm above.
4.3 Unified Modelling Language (UML)

**Learning objective**

By the end of this section you should be able to use the following design notation associated with programming paradigms:

- Unified Modelling Language (object-oriented programming).

UML is a design notation created specifically to help programmers design systems using the object-oriented programming paradigm. As its name suggests, it is a formal system for creating models for software design. UML provides a specification for a number of different types of diagram which can be used to represent a software system. Just like pseudocode, UML is programming language independent - it does not dictate which language will be used to translate its diagrams into source code. UML tells you what model elements and diagrams are available and the rules associated with them. It does not tell you what diagrams to create.

We are going to look at three of the many diagrams specified by UML:

- Use Case diagrams
- Class diagrams
- Sequence diagrams.

### 4.3.1 Use Case diagrams

Use case diagrams contain **Actors** (the people or entities who interact with the system) and **Use Cases** which are the procedures which they interact with.

The **Use Cases** in the diagram are identified by the circled text and the **Actor’s** interactions with them are identified by the arrows.
A large system will have several different actors. This example might be part of a sales recording system in a shop.

The first stage in creating a Use Case diagram is identifying the actors, by classifying the people who interact with the system, in this case the managers, sales assistants and customers.

The next stage is to identify the data which the actors will interact with and how they change it. The scenario above assumes that the system stores the following data:

- A stock table which stores details of items in stock and their quantities.
- A customer table which stores customer details including previous purchases.
- A staff table which stores staff details and their access privileges to the system.

The Customer is involved when a sale is made. If they are a new customer then their details are added to the customer table. When they make a purchase the sale is added to their details.

The Sales assistant also interacts with the customer table but also needs to update the stock list when a sale is made.

The manager needs to be able to update the staff table if a new sales assistant joins the company, add new products to the stock table and generate reports of sales.
4.3.2 Class diagrams

Class diagrams show the classes of objects required for the system and how they are related. Each class object will describe the instance variables and methods for that class.

This might be the use case diagram for the playlist manager we considered in the previous Programming paradigms topic.

Class diagrams show the classes of objects required for the system and how they are related. Each class object will describe the instance variables and methods for that class.

In the previous Programming paradigms topic, we defined two classes, a track class and a playlist class.
Suppose we wanted to define a new type of track:

```plaintext
CLASS HendrixTrack INHERITS TRACK

METHODS

CONSTRUCTOR Track (STRING title, REAL length)
   DECLARE THIS.trackTitle INITIALLY title
   DECLARE THIS.artist INITIALLY "Jimi Hendrix"
   DECLARE THIS.trackLength INITIALLY 0.0
END CONSTRUCTOR

END CLASS
```

We can show the relationship between the Playlist, Track and HendrixTrack classes.

![Class Diagram](image)

The Track class is associated with the Playlist class. The HendrixTrack class inherits its properties from the Track class.

**Activity: Case and Class diagrams: Concert Booking**

A concert booking system has a number of venues and needs to be able to book performers and sell tickets. It will hold data on each venue, events, ticket sales and performers. Consider what instance variables and methods might be required for a system like this and complete these class diagrams with suitable instance variables and methods.
Consider who the actors are in this system then draw the Use Case diagram for their interaction with the data.

**Activity: Case and Class diagrams: Bank statements**

In Topic 1 you completed a Bank Statement activity using class diagrams and pseudocode to define **Statement** and **Entry classes** with methods to add an entry to a statement and to display a statement.
The activity stated that a bank statement contains an array of entries, a variable to store the maximum entries it can contain, a variable to store the next available place in the list, and a balance. An entry is a record containing a date and an amount (deposit or withdrawal).

To create a Use Case diagram for a user you need to consider the data that the actor is manipulating and their interactions with it.

Once you have decided on the interactions you are going to illustrate, you can create the class diagrams. The data will be the instance variables for the classes and the interactions will be the methods.

Draw the Use Case diagram and the Class diagrams for the Bank statement scenario where the actor is the user of the system.
4.3.3 Sequence diagrams

A sequence diagram shows interactions between objects in the system. Solid lines show calls, dotted lines show data returns.

It is important to remember that UML can be used in many contexts, both as an informal method of documenting the design of a program where an agile development methodology is appropriate but also where a formal system of diagrams is needed to design and document large software engineering projects.

Activity: Case and Class diagrams

Draw the Use Case diagram and the Class diagram for the Bank statement activity in the Programming Paradigms topic.
4.4 Wireframing

Learning objective

By the end of this section you should be able to use the following design notation associated with programming paradigms:

- Wireframing (Interface design).

The user interface can have a profound effect on the success or failure of a software project. Think of how modern operating systems have changed the way users interact with them, despite the fact that the functions of an operating system has remained substantially unchanged, and they can often still be controlled form a command line if necessary.

Wireframes are used to design navigation structures and user interfaces to software and are an essential design tool which enables both the designer to share and test their ideas, and allows the client to be part of the design process.

When creating a prototype during software development, it is often useful to be able to create the interface for an application without including any functionality other than the menu drop down or dialog box selection. This allows the client to be part of the design team, particularly when a rapid prototyping development methodology is being used.

A wireframe can be anything from a sketch outlining the user interface to a program or the navigational structure of a web page, to a detailed design which has been built using a graphics package or dedicated wireframing software. The amount of detail in a wireframe and the number of iterations it goes through in the design process will depend upon the software development methodology being used.

The process may start with a rough sketch of the interface:
Then a wireframing tool may be used to create a more detailed design:

Finally a detailed mock-up might be produced using a graphic design tool:

There are a number of stand-alone and online wireframing programs which can be used to design interfaces for web applications or stand-alone software. These programs often provide the option of design prototypes which the user can interact with to illustrate external links, menu options, dialogue boxes etc.
4.5 Conclusion

Summary

- Pseudocode is a method of describing an algorithm, normally used when designing a solution using the imperative programming paradigm.
- Pseudocode can have, but does not need to have a formally defined syntax.
- Unified Modelling Language (UML) is a formal system defining types of diagram which can be used when designing a solution to a problem using the object oriented programming paradigm.
- Use Case Diagrams, Class Diagrams and Sequence Diagrams are just a few of the diagrams contained in the UML specification.
- Wireframes are a useful design tool for the user interface to a web application or a stand-alone software.
4.6 End of topic test

End of topic test

Q7: What is the relationship between pseudocode and source code?
   a) A one to one relationship between every line of pseudocode and every line of source code.
   b) There is no relationship between pseudocode and source code.
   c) A one to many relationship between lines of pseudocode and lines of source code.
   d) Pseudocode is converted to source code but cannot always be directly translated into source code.

Q8: Which of these is NOT a program design notation?
   a) Pseudocode
   b) UML
   c) Wireframes
   d) Syntax checker

Q9: In the context of pseudocode, what is elision?
   a) Natural language segment of pseudocode which needs to be expanded.
   b) A section of pseudocode which cannot be translated into source code.
   c) A section of pseudocode with a one to one relationship to source code.
   d) A section of code missing from an algorithm.

Q10: What is an actor in a use case diagram?
   a) A person or entity that interacts with the system.
   b) An object derived from a class.
   c) A method.
   d) A relationship between a person and a system.

Q11: A class diagram illustrates?
   a) How actors interact with classes.
   b) Interactions between objects.
   c) Relationships between classes.
   d) Relationships between actors.
Q12: A use case diagram illustrates?

a) How actors interact with the system.
b) Interactions between objects.
c) Classes and their relationships.
d) Relationships between actors.

Q13: Which of these might Wireframing be used for?

a) Creating pseudocode
b) Prototype design
c) Interface Design
d) Data flow design
Topic 5

Standard algorithms

Contents

5.1 Revision . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
5.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.3 Binary search . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
  5.3.1 Binary search algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
  5.3.2 Comparing the binary search with the linear search . . . . . . . . . . 85
5.4 Sorting algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
  5.4.1 Selection sort using two lists . . . . . . . . . . . . . . . . . . . . . . . . . 88
  5.4.2 Bubble sort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
  5.4.3 Insertion sort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.5 Other sorting algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
  5.5.1 Quicksort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 101
  5.5.2 Merge sort . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102
  5.5.3 Comparing sorting algorithms . . . . . . . . . . . . . . . . . . . . . . . . 102
5.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103
5.7 End of topic test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 104

Prerequisite knowledge

From your studies at Higher you should already:

- recognise appropriate use of the following standard algorithms:
  - input validation;
  - find minimum/maximum;
  - count occurrences;
  - linear search.

Learning objectives

By the end of this topic you will be able to:

- use the following algorithms:
  - binary search;
- selection sort with two lists;
- insertion sort;
- bubble sort.

- Compare a number of different sorting algorithms.
5.1 Revision

Quiz: Revision

Q1:

This is an example of?

a) Counting occurrences.
b) Input validation.
c) Linear search.
d) Finding the maximum.

Q2:

Dim numbers(10) As Integer
Dim counter as Integer

For counter = 0 To 10
    numbers(Counter) = Int(Rnd * 10) + 1
Next counter

value = numbers(0)
For counter = 1 To 10
    If Value < numbers(counter)
        Then value = numbers(counter)
    Next counter

Print value

This is an example of?

a) Counting occurrences.
b) Input validation.
c) Linear search.
d) Finding the maximum.
Q3: The linear search algorithm uses?

a) A fixed loop and a boolean variable.
b) A conditional loop and a boolean variable.
c) A maximum value and a boolean variable.
d) A minimum value and a boolean variable.

Q4: The counting occurrences algorithm uses?

a) A fixed loop.
b) A conditional loop and a boolean variable.
c) A conditional loop.
d) A minimum value and a boolean variable.

Q5:

```
Dim numbers(10) As Integer
Dim counter as Integer

For counter = 0 To 10
    numbers(counter) = Int(Rnd * 10) + 1
Next counter

total = 0
value = 5
For counter = 0 To 10
    If Value = numbers(counter) Then
        total = total + 1
    End If
Next counter

Print total
```

This is an example of?

a) Counting occurrences.
b) Input validation.
c) Linear search.
d) Finding the maximum.
5.2 Introduction

You will have studied the linear search algorithm at Higher. A linear search is a "Brute Force" search. The algorithm is effectively:

```
REPEAT
  <Check every item in an array in turn>
UNTIL
  <the target is found>
OR
  <the end of the array is reached>
```

If you are storing data in arrays, then searching and sorting them efficiently becomes more and more important as the amount of data increases. A linear search is the only way to find an item in an array if the data is not stored in a sequential way. Once the data is sorted however, searching becomes much more efficient. We use sorted lists in many contexts: databases, libraries, timetables, indexes etc. Lists like these would be impossible to search efficiently if they were not sorted. We are going to look at the binary search algorithm, which is an algorithm to search a sorted array, and then we will look at a variety of sorting algorithms.

5.3 Binary search

**Learning objective**

By the end of this topic you should be able to use the following algorithm:

- binary search.

Although a list is sorted, from the point of view of the computer, there is no certain way of knowing what proportion of each category of item is present in the list. When looking for a book by a specific author in a library, we have some knowledge of the frequency of the letters of the alphabet at the beginning of surnames - there are few people whose name starts with X for instance. We would also look near the beginning of the shelves if the author's surname started with A. As far as the binary search is concerned however, the assumption is that the frequency of occurrence of items in the list is random, so the best approach is to look in the middle of the list first.

Once the item in the middle of the list is examined, the list can be divided into two halves - one which could contain the item being searched for and one which definitely doesn't. From then on the same technique can be used on the part of the list which contains the item being searched for and again dividing the list into two halves and so on until the position of the search item is identified. The binary search is so-called because it continuously divides a list in two.

Consider the following array of length 10, containing 10 integers in numerical order with a search key of 82. (A search key is the value we are looking for).
The list contains 10 elements so the mid point of the list is position 5 (which as an index value of 4. The value at the mid point, in this case 57, is referred to as the Mid Value. Choosing this position will produce two sub lists - a left list and a right list. Now proceed as follows:

1. Compare the mid value with the search key. 82 > 57 so ignore the left list;
2. Create a new mid location between location 4 and location 9. This location 6 (72).
3. Now compare the new mid value with the search key. 82 < 72) so ignore the left list;
4. Create new mid location between location 6 and location 9. This is location 7 (82).
5. Compare the mid value with the search key. 82 = 82 so the key is found at position 7.

The binary search is very efficient, particularly for large lists. Doubling the size of a list means only one extra split is required.
### 5.3.1 Binary search algorithm

A simple description of this algorithm would be:

```
REPEAT
  <Find the middle item in the list and compare it with the item to be searched for.>
  <Change the top and bottom pointers of your list to define a smaller list containing the target>
UNTIL
  <Target is found>
OR
  <list is empty>
```

If we expand this algorithm we get:

```
<Declare boolean, beginning, middle and end variables>
RECEIVE searchKey FROM (INTEGER) KEYBOARD
REPEAT
  <If middle item < search key then set the beginning of the sub-list to just after the middle>
  <If middle item > search key then set the end of the sub-list to just before the middle>
  <If middle item = search key then search key has been found>
UNTIL
  <Target is found>
OR
  <list is empty>
```
The complete algorithm would be:

```plaintext
PROCEDURE binarySearch(ARRAY OF INTEGER myArray)

DECLARE found AS BOOLEAN INITIALLY false
DECLARE startPos INITIALLY 0
DECLARE endPos INITIALLY length(myArray)
DECLARE middle AS INTEGER INITIALLY 0
DECLARE searchKey INITIALLY 0

RECEIVE searchKey FROM KEYBOARD
REPEAT
  SET middle TO (startPos + endPos) / 2
  # use integer division in case it is an odd number
  IF array[middle] < searchKey THEN
    SET startPos TO middle + 1
  ELSE
    SET endPos TO middle -1
  END IF
  IF array[middle] = searchKey THEN
    SET found TO true
    SEND "Found at position " & middle TO DISPLAY
  END IF
UNTIL found = true OR ( startPos > endPos )
IF found = false THEN
  SEND "Not found" TO DISPLAY
END IF

END PROCEDURE
```

**Activity: Binary search algorithm - Trace table**

DETERMINE myArray INITIALLY [1,2,3,4,5,6,7,8,9,10]

Set up a trace table for the binary search algorithm and trace the values of the found, startPos, endPos and middle when the values 9 and 11 are searched for:

<table>
<thead>
<tr>
<th>found</th>
<th>Middle</th>
<th>startPos</th>
<th>endPos</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

.................................

**Activity: Binary search algorithm - Programming**

Implement the above algorithm in the programming language of your choice.

Test your program by searching for values:
1. near the start of the list;
2. at the end of the list;
3. in the middle of the list;
4. that are not in the list.

5.3.2 Comparing the binary search with the linear search

The linear search is a simple algorithm whereas the binary search is a more complex one, and also requires the list to be sorted, which requires more processor time. If a list is very small, it may be more efficient to use a linear search than to sort it first and then use a binary search.

The efficiency of a search can be compared by calculating the average number of comparisons required to find the key. In a linear search, on average the number of comparisons required will be half the total number of items in the list \( n \).

In a binary search the maximum number of comparisons required can be calculated by finding the power of 2 which is greater than the total number of items in the list \( n \).

<table>
<thead>
<tr>
<th>List size</th>
<th>Number of comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2 ( (2^2 = 4) )</td>
</tr>
<tr>
<td>7</td>
<td>3 ( (2^3 = 8) )</td>
</tr>
<tr>
<td>15</td>
<td>4 ( (2^4 = 16) )</td>
</tr>
<tr>
<td>28</td>
<td>5 ( (2^5 = 32) )</td>
</tr>
<tr>
<td>60</td>
<td>6 ( (2^6 = 64) )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>5978</td>
<td>16 ( (2^{15} = 65536) )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linear search</th>
<th>Binary search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple algorithm</td>
<td>Complex algorithm</td>
</tr>
<tr>
<td>Data can be unsorted</td>
<td>Data must be sorted</td>
</tr>
<tr>
<td>Number of comparisons required on average is ( n /2 )</td>
<td>Number of comparisons required on average is ( x ) where ( 2^x &gt; n )</td>
</tr>
<tr>
<td>Slow for large lists</td>
<td>Fast for large lists</td>
</tr>
</tbody>
</table>
Activity: Comparing the binary search and linear search

Binary search:

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Mid Value  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>57</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Mid Value  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Mid Value  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>82</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Mid Value  Comparisons

Found!
Linear search:

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Comparisons

<table>
<thead>
<tr>
<th>16</th>
<th>29</th>
<th>34</th>
<th>48</th>
<th>57</th>
<th>59</th>
<th>72</th>
<th>82</th>
<th>90</th>
<th>91</th>
</tr>
</thead>
<tbody>
<tr>
<td>82</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search Key  Comparisons

Found!
5.4 Sorting algorithms

Learning objective

By the end of this topic you should be able to implement the following sorting algorithms:

- selection sort using two lists;
- bubble sort;
- insertion sort.

Sorting information held by any organisation or business is crucial if they want to be able to easily retrieve any of that information again. Traditionally, physical stores of information like libraries, telephone directories, dictionaries, book indexes, transport timetables etc. are all organised sequentially to make them easy to search through. Exactly the same applies to information stored on computer. As we have seen, the binary search algorithm is a very efficient means of retrieving information from a list, but the list must be sorted first.

Finding efficient sorting algorithms have been an area of research in computing science from the early days of the subject. The Wikipedia page on sorting lists over forty different algorithms. The efficiency of a sorting algorithm will depend on a number of factors:

- how many comparisons are required on average per number of items to be sorted;
- how many times items in the list have their positions swapped;
- whether they have different results depending on whether they are dealing with a partially sorted list or a random one;
- whether they require any additional memory to perform the sort.

5.4.1 Selection sort using two lists

This is a very simple algorithm for sorting items which uses two lists.

For each item from listA do
  <find the minimum value in list A and place it in position in list B>
  <replace the item removed from list A with a dummy value>
End for each

Selection sort using two lists

Selection sort using two lists:

The following shows two lists of numbers being sorted from the lowest number to the greatest number using selection sort using two lists (List A and List B).

The algorithm goes through List A comparing pairs of adjacent numbers until it reaches the end of the list and records the index position which has the lowest value.

The numbers highlighted in red are being compared:
Once the algorithm reaches the end of the list, the index position is 1 as that contained the smallest value (1) - this is moved into List B and a dummy value (D) is used to mark the position in List A.

It then starts again at the beginning of List A and repeats for Pass 2:

**Pass 2**

<table>
<thead>
<tr>
<th>Index Position</th>
<th>Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4</td>
<td></td>
</tr>
<tr>
<td>List A</td>
<td></td>
</tr>
<tr>
<td>6 D 3 2 8</td>
<td>5</td>
</tr>
<tr>
<td>6 D 3 2 8</td>
<td>6</td>
</tr>
<tr>
<td>6 D 3 2 8</td>
<td>7</td>
</tr>
<tr>
<td>6 D 3 D 8</td>
<td>8</td>
</tr>
</tbody>
</table>

Once the algorithm reaches the end of the list, the index position is 3 as that contained the smallest value (2) - this is moved into List B and a dummy value (D) is used to mark the position in List A.

It then starts again at the beginning of List A and repeats for Pass 3:
Once the algorithm reaches the end of the list, the index position is 2 as that contained the smallest value (3) - this is moved into List B and a dummy value (D) is used to mark the position in List A.

It then starts again at the beginning of List A and repeats for Pass 4:

**Pass 4**

<table>
<thead>
<tr>
<th>List A</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>List B</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Once the algorithm reaches the end of the list, the index position is 0 as that contained the smallest value (6) - this is moved into List B and a dummy value (D) is used to mark the position in List A.

It then starts again at the beginning of List A and repeats for Pass 5:
Once the algorithm reaches the end of the list, the index position is 4 as that contained the smallest value (8) - this is moved into List B and a dummy value (D) is used to mark the position in List A.

This algorithm described in this way is easy to understand, because it is similar to how we might sort a small set of items in real life, such as a deck of playing cards, or books from a box into shelves (but without the need for a dummy value). It is inefficient because it has to check every item in listA each time the loop executes, and because it requires a second list with the same storage space as the first. The extra storage space needed can become a serious problem if very large lists of items are being sorted.

One complication is how to choose the dummy value. If the lists are numeric arrays, then a number must be used, which will need an additional test when selecting which item to move from list A to list B. A text value can only be used as the dummy value if the programming language allows a data type for arrays which can be either a string or a numeric value.

The actual selection sort list uses a nested loop. The outer loop fills the destination list B with values from the source list A. The inner loop uses the outer loop counter to keep track of the position of the minimum value as the list is progressively checked.
PROCEDURE selectionSort()

DECLARE dummy AS STRING INITIALLY "x"
DECLARE minimumPosition INITIALLY 0

FOR listBcounter FROM 0 TO listLength DO
    SET minimumPosition TO listBcounter
    FOR listAcounter FROM 0 TO listLength DO
            SET minimumPosition TO listAcounter
        END IF
    END FOR
    SET listB[listBcounter] TO listA[minimumPosition]
    SET listA[minimumPosition] TO dummy
END FOR

END PROCEDURE

Activity: Selection sort

Implement the selection sort algorithm in your chosen programming language.
If we wanted to count the number of comparisons and the number of times the selection sort executes a loop, then we could amend the algorithm like this:

```
PROCEDURE selectionSort()

DECLARE dummy INITIALLY "x"
DECLARE comparisons INITIALLY 0
DECLARE passes INITIALLY 0
DECLARE minimumPosition INITIALLY 0

FOR listBcounter FROM 0 TO listLength DO
    SET passes TO passes + 1
    SET minimumPosition TO listBcounter
    FOR listAcounter FROM 0 TO listLength DO
        SET comparisons TO comparisons + 1
            SET minimumPosition TO listAcounter
        END IF
    END FOR
    SET listB[listBcounter] TO listA[minimumPosition]
    SET listA[minimumPosition] TO dummy
END FOR
SEND "Comparisons: " & comparisons TO DISPLAY
SEND "Passes: " & passes TO DISPLAY

END PROCEDURE
```

Because it uses fixed loops, the selection sort using two lists will always have to make \( n^2 - n \) comparisons and will make \( n \) passes through the list and will move \( n \) items from one array to another where \( n \) is the number of items in the list to be sorted.

Because it uses two arrays, the selection sort has serious memory overhead problems when sorting large lists.

In practice, because of its inefficiencies, the selection sort is not an algorithm that would be implemented in a real application. It is given here as an introduction to sorting algorithms.

### 5.4.2 Bubble sort

The bubble-sort is so named, because items in the list appear to "bubble" to the top of the list. The bubble sort swaps items between positions within the list, so does not require additional memory to operate. If a variable is introduced to keep track of how many swaps have occurred in a pass through the list, the bubble-sort also has the advantage of being able to stop when the list has been sorted, making it much faster when sorting a partially sorted list.

In its simplest form, the bubble-sort compares each item in the list in turn with the next item in the list. If the first is larger than the second then it will swap them. It will continue until it reaches the last item in the list which will mean that the largest item in the list has "bubbled" up to the last position. The process is now repeated, but this time stopping at
the item in the second last place in the list. This continues until it reaches the beginning
of the list.

**Bubble sort**

**First pass:**
The following array needs to be sorted using bubble sort. The numbers highlighted in
red are being compared.

6 and 1 are compared, and swapped:

![Array with 6 and 1 swapped]

6 and 3 are compared, and swapped:

![Array with 6 and 3 swapped]

6 and 2 are compared, and swapped:

![Array with 6 and 2 swapped]

6 and 8 are compared, and not swapped:

![Array without swap]
**Second pass:**
The algorithm goes to the beginning of the array after the first pass.

1 and 3 are compared, and not swapped:

```
1 3 2 6 8
```

3 and 2 are compared, and swapped:

```
1 3 2 6 8
```

3 and 6 are compared, and not swapped:

```
1 2 3 6 8
```

**Third pass:**
Although the array is now sorted, the algorithm does not know if it is complete until it does one whole pass without any swaps.

1 and 2 are compared, and not swapped:

```
1 2 3 6 8
```

2 and 3 are compared, and not swapped:

```
1 2 3 6 8
```

..........................................

© HERIOT-WATT UNIVERSITY
This algorithm uses two FOR loops, one inside the other. The outer loop counts down from a value 2 less than the size of the list. This is because the maximum number of passes is one less than the length of the list, and the list starts at index 0.

The inner loop counts up from 0 to the value of the outer loop counter, so that each time the outer loop completes, the inner loop only has to process a smaller section of the list because the rest has already been sorted.

Both of the loops in this algorithm are unconditional loops, so the bubble-sort will always complete $n$ passes through the list, and make $n^2$ comparisons where $n$ is the number of items in the list. In the worst case scenario, where the list to be sorted is in reverse order, there will be $n^2$ swaps as well. Since a swap only occurs after any comparison if the first item is larger than the second item, the number of swaps depends on how partially sorted the original list is. This algorithm as it stands does not check to see whether there have been any swaps, so a partially sorted list will take the same number of passes as a sorted one.

**Activity: Bubble sort**

Implement the bubble-sort algorithm in your chosen programming language.

The bubble-sort algorithm can be made more efficient by introducing a Boolean variable which keeps track of whether there have been any items swapped in a pass. If there have been no swaps, then the outer loop can be terminated. In this case, the outer loop needs to be a conditional one. The inner loop remains a fixed loop with a variable to count how many swaps there have been in that pass.
Activity: Bubble sort 2
Amend the bubble-sort algorithm to make it more efficient using this suggestion.

..........................................

Activity: Bubble sort 3
Amend the algorithm to keep track of the number of passes, swaps and comparisons. Implement it in your chosen programming language and test your algorithm with a selection of lists - unsorted, partially sorted and sorted.

..........................................

5.4.3 Insertion sort
The insertion sort is similar to how we would sort a small set of items such as a hand of cards. We have the advantage of being able to see them all at once, however the computer can only see the value of two items at a time.

The insertion sort algorithm starts at one end of the list and progressively sorts each subsequent item into the list until it reaches the last item.
**Insertion sort**
The following array needs to be sorted using insertion sort. The numbers highlighted in red are being compared:

1) 6 is compared with 8 and swapped:

```
  8  6  1  3  2
```

2) 1 is compared with 8 and swapped:

```
  6  8  1  3  2
```

... and then with 6 and swapped:

```
  6  1  8  3  2
```

3) 3 is compared with 8 and swapped:

```
  1  6  8  3  2
```

... and then with 6 and swapped; it is also compared with 1 but is not swapped:

```
  1  6  3  8  2
```
4) 2 is compared with 8 and swapped:

1 3 6 8 2

... and then with 6 and swapped:

1 3 6 2 8

... and then with 3 and swapped:

1 3 2 6 8

... and then with 1 but not swapped:

1 2 3 6 8

5) All the numbers in the array are now sorted:

1 2 3 6 8

As in the bubble-sort, a fixed outer loop iterates through the list, but this time from 1 to the list size, instead of counting down from the list size to 0.
This algorithm also uses two loops, one inside the other. This time the outer loop counts up from 1 to the length of the list, and the inner loop counts down from one less than the outer loop value down to 1 reducing the number of items to be checked each time round. Note that it checks the list[innerloop] -1 against the next item in the list (since the array index starts at zero)

Activity: Insertion sort
Implement the insertion sort algorithm in your chosen programming language.

..........................................

Activity: Insertion sort 2
Amend the algorithm to keep track of the number of passes, swaps and comparison. Implement it in your chosen programming language and test your algorithm with a selection of lists - unsorted, partially sorted and sorted.

..........................................

The number of swaps will vary according to the values in the list, but the number of passes will always be n-1 where n is the number of items in the list. This is because the outer loop goes from 1 to the number of elements (missing 0) so will always number of elements -1.

Comparing the efficiency of sorting algorithms
The number of swaps and passes required in a sorting algorithm are important because they are a factor in determining the efficiency of the sort in terms of memory requirements and processor time. Every time a swap is made, two pieces of data in an
array are moved between memory locations. In addition a temporary memory location is used in the process. Every pass requires the code inside the loop to be executed by the processor.

5.5 Other sorting algorithms

The algorithms we have been looking at are simple sorting algorithms, but are inefficient where large lists are concerned. Other more efficient algorithms are the quicksort and the merge sort.

5.5.1 Quicksort

The quicksort is a recursive algorithm. You will remember from your work in Higher Computing that recursion is a common technique in more complex programming algorithms. Recursion is where a sub program repeatedly calls itself until a simple base fact is identified.

The quicksort algorithm relies on the base fact that an empty list or one which has only 1 element is by definition, a sorted list. For any other list, it compares every item in it to a pivot chosen from the middle of the list and divides it into two sub-lists - one containing items bigger than the pivot and one containing items smaller than the pivot, then quicksorts the two sub-lists. Each sub-list will then be progressively quicksorted until each is reduced to a set of sorted lists each containing 1 item. This type of algorithm is often referred to as a "divide and rule" solution.

This quicksort algorithm is for a list whose index position ranges from $i$ to $\text{listLength}$:

```
PROCEDURE quickSort(ARRAY OF INTEGER list, INTEGER i, INTEGER listLength)

DECLARE pivotPosition AS INTEGER INITIALLY 0
IF i < listLength THEN
   SET pivotPosition TO partition(list, i, listLength)
   quickSort(list, i, pivotPosition - 1)
   quickSort(list, pivotPosition + 1, listLength)
END IF

END PROCEDURE

FUNCTION partition(ARRAY list, INTEGER i, INTEGER listLength)RETURNS INTEGER
   <swaps items in list to arrange them lower than and higher than the value i>
END FUNCTION
```

The partition function has not been exemplified here, but it takes three parameters and swaps items in the list passed to it so that they are separated into ones lower than and higher than its chosen pivot value. It returns the index value of the pivot.

The algorithm was originally designed to sort words into alphabetical order so that they could be compared to a foreign language dictionary for translation. For this reason it is not very efficient when sorting lists which contain lots of identical items, as that would
be an unlikely scenario in these circumstances.

Because it is a recursive algorithm, it is particularly suited to parallel processing, but is very memory intensive and becomes inefficient if the list to be sorted does not fit into available RAM.

5.5.2 Merge sort

The merge sort is another recursive sort algorithm, also suited to parallel processing. It divides the list into sub lists which are sorted and then merged together to make a larger sorted list. It is particularly useful if you already have two sorted lists which have to be combined together. The merge sort is also a “divide and rule” algorithm.

5.5.3 Comparing sorting algorithms

As well as evaluating sorting algorithms according to their memory requirements and processor load, they may have a different performance depending upon:

- the size of the list being sorted;
- if the list is partially sorted or not;
- if the list to be sorted is already sorted in reverse order;
- if the list contains large numbers of duplicate items.

Many sort algorithm evaluations will give best and worst case scenarios depending on these conditions.

Activity: Other sorting algorithms

Read about other common sorting algorithms and their comparison.

This Wikipedia page describes a large number of sorting algorithms:

This Youtube video shows animations for 15 common sorting algorithms:
https://www.youtube.com/watch?v=kPRA0W1kECg

Please be aware that this video contains flickering images so may cause seizures in some individuals.
5.6 Conclusion

Summary

- The linear search will find an item in an unsorted list.
- The binary search is an efficient search algorithm, but requires a sorted list.
- The binary search algorithm progressively divides a list into two sub lists until the search item is found.
- The efficiency of sorting algorithms depends upon how many comparisons and swaps they perform, and whether they require additional memory.
- The selection sort algorithm takes a list and progressively moves each item into a new sorted list.
- The insertion sort and bubble sort both swap items within a list to complete the sort.
- The quicksort and merge sort are efficient sorts which use a recursive algorithm.
- Sorting algorithms may have different performances depending on the size and type of list they are sorting.
5.7 End of topic test

End of topic test

Q6: Compare a linear search with a binary search, outlining three main differences.

Q7: In a list of 1000 items, the average search length to locate an item using a linear search would be?
   a) 250
   b) 500
   c) 750
   d) 1000

Q8: In a list of 1000 items, the maximum search length to locate an item using a binary search would be?
   a) 10
   b) 50
   c) 100
   d) 250

Q9: Which sorting algorithm uses most memory by creating a second sorted list?
   a) Bubble sort
   b) Selection sort
   c) Insertion sort
   d) Quicksort

Q10: When sorting a list of 100 items, how many comparisons would the selection sort make?
    a) 250
    b) 900
    c) 1000
    d) 9900

Q11: When sorting a list of 10 items, how many passes would an insertion sort make?
    a) 100
    b) 50
    c) 11
    d) 110
Q12: Which of these sort algorithms are most suited to a parallel processing solution?

a) Insertion sort  
b) Bubble sort  
c) Quicksort  
d) Simple sort

Q13: How many comparisons would be made by a binary search to find the value 29 in the following list assuming that the final comparison is the one which matches the search item?

4, 26, 29, 35, 59, 70, 103

a) 3  
b) 2  
c) 4  
d) 1

Q14:
Which sorting algorithm is being used here?

```
PROCEDURE sort(ARRAY list, INTEGER listLength)

   FOR outerLoop FROM listLength -2 T0 0 STEP -1
   FOR counter FROM 0 T0 outerloop DO
      IF list[counter] < list[counter+1] THEN
         Swap (list[counter], list[counter+1])
      END IF
   END FOR
END FOR

END PROCEDURE
```

```
PROCEDURE swap(REF INTEGER a, REF INTEGER b)

   DECLARE temp AS INTEGER INITIALLY a
   SET a TO b
   SET b TO temp

END PROCEDURE
```

a) Selection sort  
b) Quicksort  
c) Bubble sort  
d) Insertion sort
Topic 6

Computational constructs

Contents

6.1 Recursion ................................................................. 108
   6.1.1 Comparing Imperative and declarative programming paradigms .... 108
   6.1.2 Binary search using recursion ..................................... 111
6.2 File Handling ............................................................ 111
   6.2.1 Sequential files ................................................... 111
   6.2.2 Random files ...................................................... 114
6.3 Conclusion ............................................................... 114
6.4 End of topic test ....................................................... 115

Prerequisite knowledge

From your studies at Higher you should already know how:

- your chosen programming language handles sequential files.

Learning objectives

By the end of this topic you will be able to:

- understand the use of recursion as a programming technique;
- read and write data to and from existing files.
6.1 Recursion

Learning objective

By the end of this section you will be able to:

• understand the use of recursion as a programming technique.

Declarative programming is a style of programming which focuses on defining what a program should achieve rather than defining the method of achieving the desired output.

Imperative programs define exactly the steps that a computer will execute to achieve a specific result. Programs written in a declarative style make use of declarations which define what the output must be to be correct. This contrasts with the imperative approach where the programmer specifies a set of steps required to solve a problem.

Examples of declarative languages are:

SQL which is considered a more declarative language than an imperative because it defines what the results of a query must contain to be valid but does not provide a specific definition of what the output will be.

Prolog, is a logic programming language which is considered to be a declarative language because programs written in Prolog consist of a number of simple statements (facts) and conditional statements (rules). These facts and rules together make up the knowledge base. When a program is run, the compiler (often referred to as the inference engine) will match the inputs to the program with these facts and rules to work out their logical consequence.

Recursion is a powerful programming technique where a sub-program calls itself repeatedly until the problem it is trying to solve reaches a solution, is used in both declarative and imperative programming styles.

6.1.1 Comparing Imperative and declarative programming paradigms

Take the example of the mathematical operation known as factorial used in probability calculations. The mathematical symbol for factorial is the exclamation mark.

The mathematical definition of the factorial of a number is:

The factorial of a positive integer n, denoted by n! is the product of all positive integers less than or equal to n.

n! = (1 * 2 * 3 * 4 * ... * n)

1! = 1 * 1 = 1
2! = 1 * 2 = 2
3! = 1 * 2 * 3 = 6
4! = 1 * 2 * 3 * 4 = 24
5! = 1 * 2 * 3 * 4 * 5 = 120

The imperative approach to working out the factorial of a number would be to start with 1, and using a loop, repeatedly multiply it by successive values.
A declarative solution which uses recursion would be to define the factorial of any number as that number multiplied by the factorial of that number minus 1.

\[ n! = n \times (n-1)! \]

\[
\text{PROCEDURE getFactorial(INTEGER inputValue)} \\
\text{DECLARE counter INITIAL 0} \\
\text{DECLARE factorial INITIAL 1} \\
\text{REPEAT} \\
\text{SET counter TO counter + 1} \\
\text{SET factorial TO factorial \times counter} \\
\text{UNTIL counter = inputValue} \\
\text{SEND "The factorial of " \& inputValue \& " is " \& factorial \& " TO DISPLAY} \\
\text{END PROCEDURE}
\]

In the declarative programming example, we declare the fact that the factorial of 1 is 1. And declare the rule that the factorial of any other number \( n \) is equal to \( n \) multiplied by the factorial of \( n-1 \).

**Activity: Recursive function - Factorial**

Use the pseudocode above to create a recursive function in your chosen programming language to calculate the factorial of a number.

\[
\text{PROCEDURE getFactorial(INTEGER inputValue)} \\
\text{SEND "The factorial of" \& inputValue \& " is " \& factorial (inputValue) \& " TO DISPLAY} \\
\text{END PROCEDURE}
\]

**Activity: Recursive function - Sum of numbers**

Write the a recursive function in your chosen programming language to take a number as input and return the sum of all numbers between it and 0, including the number itself. You need to declare one fact, which is that if the number input is 1, then the sum of all numbers between it and 0 is 1.

The rule is that if any other number is input, then the sum of all numbers between it and 0 including the number itself, is that number plus the sum of all numbers between than number minus 1, and 0.
(n + the sum of all numbers between n-1 and 0 including n-1)

FUNCTION sumNums(INTEGER number)
RETURNS INTEGER
<the sumNums of 1 is 1>
<the sumNums of any other number is the sumNums of one less
  than that number + that number>
END FUNCTION

From the Higher course you might remember one of the methods of converting a positive
decimal value to binary by repeatedly dividing by two and writing down the remainder. The
remainders are then read upwards from the last one calculated.

```
<table>
<thead>
<tr>
<th></th>
<th>Convert 113 to binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td>2</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

This is a prime candidate for a recursive function which returns a binary string. The
base fact is that the binary value of decimal 1 is "1". The rule is that binary string of any
other number is the binary string of that number divided by two, concatenated with the
remainder when that number is divided by two.

**Activity: Algorithm for a string function**

Write the algorithm for a string function which takes an integer as a parameter and
returns a string which is its binary value. Implement it in your chosen programming
language.

FUNCTION binaryString(INTEGER number) RETURNS STRING
6.1.2 Binary search using recursion

Since a binary search is effectively applying the same process repeatedly on an ordered list until either the item is found or the section of the list being searched is empty, it lends itself to a recursive solution.

```
PROCEDURE binarySearch(ARRAY OF INTEGER myArray, INTEGER searchKey)
  IF <Target is found or array is empty>
    THEN
      <display result>
      ELSE
        SET newList TO split(myArray,searchKey)
        binarySearch(newList,searchKey)
    END IF
  END PROCEDURE

FUNCTION split(myArray, searchKey) RETURNS ARRAY OF INTEGER
  <Find the middle item in the myArray and compare it with the searchkey>
  <Change the top and bottom pointers the array to contain the target>
    creating newArray>
  RETURN newArray
END FUNCTION
```

6.2 File Handling

Learning objective

By the end of this section you will be able to:

• read and write data to and from existing files.

6.2.1 Sequential files

You will already know from your work in Higher Computing, that just as a computer program can receive data from an input device such as a keyboard and send data to an output device such as the display, it can also receive data from and send data to a sequential file.

A sequential file is identified by a path and filename. As the name suggests, sequential files are read starting from the from beginning, so a file cannot be read and written to simultaneously, although several different sequential files can be open for reading or writing at the same time.

If a file to be written to does not already exist, then the file must be created first.

Inserting, amending and deleting the content of a sequential file would normally require it to be read into memory, edited and then saved, but many programming languages include an APPEND option for sequential files allowing data to be added to the file without reading it into memory first.
The following three examples use an array of 10 strings which hold names.

Writing a set of string values to a new sequential text file from an array:

PROCEDURE writeFile(ARRAY OF STRING names)
    DECLARE myfile INITIALLY " N:\myfile.txt"
    CREATE myfile
    OPEN myfile
    FOR counter FROM 0 TO 9 DO
        SEND names[counter] TO myfile
    END FOR
    CLOSE myfile
END PROCEDURE

Reading a set of string values from a sequential text file into an array:

PROCEDURE readFile(ARRAY OF STRING names)
    DECLARE myfile INITIALLY " N:\myfile.txt"
    OPEN myfile
    FOR counter FROM 0 TO 9 DO
        RECEIVE names[counter] FROM myfile
    END FOR
    CLOSE myfile
END PROCEDURE

Not all programming languages will include an APPEND command. If they do, then this would be the syntax for appending a single string value to the sequential file:

PROCEDURE appendFile()
    SEND "Please enter filename to append" TO DISPLAY
    DECLARE newName INITIALLY ""
    RECEIVE newName FROM KEYBOARD
    DECLARE myfile INITIALLY " N:\myfile.txt"
    OPEN myfile
    APPEND newName TO myfile
    CLOSE myfile
END PROCEDURE
This example uses an array of 10 integers.

Reading a set of numeric values from a sequential text file into an array:

```plaintext
PROCEDURE readFile(ARRAY OF INTEGER values)
DECLARE myfile INITIALLY "N:\myfile.txt"
DECLARE numberString INITIALLY ""
OPEN myfile
FOR counter FROM 0 TO 9 DO
  RECEIVE numberString FROM myfile
  SET names[counter] TO asciiToIntger(numberString)
END FOR
CLOSE myfile
END PROCEDURE

FUNCTION asciiToIntger(STRING characters) RETURNS INTEGER
DECLARE value AS INTEGER INITIALLY 0
<convert ASCII character(s) to integer value>
RETURN value
END FUNCTION
```

Activity: Sequential files
Implement the algorithm above using your chosen programming language to:

- Open an existing file
- Write the contents of an array to the file
- Close the file
- Open an existing file
- Read the contents of the file into an array
- Close the file.

..........................................

Activity: Sequential files 2
Using your chosen programming language, write a program which will read a list of integers from a sequential text file, sort them, and then write them to a second sequential text file.

..........................................

Activity: 2-D Arrays
Adapt the maze program you wrote in the Data types and structures Topic (Section 2.2) to read in the room data from a text file.

..........................................

© HERIOT-WATT UNIVERSITY
6.2.2 Random files

A random file stores data in a way which allows it to be retrieved from any place in the file without having to read through the entire file. A parallel would be the difference between sequential and random access to data on a backing storage device. Data stored in a random file needs to be indexed so that each item can be uniquely identified. If fixed length data items are used then the position of an item in a file can easily be calculated. Random files are frequently used to store records.

The advantages of random files are:

- Fast access to data since the position of any item can be calculated from its index.
- Data items can be accessed individually instead of having to read the entire file.
- Random files can be read and written to simultaneously.

6.3 Conclusion

Summary

- Recursion is a programming technique where a function or sub-procedure calls itself repeatedly until a terminating case is reached.
- Identifying the terminating case in any operation is the most important first step in devising a recursive sub-program.
- Recursion is a common programming construct used in declarative programming.
- Most programming languages will have commands for reading from and writing to sequential files.
- In order to read from or write to a sequential file it must first be identified by filename and path.
- Random files store indexed data items which can be read individually without having to read the entire file into memory.
- Unlike sequential files, random files can be read and written to simultaneously.
6.4 End of topic test

End of topic test

Q1: Which of these statements are true of recursion?
   a) Recursion is a programming technique used only in declarative programming languages.
   b) Recursion is where a sub-program repeatedly calls itself.
   c) Recursion is not used in Object Oriented programming.
   d) Only functions can be recursive.

Q2: Why is it necessary to identify a terminating base case when designing a recursive algorithm?
   a) If the base case is missing it will not be a recursive algorithm.
   b) The program will continually call itself until running out of memory.
   c) The program will not be syntactically correct.
   d) Programs using recursion cannot run if a base case has not been specified.

Q3: Which of these statements are true of sequential files?
   a) They cannot be read and written to simultaneously true.
   b) Reading a sequential file always starts at the beginning true.
   c) They cannot be used to store real numbers.
   d) They must be saved in the same folder as the program reading them.

Q4: Which type of file would be best to use for the following tasks? Choose from Sequential / Random:
   a) Storing the readings from a temperature gauge at 10 minute intervals sequential
   b) Storing an array of records containing names addresses and telephone numbers random

Q5: What is the advantage of data items in a random file having a fixed length?
   a) The position of an item can be calculated from its index.
   b) Individual data items can be edited simultaneously.
   c) Field names do not need to be included in the file.
   d) The file size will be smaller than a sequential file.
Topic 7

Testing and documenting solutions

Contents

7.1 Revision ................................................................. 118
7.2 Introduction ............................................................ 119
7.3 Component testing .................................................... 119
7.4 Integration testing ..................................................... 120
7.5 Acceptance (beta) testing ........................................... 121
7.6 Usability testing ......................................................... 121
7.7 Accessibility testing ................................................... 122
7.8 Summary ................................................................. 123
7.9 End of topic test ......................................................... 124

Prerequisite knowledge

From your studies at Higher you should already know how to:

• construct a test plan;
• describe comprehensive testing;
• describe systematic testing explain the difference between syntax, execution and logic errors;
• understand how dry runs, trace tables, trace tools and breakpoints are used in the debugging process.

Learning objectives

By the end of this topic you should be able to describe the following types of testing:

• component testing;
• integration testing;
• acceptance (Beta) testing;
• usability testing;
• accessibility testing.
# 7.1 Revision

## Quiz: Revision

### Q1:
Fill in the missing values in this trace table:

```assembly
DECLARE numbers INITIALLY [17, 15, 4, 7, 8]

PROCEDURE findMinimum(ARRAY OF INTEGER numbers)

DECLARE minimumValue INITIALLY numbers[0]
FOR counter FROM 1 TO 5 DO
    IF minimumValue > numbers[counter] THEN
        SET minimumValue TO numbers[counter]
    END IF
END FOR
SEND ["The smallest value was 
\& minimumValue"] TO DISPLAY

END PROCEDURE
```

<table>
<thead>
<tr>
<th>minimumValue</th>
<th>counter</th>
<th>numbers[counter]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

### Q2:
A computer program is designed to accept input values between 0 and 99 as whole numbers. If the value 99 was entered this would be an example of:

a) invalid data.

b) exceptional data.

c) extreme data.

d) normal data.

### Q3:
Alpha testing is typically carried out by end users of a program.

a) True

b) False

### Q4:
Beta testing is typically carried out by end users of a program.

a) True

b) False
7.2 Introduction

Testing and the documentation which accompanies it is crucial to the success of any software project. Testing cannot prove that there are no errors, but if testing is done in a systematic way, and if the software is tested at every stage in the development process, then developers and their clients can be confident that everything possible has been done to eliminate the possibility of error. This section will look at the testing which is done at all software development stages from initial coding to final delivery and evaluation. The number of different stages at which testing is done will depend upon the complexity of the project, but there will always be a need to test regularly during the development of a project, not just on its completion.

Every testing stage should be accompanied by documentation outlining the tests done including test data, results and any errors found with details of how they were corrected. The documentation will be invaluable to those testing subsequent stages and will also aid anyone who needs to re-visit a previous stage in the light of subsequent previously undetected problems.

7.3 Component testing

Component testing is where the modules which make up the program are tested independently of each other. This testing can take place during the development process and does not need every part of the project to have been completed before the testing takes place. It makes sense to test individual components of a project while they are being written rather than waiting until they are combined together making errors difficult to spot if there are several which may interact with each other to give unpredictable results.

When testing partially completed code, programmers often make use of stubs and drivers.

A stub is a dummy function or procedure which can be used in place of one which has yet to be written, and they are used when there needs to be some code present to indicate when that sub program will be called and if that call has been successful. A stub may just be a single line of code which stands in for the missing sub-program but which ensures that the rest of the code will still function.

A driver is a piece of code designed to test a function or procedure which has been written, but when the rest of the program has not been completed. A driver would call the sub-program with the appropriate parameters to test that it functions correctly and delivers the correct outputs.

A stub which could be used when a sorting procedure is required in a program but has yet to be written:

```
PROCEDURE sort(ARRAY OF INTEGER numbers)
SEND "sort procedure called. This procedure will sort a numeric array passed as a parameter into ascending order. " TO DISPLAY
END PROCEDURE
```
A driver which could be used to test the sort procedure before the rest of the program has been completed:

```plaintext
PROCEDURE testSort()
    DECLARE numbers INITIALLY [4,78,2,67,3,156,5,987,3,23]
    sort(numbers)
    FOR counter FROM 0 TO 9 DO
        SEND numbers [counter] TO DISPLAY
    END FOR
END PROCEDURE
```

As with all testing, component testing should be documented with test data and results, so that effort is not duplicated if any module has to be re-visited in the light of problems encountered at a later stage.

### 7.4 Integration testing

Once individual modules in a software project have been built and tested, the next stage is to make sure that they all work together. The flow of data between modules needs to be checked for data integrity, and it needs to be checked that the modules work together as planned. Usually a test plan will be written, the inputs to the test being the modules which were tested at the component testing stage. There are a number of different approaches to integration testing depending on the complexity of the project. Small groups of modules which should work together may be tested as a unit, then combined with other units. Alternatively if the project is a relatively small one, all the modules may be combined and the system tested as a whole.

An integration test plan will identify:

- the original software specification;
- any test documentation from the component testing stage;
- a list of test data to be used;
- a list of conditions which must be in place before the setting starts;
- details of what constitutes a successful test.

The advantage of integration testing is that it tests how modules work together, and tests features which would be impossible to test at individual module level.

Both component and integration testing are known as **alpha testing** which is done in-house by members of the development team, or by a testing group appointed by the developers.
7.5 Acceptance (beta) testing

Acceptance testing is testing which is done by the client in the case of bespoke software systems, or by potential customers in the case of commercial software. As its name suggests, the end result of acceptance testing is that the client accepts delivery of the software, and checks that it is fit for purpose. Acceptance testing is sometimes called beta testing to distinguish it from alpha testing. Alpha testing is important, but because it is done by programmers who may be associated with the project; their tests can suffer from unintentional bias or just the inability to imagine the misunderstanding which a novice user may experience. Some software companies release Alpha versions of their software to the public to get feedback relatively early in the development of the software. When this happens the company make clear that the software is an early release and will contain errors. Beta testing, because it is done by people who will actually be using the software, can spot problems and bugs which have previously gone unnoticed.

Commercial software companies may release beta versions of their software to selected users such as computer journalists or current users of previous versions of the software. These individuals receive the software free or at reduced cost, and earlier than ordinary members of the public in return for recording any bugs they might find. The company gains valuable information on bugs which they can fix before the final release.

7.6 Usability testing

Again as you might guess form the name, this type of testing is designed to ensure that the software is as easy to understand and use as possible, and that the interface is suitable for everyone who might be using it. The following criteria can be used to test and evaluate a user interface to a piece of software:

• **Appropriate**: The interface should be designed with the type of user and the tasks they will be performing in mind. Ideally it should mirror the real world as much as possible, so that it uses language and concepts familiar to the user. Exactly how this is done can be a matter of fashion as well as functionality. The term "skeuomorphism" refers to the way some interfaces try to mimic their physical counterparts. A radio app for instance, may have controls which look like tuning knobs and switches even though they do not aid and in some case may even hinder its functionality.

• **Customisable**: The interface must be able to be changed to suit the needs of the user, without compromising its functionality. A customisable interface provides shortcut keys for experienced users, with menus and dialogue boxes for novice users. Users should be able to add frequently used tools to a menu or toolbar and change colours or contrast if required. If the software is to be sold and used in different countries, regional versions should be available in local languages and character sets.

• **Accessible**: The interface must be customisable to the extent that users who need a screen reader, require high contrast screen displays or who have other eyesight problems can still operate it comfortably.
• **Consistent**: The interface should provide menus and dialogue boxes so that commands and other operations are grouped together in a logical way. For example, the contents of the file and edit menus in many applications will all contain many of the same familiar options.

• **Controllable**: The interface should support the user so that they always have the option of undoing crucial operations. There should always be an "escape" from a sequence of operations.

• **Helpful**: The interface should provide help on request, and it should be available in the context of the task being undertaken. Documentation should be comprehensive and always be available if required.

You can watch a video on the history of user interfaces here:

https://www.youtube.com/watch?v=vALW9fVOXHQ

### 7.7 Accessibility testing

Accessibility testing is done to see how well a piece of software or an information system caters for users with disabilities. Disabilities vary from individual to individual so most testing for accessibility is done to check that the system meets a recognized accessibility standard. The World Wide Web Consortium (W3C) has created a set of Web Content Accessibility Guidelines to help web designers make their sites accessible.

Conditions which require consideration include:

- Impaired motor control
- Blindness/Impaired vision
- Colour blindness
- Deafness/Impaired hearing
- Epilepsy
- Dyslexia.

Users who suffer from impaired motor control often find a mouse or touchscreen difficult to use. A keyboard alternative should be available for all operations.

Users who are blind or suffer from impaired vision use **screen readers**. To help, information on pages should be arranged with text near the top so that this is the first thing the reader encounters. Pages with graphics need alternative text descriptions of their images. Links should be labelled. For users with impaired vision, the ability to change the font size or contrast will make it easier for them to use the interface.

When choosing a colour scheme for an interface, combinations of colours which look the same to colour blind users should be avoided.

Users with epilepsy can have an attack triggered by flashing colours or flickering images. These should be avoided.
Users with dyslexia often find particular colour combinations make reading information easier. An interface which allows users to customise colours or apply a 3rd party "skin" helps these users.

Users who are deaf or suffer from impaired hearing will not find audible warnings or notifications helpful. There should always be a visual indication when a user is being informed of an event or being asked for information.

**Activity: W3C Web Content Accessibility Guidelines**

Research the W3C Web Content Accessibility Guidelines on the Internet and use the information to make a list of criteria for evaluating an interface for accessibility.

7.8 **Summary**

- Testing of any sort should be accompanied by documentation.
- Programmers doing component testing will make use of stubs and drivers if the code they are testing is part of an incomplete project.
- Integration testing checks to make sure that individual modules work together as planned.
- Acceptance (beta) testing is done by users who were not involved at the implementation stage and who are therefore less likely to show unintentional bias in favour of the application.
- Usability testing is concerned with the quality and appropriateness of the user interface.
- Accessibility testing is concerned with how well an application and its user interface caters for users with disabilities.
7.9 End of topic test

End of topic test

Q5: Which documentation would be needed if the testing stage needs to be revisited in the light of previously undetected problems?

a) The source code.
b) The test data used.
c) The requirements specification.
d) The test plan and expected results.

Q6: Why should testing always be accompanied by documentation?

Q7: In component testing, what is a stub?

a) A dummy function or procedure.
b) Code designed to test a function or procedure.
c) The result of a component test.
d) Partially completed code.

Q8: In component testing, what is a driver?

a) A dummy function or procedure.
b) Code designed to test a function or procedure.
c) The result of a component test.
d) Partially completed code.

Q9: Which of these groups of people will not be involved in Integration testing?

a) Developer staff
b) Programmers
c) Clients
d) Beta testers

Q10: What type of testing is performed on the finished product prior to acceptance by the client?

a) Alpha testing
b) Beta testing
Q11: Which of these features make software "controllable"?

a) There should always be an option to escape from a crucial operation.

b) The interface should be customisable.

c) Menu items should always be grouped together logically.

d) Help should always be provided on request.

Q12: What accessibility functions might a user with dyslexia or colour blindness find useful in an interface?

Q13: Why should you avoid flickering colours or images in a user interface?
Topic 8

End of Unit 1 test
End of Unit 1 test

**Q1:** Which of these statements are true of modularity?

a) Modularity improves readability.
b) Global variables improve modularity.
c) Modularity improves portability.
d) Modularity is a consequence of top down analysis.

**Q2:** In object oriented programming _______ enables you to hide, inside an object, both the instance variables and the methods that act on them.

a) encapsulation
b) inheritance
c) recursion
d) iteration

**Q3:** In Object Oriented programming _______ is an abstract idea that can be represented with instance variables and methods.

a) a class
b) an object
c) a function
d) a variable

**Q4:** In Object Oriented programming _______ is the process of creating new classes, called sub-classes, from an existing class.

a) encapsulation
b) inheritance
c) recursion
d) iteration

**Q5:** In concurrent programming, a _______ is where two threads access the same data simultaneously resulting in unpredictable results.

a) deadlock
b) race condition
c) priority condition
d) resource starvation
Q6: In concurrent programming, a _____________ where two threads are competing for the same resource resulting in neither completing its operation.

a) deadlock  
b) race condition  
c) priority condition  
d) resource starvation

Q7: Describe the five variables required for the implementation of a **queue** class.

Q8: The following diagram represents a queue:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>42</td>
<td>7</td>
<td>21</td>
<td>94</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Front = 1  
Rear = 4

What would the values of Front and Rear be after the following operations:

Join 9  
Leave  
Join 35

a) Front = 0, Rear = 4  
b) Front = 6, Rear = 3  
c) Front = 6, Rear = 4  
d) Front = 2, Rear = 6

Q9: Which of these software development methodologies is least likely to involve the client with the coding and testing stage of a project?

a) Scrum  
b) Waterfall model  
c) Spiral development  
d) Incremental prototyping

Q10: Which of these statements is the best definition of a prototype?

a) A completed project ready for testing.  
b) A partially completed project.  
c) A working model designed for the purpose of evaluation.  
d) A scale model of a project.
Q11: In a list of 100 items, the average search length to locate an item using a **linear** search algorithm would be?

a) 25  
b) 50  
c) 75  
d) 100  

Q12: In a list of 1000 items, the maximum search length to locate an item using a **binary** search algorithm would be?

a) 10  
b) 50  
c) 100  
d) 250  

Q13: Which sorting algorithm uses most memory by creating a second sorted list?

a) Bubble sort  
b) Selection sort  
c) Insertion sort  
d) Quicksort  

Q14: When sorting a list of 100 items, how many comparisons would the **selection** sort make?

a) 250  
b) 900  
c) 1000  
d) 9900  

Q15: When sorting a list of 10 items, how many passes would an **insertion** sort make?

a) 100  
b) 50  
c) 55  
d) 110  

© HERIOT-WATT UNIVERSITY
Q16:
Which sorting algorithm is being used here?

```
PROCEDURE sort(ARRAY OF INTEGER list, INTEGER listLength)
FOR outerLoop FROM listLength TO 0 STEP -1 DO
    FOR counter FROM 0 TO outerLoop -1 DO
        IF list[counter] < list[counter+ 1] THEN
            swap (list[counter], list[counter+1])
        END IF
    END FOR
END FOR
END PROCEDURE

PROCEDURE swap(REF a, REF b)
    DECLARE temp AS INTEGER INITIALLY a
    SET a TO b
    SET b TO temp
END PROCEDURE
```

a) Selection sort  
b) Quicksort  
c) Bubble sort  
d) Insertion sort

Q17: Which of these statements are true of recursion?

a) Recursion is a programming technique used only in declarative programming languages.  
b) Recursion is where a sub-program repeatedly calls itself.  
c) Recursion is not used in Object Oriented programming.  
d) Only functions can be recursive.

Q18: Which of these statements are true of sequential files?

a) They cannot be read and written to simultaneously.  
b) Reading a sequential file always starts at the beginning.  
c) They cannot be used to store real numbers.  
d) They must be saved in the same folder as the program reading them.

Q19: Which type of file would be best to use for the following tasks? Choose from Sequential / Random:

a) Storing the readings from a humidity gauge at 15 minute intervals.  
b) Storing an array of records containing names and scores.
Q20: In component testing, a dummy function or procedure used to test an incomplete program is a:
   a) stub
   b) driver
   c) sub procedure
   d) code block

Q21: In component testing, a section of code designed to test a function or procedure is a:
   a) stub
   b) driver
   c) sub procedure
   d) code block

Q22: Which of these groups of people will be involved in Integration testing?
   a) Developers
   b) Programmers
   c) Clients
   d) Beta testers

Q23: What type of testing is performed on the finished product prior to acceptance by the client?
   a) Alpha testing
   b) Beta testing

Q24: In this recursive function written in Visual Basic 6, which statement stops the function continuing to call itself forever?

Function sumnums(number)
  If number = 1 Then
    sumnums = 1
  Else: sumnums = sumnums(number - 1) + number
  End If
End Function

   a) If number = 1 Then sumnums = 1
   b) Else: sumnums = sumnums(number - 1) + number
Q25: This function returns the number at position n in the Fibonacci series. (The Fibonacci series is where each number is equal to the sum of the previous two numbers in the series)

```plaintext
LINE1: FUNCTION fibonacci (INTEGER number) RETURNS INTEGER
LINE2: IF number = 1 OR number = 2 THEN
LINE3: RETURN 1
LINE4: ELSE RETURN (fibonacci (number -1) + (fibonacci (number -2))
LINE5: END IF
LINE6: END FUNCTION
```

On which line is the recursive function call?

a) Line 1  
b) Line 2  
c) Line 3  
d) Line 4

Q26: On which line is the base case for the recursion?

a) Line 1  
b) Line 2  
c) Line 3  
d) Line 4

Q27: This code populates a 2D integer array:

```plaintext
PROCEDURE fillArray (ARRAY OF INTEGER myArray) 
FOR row FROM 0 TO 4 DO 
    FOR column FROM 0 TO 4 DO 
        SET myArray[row][column] TO row + column + 1 
    END FOR 
END FOR 
END PROCEDURE 
```

What value would the following command send to the screen?

```plaintext
SEND myArray [2] [2] TO DISPLAY 
```

a) 2  
b) 4  
c) 5  
d) 6
**Q28:** When an interface supports the user so that they always have the option of undoing or escaping from crucial operations, it is said to be:

a) controllable  
b) consistent  
c) customisable  
d) correct

**Q29:** Flickering images and colours in an interface can be dangerous for users who suffer from:

a) blindness  
b) dyslexia  
c) epilepsy  
d) deafness
Glossary

1-D Array
An array with a single index.

2-D Array
An array with two indexes.

Actor
Individuals who interact with a system in a Use Case Diagram or a Sequence Diagram.

Algorithm
a detailed sequence of steps which, when followed, will accomplish a task.

Alpha testing
tests performed by the programming team during the implementation phase.

Beta testing
tests performed by a group of clients or users prior to accepting the software.

Circular queue
A queue which uses a system where only the start and end pointers move, allowing the array to behave as if its spaces were arranged in a circle.

Class
a class in Object Oriented Programming is code which defines the data an object of that type can use (its instance variables) and what it can do (its methods).

Class diagram
a table containing the name of a class together with its instance variables and methods.

Concurrency control
In concurrent computing, ensuring that threads operate in the correct sequence and priority.

Data integrity
when data transfer between components hardware or software components occurs without errors.

Deadlock
where two threads are competing for the same resource resulting in neither completing its operation.

Driver
a piece of code designed to test a function or procedure in an incomplete program.

Elision
natural language segment text within pseudocode describing a process in the algorithm which needs to be expanded.
Encapsulation
a characteristic of objects in object-oriented programming which means that objects are closed systems which cannot be altered from outside.

Extreme programming
a type of Agile software development involving frequent communication with the client, frequent releases and the expectation of several changes in requirements.

Formal Parameter
the parameters in the definition of a procedure or function.

Inference Engine
the logic used by a declarative programming language to match a query with the facts and rules in its knowledge base.

Inheritance
used in object-oriented programming, the sharing of characteristics between a class of objects and a newly created sub-class. This allows code re-use by extending an existing class.

Instance Variables
In Object Oriented Programming the instance variables of an object are the data items which that object uses.

Instantiation
the creation of a new object from a class.

Knowledge base
the facts and rules which constitute a program which uses the declarative programming paradigm.

LIFO
Last in First Out queue structure.

Linked list
A linked list is a list whose data items have a link to the next data item in the list, allowing it to expand and contract dynamically.

Methods
in object-oriented programming this refers to the behaviour of an object and how it can manipulate instance variables. Methods are defined inside the class which was used to create the object.

Modularity
where a program is constructed as a series of sub programs which are self-contained and can be re-used within that program.

Nested Loop
A computational construct consisting of a fixed loop within another fixed loop.
Object
In Object Oriented Programming an object is created from a class. An object inherits both Methods and Instance Variables from the Class which created it.

Overflow
The error when an attempt is made to push an item of data onto a stack or queue which is full.

Pop
To retrieve an item from a stack or queue.

Procedural programming
a programing paradigm where a series of commands are executed in sequence.

Programming Paradigm
a recognised style of computer programming.

Properties
In Object Oriented Programming the properties of an object are the instance variables or data items which that object uses.

Prototype
a working model designed for evaluation which may not have all the features of the final program.

Pseudocode
an informal high-level description of how a computer program functions.

Push
To place an item of data in a stack or queue.

Queue
An array of data items where items placed on a queue are added to the end and retrieved from the front. A queue operates on a "First in First Out" principle.

Race Condition
Where two threads simultaneously operate on the same data, but different timings result in unpredictable results.

Record
A record is a data structure which can contain data items of different types.

Recursion
a common declarative programming technique where code repeatedly calls itself until a simple base fact is identified.

Reference Language
a formally specified code for writing computer algorithms.
**Resource Starvation**
Where one thread in a concurrent program accesses a resource locking lower priority threads as a result.

**Screen reader**
software used by users who are blind or have impaired vision which reads the test on a web page or the interface of a program.

**Semantics**
the semantics of a block of pseudocode or program code is its meaning or logic. Computer code is semantically correct if it does what the programmer intended.

**Skeuomorphism**
when an interface mimics the hardware which an application is replacing.

**Stack**
An array of data items which can only be accessed from the top A stack operates on a "Last in First Out" principle.

**Stack pointer**
The variable storing the position of the empty space at the top of a stack.

**Stub**
a short section of code which stands in for a missing sub-program when testing a partially completed program.

**Syntax**
the syntax of a block of pseudocode or program code is the set of grammatical rules which it follows. Computer code is syntactically correct if it follows all the rules of the language.

**Thread**
a sequence of instructions running concurrently with another sequence.

**Top-down analysis**
breaking a problem down into smaller sub-problems in order to make it easier to build a solution.

**Underflow**
The error when an attempt is made to retrieve an item of data from an empty stack or queue.

**Waterfall Model**
the traditional software development model which assumes an iterative linear progression through the analysis, design, implementation, testing, documentation, evaluation, and maintenance stages.
Answers to questions and activities

1 Programming paradigms

Quiz: Revision (page 3)

Q1:  b) Declarative
Q2:  d) Object-oriented
Q3:  b) Declarative
Q4:
   a) Keyword highlighting
   b) Automatic indentation
   c) Search and replace
   d) Debugging tools
Q5:  d) A set of pre-tested classes which can be used in a program.
Q6:  b) Actual parameters
Q7:  a) Formal parameters

Activity: PlayList (page 15)

Expected answer

   1. DECLARE recentTracks INITIALLY PlayList[10]
   2. recentTracks.AddTrack(track{"Rockaway Beach", "The Ramones", 3.4})
   3. recentTracks.ShowTracks()
   4. myTracks.DeleteTrack(1)

Activity: Class (page 15)

Expected answer
CLASS Entry IS {STRING date, REAL amount}

METHODS

PROCEDURE showEntry()
   SEND THIS.date & " " & THIS.amount TO DISPLAY
END PROCEDURE

END CLASS

CLASS Statement IS {ARRAY OF Entry entries, INTEGER next, INTEGER maxnumber, REAL balance}

METHODS

CONSTRUCTOR Statement()

DECLARE THIS.balance INITIALLY 0
DECLARE THIS.next INITIALLY 0

END CONSTRUCTOR

PROCEDURE addEntry(Entry newEntry)
   IF THIS.next = THIS.maxnumber THEN

© HERIOT-WATT UNIVERSITY
<error - too many entries>

ELSE
   SET THIS.entries[next] TO newEntry
   SET THIS.next TO THIS.next + 1
   SET THIS.balance TO THIS.balance + newEntry.amount
END IF
END PROCEDURE

PROCEDURE displayStatement()
   FOR counter FROM 0 TO next - 1 DO
      THIS.entries[counter].showEntry()
   END FOR
END PROCEDURE

END CLASS

End of topic test (page 21)

Q8:
   b) Keeping variables local improves modularity of code.
   c) A variable is something whose value can be changed.

Q9:
   a) They underlie all programming languages.
   c) They can be combined to create more powerful computational constructs.

Q10:
   a) Modularity improves the readability of code.
   d) Formal parameters are those declared in the definition of a sub-program.

Q11: a) Encapsulation

Q12: a) A Class

Q13: b) Inheritance

Q14: x_pos, y_pos, and name
Q15:

Icon

INTEGER x_pos
INTEGER y_pos
STRING name

DesktopIcon

STRING imagePath

MoveTo (xpos, ypos)
Delete()

ShortcutIcon

STRING appPath

EntityIcon

STRING entityPath

Q16: b) Race Condition

Q17: a) Deadlock
2 Data types and structures

Quiz: Revision (page 26)

Q1:  h) ARRAY of CHARACTER
Q2:  j) ARRAY of STRING
Q3:  i) ARRAY of BOOLEAN
Q4:  j) ARRAY of STRING
Q5:  k) ARRAY of RECORD

Activity: Records and arrays of records (page 33)

Expected answer

Possible partial answer:

```plaintext
PROCEDURE main ()
    Setup (productInventory)

<ask manager for choice>

Display(productInventory)
END PROCEDURE

RECORD product IS{STRING productName, INTEGER stockNumber, REAL price, BOOLEAN reorder}

PROCEDURE setup (ARRAY OF Product productInventory )

SET productInventory [0] TO { productName = "USB cable", stockNumber = 624, price = 1.74, reorder = false}
SET productInventory [1] TO { productName = "HDMI adaptor", stockNumber = 523, price = 5.00, reorder = false}
SET productInventory [2] TO { productName = "DVD-RW pack", stockNumber = 124, price = 10.99, reorder = false}

END PROCEDURE

PROCEDURE display (ARRAY OF Product productInventory)

SEND "Items to be re-ordered" TO DISPLAY
FOR counter FROM 0 TO 2 DO
    IF productInventory.reorder = true THEN
        SEND productName[counter] TO DISPLAY
        <new line>
    END IF
END FOR
```
Activity: Push and Pop within a stack (page 39)

Q6:

Push A
Push B
Push C

Stack

Q7:

Note the new top of Stack pointer

Top of Stack

Bottom of Stack

Using this system the last number in is always the first number out. A stack is therefore called a **LIFO** structure (**Last In First Out**).

End of topic test (page 48)

Q8: An array, a stack pointer and a maximum stack size.

Q9: d) Front = 2, Rear = 6

Q10: d) 7

Q11: d) Front = 2, Rear = 4

Q12:

```
FOR row FROM 0 to 5 DO
    FOR column FROM 0 TO 7 DO
        SET stringArray[row,column] TO "password"
    END FOR
END FOR
```
END FOR

Q13:

FOR column FROM 0 to 7 DO
   SEND stringArray[2, column] to DISPLAY
END FOR
3 Development methodologies

Quiz: Revision (page 52)

Q1:  a) Programmers
Q2:  c) Editability
Q3:  d) A reduction in the need for teams to communicate about their activities.
Q4:  d) A high-level description of how a computer program functions.
Q5:  b) Analysis, design, implementation, testing, documentation, evaluation, maintenance.
Q6:  a) Systems Analyst

Activity: Sub tasks (page 54)

Expected answer

<table>
<thead>
<tr>
<th>School</th>
<th>Teacher</th>
<th>Pupil</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teacher[ ],</td>
<td>{TeacherID, Name,</td>
<td>(PupilID,</td>
<td>{CourseID,</td>
</tr>
<tr>
<td>Pupil[ ],</td>
<td>Classroom, Course[ ]}</td>
<td>Name}</td>
<td>Classroom, Teacher,</td>
</tr>
<tr>
<td>addPupil()</td>
<td>assignCourse()</td>
<td>showPupil()</td>
<td>Pupil[ ]}</td>
</tr>
<tr>
<td>addCourse()</td>
<td></td>
<td></td>
<td>assignPupil()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>removePupil()</td>
</tr>
</tbody>
</table>

End of topic test (page 58)

Q7:  b) Waterfall model
Q8:  c) A working model designed for evaluation.
Q9:  a) A development methodology which progresses in a series of stages, each one dependent on the successful completion of the previous one.
Q10: d) Testing of the finished project can be outsourced.
4 Design notations

Quiz: Revision (page 60)

Q1:  d) Top-down design
Q2:  c) Pseudocode
Q3:  d) Data flow diagram
Q4:  a) Wireframe
Q5:  c) Pseudocode
Q6:  b) breaking a large and complex problem into smaller, more manageable sub-problems until they become trivial.

Activity: Pseudocode (page 64)

Expected answer

Examples of two possible answers:

JavaScript:

```javascript
window.onload = function() {
    var clickMeButton = document.getElementById('button1');
    clickMeButton.onclick = fillarray;
    var clickMeButton = document.getElementById('button2');
    clickMeButton.onclick = linearSearch;
}

function linearSearch()
{
    var numbertofind = usernumber.value;
    var found = false;
    var i=0;
    while (!found & i<numericArray.length) {
        if (numericArray[i]==numbertofind){
            found = true;
            document.getElementById('printarea2').innerHTML = ("found at position "+ (i+1));
        }
        i++;
    }
    if (!found & i==numericArray.length) {
        document.getElementById('printarea2').innerHTML = ( numbertofind + " not found");
    }
}

function fillarray()
{  
    numericArray=[];
    for(var i=0; i < 10; i++) {
```
numericArray[i] = Math.floor((Math.random() * 10) + 1);
}
document.getElementById('printarea1').innerHTML = numericArray;
}

Visual Basic 6:

Dim numbers(10) As Integer
Private Sub CmdFill_Click()
Dim Counter as integer
For Counter = 1 To 10
    numbers(Counter) = Int(Rnd * 10) + 1
    Picture1.Print numbers(Counter);
Next Counter
End Sub

Private Sub cmdFind_Click()
Dim found As Boolean
Dim itemToFind As Integer
Dim found_at As Integer
itemToFind = Val(InputBox("What number do you want to find?"))
Do While Not found And Counter < 10
    If itemToFind = numbers(Counter) Then
        found = True
        Picture2.Print "Found at position "; Counter
    Else: Counter = Counter + 1
End If
Loop
End Sub

Activity: Pseudocode 2 (page 64)

Expected answer

PROCEDURE getCustomerChoice()
DECLARE answer AS STRING INITIALLY ""
FOR counter FROM 0 TO 2 DO
    SEND "Do you wish to purchase a " & shoppingBasket[counter].productName & " Enter Y or N" TO DISPLAY
    RECEIVE answer FROM KEYBOARD
    IF answer = "Y" OR answer = "y" THEN
        SET shoppingBasket[counter].purchased TO true
        SEND shoppingBasket[counter].productName & " added to shopping basket" TO DISPLAY
    END IF
END FOR
END PROCEDURE
Activity: Case and Class diagrams: Concert Booking (page 68)

Expected answer

The Event class is associated with the Venue class. The Ticket and the Band class are associated with the Event class.

Activity: Case and Class diagrams (page 71)

Expected answer
Use Case diagram:

User

- Add Entry
- Display Statement
- Show Entry

Use Class diagram:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance next</td>
<td>date</td>
</tr>
<tr>
<td>maxnumber entries[]</td>
<td>amount</td>
</tr>
<tr>
<td>addEntry() displayStatement()</td>
<td>showEntry()</td>
</tr>
</tbody>
</table>

End of topic test (page 75)

Q7:  d) Pseudocode is converted to source code but cannot always be directly translated into source code.

Q8:  d) Syntax checker

Q9:  a) Natural language segment of pseudocode which needs to be expanded.

Q10: a) A person or entity that interacts with the system.

Q11: c) Relationships between classes.

Q12: a) How actors interact with the system.

Q13: 
  b) Prototype design
  c) Interface Design
5 Standard algorithms

Quiz: Revision (page 79)

Q1:  b) Input validation.
Q2:  d) Finding the maximum.
Q3:  b) A conditional loop and a boolean variable.
Q4:  a) A fixed loop.
Q5:  a) Counting occurrences.

Activity: Binary search algorithm - Trace table (page 84)

Expected answer

searchKey = 9

<table>
<thead>
<tr>
<th>found</th>
<th>Middle</th>
<th>StartPos</th>
<th>endPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>true</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

"Found at position 9"

searchKey = 11

<table>
<thead>
<tr>
<th>found</th>
<th>Middle</th>
<th>StartPos</th>
<th>endPos</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>9</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>false</td>
<td>10</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

"Not found"

Activity: Binary search algorithm - Programming (page 84)

Expected answer

Visual Basic 6

Dim myArray(10) As Integer

Private Sub binarySearch()
    Dim found As Boolean

Dim startpos As Integer
Dim endpos As Integer
Dim key as Integer

found = False
startpos = 0
dpos = 10
key = InputBox("Please enter number to find")

Do
middle = (startpos + endpos) \ 2
If myarray(middle) < Key Then
startpos = middle + 1
Else: endpos = middle - 1
End If
If myarray(middle) = Key Then
found = True
MsgBox ("found at " & middle)
End If
Loop Until (found = True) Or startpos > endpos
If found = False Then MsgBox ("not found")

End Sub

Activity: Bubble sort 2 (page 97)

Expected answer
Possible solution:

```
PROCEDURE bubblesort(ARRAY OF INTEGER list)

DECLARE listlength INITIALLY length (list)
DECLARE sorted AS BOOLEAN INITIALLY false
DECLARE outerloop INITIALLY 0
DECLARE swaps INITIALLY 0

WHILE sorted = false DO
    SET outerloop TO listlength
    SET swaps TO 0
    FOR counter FROM 0 TO outerloop 2 DO
        IF list[counter] > list[counter+1] THEN
            swap(list[counter], list[counter+1])
        SET swaps TO swaps + 1
    END IF
    END FOR
    IF swaps = 0 THEN
        SET sorted TO true
    END IF
    SET outerloop TO outerloop -1
END WHILE

END PROCEDURE

PROCEDURE swap(REF a, REF b)

DECLARE temp AS INTEGER INITIALLY a
SET a TO b
SET b TO temp

END PROCEDURE
```

Activity: Bubble sort 3 (page 97)

Expected answer

Possible solution:

```
PROCEDURE bubblesort(ARRAY OF INTEGER list)

DECLARE listlength INITIALLY length (list)
DECLARE swapCount INITIALLY 0
DECLARE swaps INITIALLY 0
DECLARE passes INITIALLY 0
DECLARE comparisons INITIALLY 0
```
DECLARE sorted AS BOOLEAN INITIALLY false
DECLARE outerLoop INITIALLY 0

WHILE sorted = false DO
    SET outerLoop TO listlength
    SET swaps TU 0
    FOR counter FROM 0 TO outerLoop 2 DO
        SET comparisons TO comparisons + 1
        IF list[counter] > list[counter+1] THEN
            swap(list[counter], list[counter+1])
            SET swaps TU swaps + 1
            SET swapCount TO swapCount + 1
        END IF
    END FOR
    IF swaps = 0 THEN
        SET sorted TO true
    END IF
    SET outerLoop TO outerLoop -1
    SET passes TO passes + 1
END WHILE
SEND "Comparisons: " & comparisons TO DISPLAY
SEND "Passes: " & passes TO DISPLAY
SEND "swaps: " & swapCount TO DISPLAY

END PROCEDURE

Possible solution in Visual Basic 6:

Dim list(10) As Integer
Dim swapcount As Integer
Dim passes As Integer
Dim comparisons As Integer

Private Sub bubblesort()
    Dim Sorted as Boolean
    Dim outerloop as Integer

    swapcount = 0
    passes = 0
    comparisons = 0

    Sorted = False
    Do While Sorted = False
        outerloop = 10
        swaps = 0
        For Counter = 0 To outerloop - 1
            comparisons = comparisons + 1
            If list(Counter) > list(Counter + 1) Then
                Swap list(Counter), list(Counter + 1)
            End If
        Next
        If swaps = 0 Then
            Sorted = True
        End If
    Next
End Sub
swaps = swaps + 1
swapcount = swapcount + 1
End If
Next Counter
If swaps = 0 Then Sorted = True
outerloop = outerloop - 1
passes = passes + 1
Loop

Private Sub swap(ByRef a, ByRef b)
temp = a
a = b
b = temp
End Sub

Private Sub cmdsort_Click()
Call bubblesort
Dim Counter As Integer
For Counter = 0 To 10
Picture2.Print list(Counter)
Next Counter
Picture3.Print "passes: "; passes
Picture3.Print "swaps: "; swapcount
Picture3.Print "comparisons: "; comparisons
End Sub

Private Sub fillarray_Click()
Randomize
For Counter = 0 To 10
' list(Counter) = Counter 'sorted list
list(Counter) = Int((Rnd * 10) + 1) ' unsorted list
Picture1.Print list(Counter)
Next Counter
End Sub

Activity: Insertion sort 2 (page 100)

Expected answer

Possible answer:

PROCEDURE insertionSort(ARRAY OF INTEGER list)

DECLARE swapcount INITIALLY 0
DECLARE passes INITIALLY 0
DECLARE comparisons INITIAL 0

FOR outerloop FROM 1 TO listLength DO
    FOR innerloop FROM outerloop TO 1 STEP -1 DO
        SET comparisons TO comparisons + 1
        IF (list[innerloop -1] > list[innerloop]) THEN
            swap(list[innerloop -1] , list[innerloop])
        END IF
        SET swapcount TO swapcount + 1
    END FOR
    SET passes TO passes + 1
END FOR

SEND "Comparisons: " & comparisons TO DISPLAY
SEND "Passes: " & passes TO DISPLAY
SEND "swaps: " & swapcount TO DISPLAY

END PROCEDURE

PROCEDURE swap(REF a, REF b)

DECLARE temp AS INTEGER INITIAL 0
SET a TO b
SET b TO temp

END PROCEDURE

End of topic test (page 104)

Q6: Linear search is slower than a binary search. Linear search can operate on unordered lists. Binary lists must be ordered. Binary search algorithm is more complex.

Q7: b) 500

Q8: a) 10

Q9: b) Selection sort

Q10: d) 9900

Q11: c) 11

Q12: c) Quicksort

Q13: a) 3

Q14: c) Bubble sort
6 Computational constructs

Activity: Recursive function - Factorial (page 109)

Expected answer

This is an example of the algorithm implemented in Prolog.

```prolog
factorial(1,1).

factorial(Num,Answer):-
    Nextnum is Num - 1,
    factorial(Nextnum,Nearly_answer),
    Answer is Num * Nearly_answer.
```

Activity: Recursive function - Sum of numbers (page 109)

Expected answer

Pseudocode answer:

```pseudocode
FUNCTION sumNums(INTEGER number) RETURNS INTEGER

    IF number = 1 THEN
        RETURN number
    ELSE
        RETURN sumNums(number-1) + number
    END IF

END FUNCTION
```

Possible Prolog answer:

```prolog
sumnums(1,1).
sumnus(Number,Sum):-
    Newnumber is Number - 1,
    sumnums(Newnumber, SmallerSum),
    Sum is SmallerNumber + Number.
```

Possible Visual Basic answer:

```vbnet
Private Sub getAnswer_Click()
    Value = InputBox("enter number to be summed")
    answer = sumnums(Value)
    Print answer
End Sub

Function sumnums(number)
    If number = 1 Then
```
sumnums = 1
Else: sumnums = sumnums(number - 1) + number
End If
End Function

Activity: Algorithm for a string function (page 110)

Expected answer
Possible answer:

FUNCTION binaryString(INTEGER number) RETURNS STRING

DECLARE newDigit INITIALLY 0
DECLARE smallerNumber INITIALLY 0
DECLARE newString INITIALLY ""
IF number = 1 THEN
RETURN "1"
ELSE
SET newDigit TO asciiValue(number MOD 2)
SET smallerNumber TO number \ 2 #integer division
SET newString TO binaryString(smallerNumber)
RETURN newString & newDigit

END FUNCTION

FUNCTION asciiValue(INTEGER number) RETURNS STRING
DECLARE newString INITIALLY ""
<convert single digit to ASCII value of number as newString >
RETURN newString
END FUNCTION

Possible Visual Basic 6 solution:

Private Sub GetBinary_Click()
    Dim Value as integer
    Value = InputBox("enter a number to convert")
    Picture1.Print binaryString(Value)
End Sub

Function binaryString(number)
If number = 1 Then
    binaryString = "1"
Else:
    newdigit = number Mod 2
    newnumber = number \ 2
    binaryString = binaryString(newnumber) + Str$(newdigit)
End If
End Function
Activity: Sequential files (page 113)

Expected answer

Possible answer using Visual Basic:

Private Sub CmdWriteToFile_Click()
    Filename = inputbox ("Please enter filename")
    Open Filename For Output As #1
    Dim row as integer
    For row = 0 To 9
        Print #1, stringarray(row)
    Next row
    Close #1
End Sub

Private Sub CmdReadFromFile_Click()
    Filename = inputbox ("Please enter filename")
    Open Filename For Input As #1
    Dim row as integer
    For row = 0 To 9
        Input #1, stringarray(row)
    Next row
    Close #1
End Sub

Activity: Sequential files 2 (page 113)

Expected answer

PROCEDURE main()
    DECLARE numbers AS ARRAY OF INTEGER INITIALLY []
    DECLARE sorted INITIALLY "n:\sorted.txt"
    DECLARE unsorted INITIALLY "n:\unsorted.txt"
    readFile(numbers, unsorted)
    bubbleSort(numbers)
    writeFile(numbers, sorted)
END PROCEDURE

Reading a file into 2D array:

PROCEDURE readFile(ARRAY OF ARRAY OF INTEGER maze)
    DECLARE myfile INITIALLY " N:\myfile.txt"
    DECLARE numberString INITIALLY ""
    OPEN myfile
    FOR row FROM 0 TO 9 DO
        FOR column FROM 0 TO 3 DO
            RECEIVE numberString FROM myfile
        NEXT column
    NEXT row
END PROCEDURE
SET maze[row][column] TO asciToInteger(numberString)
END FOR
END FOR
CLOSE myfile
END PROCEDURE

FUNCTION asciToInteger(STRING characters) RETURNS INTEGER
DECLARE value AS INTEGER INITIALLY 0
<convert ASCII character(s) to integer value>
RETURN value
END FUNCTION

End of topic test (page 115)

Q1: b) Recursion is where a sub-program repeatedly calls itself.

Q2: b) The program will continually call itself until running out of memory.

Q3:
a) They cannot be read and written to simultaneously true.
b) Reading a sequential file always starts at the beginning true.

Q4:
a) Storing the readings from a temperature gauge at 10 minute intervals sequential
b) Storing an array of records containing names addresses and telephone numbers random

Q5: a) The position of an item can be calculated from its index.
7 Testing and documenting solutions

Quiz: Revision (page 118)

Q1:

<table>
<thead>
<tr>
<th>minimumValue</th>
<th>counter</th>
<th>numbers[counter]</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Output: The smallest value was 4.

Q2: c) extreme data.

Q3: b) False

Q4: a) True

Activity: W3C Web Content Accessibility Guidelines (page 123)

Expected answer

- Availability of text alternatives
- Adaptability
- Ease of navigations
- Keyboard accessibility
- Readability
- Predictability
- Availability of help.

End of topic test (page 124)

Q5: b) The test data used.

Q6: Documentation will be invaluable to those testing subsequent stages and will also aid anyone who needs to re-visit a previous stage in the light of subsequent previously undetected problems.

Q7: a) A dummy function or procedure.

Q8: b) Code designed to test a function or procedure.


Q10: b) Beta testing
Q11: a) There should always be an option to escape from a crucial operation.

Q12: The ability to change colour schemes.

Q13: Users with epilepsy can have an attack triggered by flashing colours or flickering images.
8 End of Unit 1 test

End of Unit 1 test (page 128)

Q1:
   a) Modularity improves the readability.
   c) Modularity improves portability.
   d) Modularity is a consequence of top down analysis.

Q2:  a) encapsulation

Q3:  a) a class

Q4:  b) inheritance

Q5:  b) race condition

Q6:  a) deadlock

Q7:
   • An array (the array type depends on the data in the queue),
   • the start_pointer,
   • the rear_pointer,
   • the current_queue_size and
   • the maximum_queue_size (all integer variables).

Q8:  d) Front = 2, Rear = 6

Q9:  b) Waterfall model

Q10: c) A working model designed for the purpose of evaluation.

Q11: b) 50

Q12: a) 10

Q13: b) Selection sort

Q14: d) 9900

Q15: c) 55

Q16: c) Bubble sort

Q17: b) Recursion is where a sub-program repeatedly calls itself.

Q18:
   a) They cannot be read and written to simultaneously.
   b) Reading a sequential file always starts at the beginning true.
Q19:
  a) Storing the readings from a humidity gauge at 15 minute intervals.  Sequential
  b) Storing an array of records containing names and scores.  Random

Q20: a) stub

Q21: b) driver

Q22:
  a) Developers
  b) Programmers

Q23: b) Beta testing

Q24: a) If number = 1 Then sumnums = 1

Q25: d) Line 4

Q26: b) Line 2

Q27: c) 5

Q28: a) controllable

Q29: c) epilepsy